Computer Graphics and Geometric Modeling

Max K. Agoston

Computer Graphics and Geometric Modeling

Implementation and Algorithms

Springer

Max K. Agoston, MA, MS, PhD
Cupertino, CA 95014, USA

British Library Cataloguing in Publication Data
Agoston, Max K.
Computer graphics and geometric modeling:implementation & algorithms
1. Computer graphics 2. Geometry—Data processing 3. Computer-aided design
4. Computer graphics—Mathematics I. Title
006.6

ISBN 1852338180

Library of Congress Cataloging-in-Publication Data

Agoston, Max K.
Computer graphics & geometric modeling/Max K. Agoston.
p. cm.
Includes bibliographical references and index.
Contents: Implementation & algorithms
ISBN 1-85233-818-0 (v. 1 : alk. paper)
1. Computer graphics. 2. Geometry—Data processing. 3. Mathematical models. 4. CAD/CAM systems. I. Title.

T385.A395 2004
006.6—dc22 2004049155

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

ISBN 1-85233-818-0
Springer is part of Springer Science+Business Media
springeronline.com

© Springer-Verlag London Limited 2005
Printed in the United States of America

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Typesetting: SNP Best-set Typesetter Ltd., Hong Kong
34/3830-543210 Printed on acid-free paper SPIN 10971451

Preface

This book and [AgoM05] grew out of notes used to teach various types of computer graphics courses over a period of about 20 years. Having retired after a lifetime of teaching and research in mathematics and computer science, I finally had the time to finish these books. The two books together present a comprehensive overview of computer graphics as seen in the context of geometric modeling and the mathematics that is required to understand the material. Computer graphics itself is a multifaceted subject, but it has grown up. It is no longer necessary that a book on graphics demonstrate the diversity of the subject with a long list of "fun" projects at the expense of the mathematics. From movies, television, and other areas of everyday life, readers have already seen what graphics is about and what it can do. It follows that one should be able to present the geometric modeling aspect of the subject in a systematic fashion. Unfortunately, the sheer amount of material that I wanted to cover meant that it had to be divided into two parts. This book contains the practical stuff and describes the various algorithms and implementation issues that one runs into when writing a geometric modeling program. The book [AgoM05] provides the mathematical background for the underlying theory. Although each book can be read by itself without reading the other, one will get the most benefit from them if they are read in parallel.

The intended audience of this book (and the combined two volumes especially) is quite broad. It can be used in a variety of computer graphics courses or by those who are trying to learn about graphics and geometric modeling on their own. In particular, it is for those who are getting involved in what is referred to as computer-aided design (CAD) or computer-aided geometric design (CAGD), but it is also for mathematicians who might want to use computers to study geometry and topology. Both modeling and rendering issues are covered, but the emphasis is on the former. The basic prerequisites are that the reader has had an upper division data structure course, minimally three semesters of calculus, and a course on linear algebra. An additional course on advanced calculus and modern algebra would be ideal for some of the more advanced topics. On the companion CD there is a geometric modeling program (GM) that implements many of the algorithms discussed in the text and is intended to provide a programming environment both for further experimentation and application development. Another program (SPACE) on the CD is an application that uses some of the more advanced geometric modeling concepts to display the intrinsic

geometry of two- and three-dimensional manifolds. Both programs were written using the Microsoft Visual C++ compiler (and OpenGL) and run under Microsoft Windows 98 or later. Their source code and documentation are included on the CD. The ReadMe file on the CD lists what all is on the CD and also contains instructions for how to use what is there.

As I began to develop this book on geometric modeling, one concern obviously was to do a good job in presenting a thorough overview of the practical side of the subject, that is, the algorithms and their implementation details. However, there were two other goals that were important from the very beginning. One was to thoroughly explain the mathematics and the other, to make the material as self-contained as possible. In other words, pretty much every technical term or concept that is used should be defined and explained. The reason for putting all the computer graphics–related material into one book and all the mathematics into the other rather than interweaving the material was to keep the structure of the implementation of a modeling program as clear as possible. Furthermore, by separating out the mathematics it is easier for readers to skip those mathematical topics that they are already familiar with and concentrate on those with which they are not. In general, though, and in particular as far as instructors using this book are concerned, the intent is that the material in the two books be covered in parallel. This is certainly how I always taught my courses. An added motivation for the given division was that the applied part of geometric modeling was often a moving target because, largely due to improvements in hardware (faster CPUs, more memory, more hard disk space, better display devices), the way that one deals with it is changing and will continue to change in the future. This is in contrast to the supporting mathematics. There may be new mathematics relevant to computer graphics in the future but it will be a long time before the mathematics I do discuss will lose its relevance. A lot of it, in fact, is only now starting to be used as hardware becomes capable of dealing with computationally expensive algorithms.

One property that sets this book apart from others on geometric modeling is its breadth of coverage, but there is another. The combined books, this one and [AgoM05], differ from other books on computer graphics or geometric modeling in an important way. Some books concentrate on implementation and basically add the relevant mathematics by tossing in appropriate formulas or mathematical algorithms. Others concentrate on some of the mathematical aspects. I attempt to be as comprehensive on both the implementation and theory side. In [AgoM05] I provide a **complete** reference for all the relevant mathematics, but not in a cookbook fashion. A fundamental guiding principle was to present the mathematics in such a way that the reader will see the motivation for it and **understand** it. I was aiming at those individuals who will want to take the subject further in the future and this is not possible without such understanding. Just learning a few formulas is not good enough. I have always been frustrated by books that throw the reader some formulas without explaining them. Furthermore, the more mathematics that one knows, the less likely it is that one will end up reinventing something. There are instances (such as in the case of the term "geometric continuity") where unfamiliarity with what was known caused the introduction of confusing or redundant terminology. The success or failure of the two books should be judged on how much understanding of the mathematics and algorithms the reader gets. In the case of this book by itself, it is a question of whether or not the major topics were covered adequately. In any case, because I

emphasize understanding what is going on, there is a natural emphasis on theory and not on tricks of the trade. The reader will also not find any beautiful glossy pictures.

Clearly, no one book can cover all that falls under the general heading of geometric modeling. As usual, the topics that are in fact covered and the degree to which they are covered reflect my own bias. In a large field, there are many special topics and it should not be surprising that some are not discussed at all and others only briefly in an overview. On the other hand, one would expect to see a discussion of principles and issues that are basic to the field as a whole. In that regard, I would like to alert the reader to one topic, namely, the issue of robustness of algorithms and computations, that really is a central issue in geometric modeling, but is not dealt with as thoroughly as it should be, given its importance. The only excuse for this is that to do this topic justice would have entailed a much larger book. It is important that readers who want to do serious work in geometric modeling realize that they will have to get more information elsewhere on this. The discussion in Section 5.10 is inadequate (although I do devote the brief Chapter 18 to interval analysis). When it comes to the bibliography, as large as it is, it is also a victim of space constraints. In some cases I have tried to compensate for the lack of completeness by giving references to books or papers where additional references could be found.

Most of this book covers material that is not new, but a few algorithms have not appeared in print before. One is the approach to trimmed surfaces based on the Vatti clipping algorithm described in Section 14.4. Another is the result in Section 17.5 about convex set intersections, which motivated the algorithm described in Section 13.2. Another aspect of the book that should be noted is Chapter 16 and the SPACE program. Although the material on intrinsic geometry in Sections 16.3 and 16.4 did not get developed as much as I would have liked, it is a start. The extensive material on topology in [AgoM05], in particular algebraic and differential topology, has heretofore not been found in books on geometric modeling. Although this subject is slowly entering the field, its coming has been slow. Probably the two main reasons for this are that computers are only now getting to be powerful enough to be able to handle the complicated computations and the material involves exceptionally advanced mathematics that even mathematics majors would normally not see until graduate school.

Here is how the material in this book has been used successfully in teaching three different types of semester courses on computer graphics in the Department of Mathematics and Computer Science at San Jose State University. The courses were

(1) Introduction to Computer Graphics,
(2) Computer Graphics Algorithms, and
(3) Geometric Modeling.

The first two were upper-division undergraduate courses and the third was a graduate course. The prerequisites for the introductory course were three semesters of calculus, linear algebra, and an upper division course in data structures. The only prerequisite to both the algorithm and geometric modeling course was the introductory computer graphics course. Some of the material in the introductory course was briefly reviewed in the other two courses. The courses used material from the following parts of this book and [AgoM05] (but the material was not necessarily dis-

cussed in the order listed, and listing a chapter or section in no way means that all of it was covered):

Introduction to Computer Graphics:	Chapters 1–4, a quick overview of Chapters 5, 6, 11, 12, and a brief discussion of visible surface algorithms and shading from Chapters 7 and 10. From [AgoM05]: Chapters 1–3.
Computer Graphics Algorithms:	Chapters 2–10, just enough of Chapter 12 to have surfaces to render, Sections 21.6–21.8, and Chapter 22. From [AgoM05]: Chapter 1 and Sections 4.5, 4.7, 8.1–8.5.
Geometric Modeling:	Chapters 3–6, 11, 12, a sampling of topics from Chapters 13–15, and Sections 17.4–17.5. From [AgoM05]: Review of parts of Chapters 1 and 2, Sections 4.2, 4.5, 4.7, Chapter 6, and Sections 8.1–8.5, 9.2–9.4, 9.9.

The numbering of items in this book uses the following format: x.y.z refers to item number z in section y of chapter x. For example, Theorem 12.7.1 refers to the first item of type theorem, proposition, lemma, or example in Section 7 of Chapter 12. Algorithm 14.3.1 refers to the first algorithm in Section 3 of Chapter 14. Tables are numbered like algorithms. Figures are numbered by chapter, so that Figure 14.7 refers to the seventh figure in Chapter 14. Exercises and programming projects at the end of chapters are numbered by section.

Finally, some comments on the language used in this book to describe algorithms. Even though the C/C++ language is the language of choice for most people writing computer graphics programs, with the exception of some initialization code found in Section 1.6, we have opted to use "the" more universal "standard" algorithmic language. The reason is that this book is mostly about concepts that are independent of any particular programming language and low-level optimization issues that hinge on the language being used do not play any role. Every reader with some general computer experience will understand the language used here (a detailed description of its syntax can be found in Appendix B) and so there seemed to be little point to restrict the audience to those familiar with C. Consider the following points:

(1) There is little difference between the code that is presented and what the corresponding C code would look like, so that any translation would be straightforward.

(2) The emphasis in the code and algorithms in this book is on **clarity** and the fact is that even in simple constructs like a for-loop or a case statement, C has more complicated syntax and uses more obscure terminology which would make it harder for the non-C reader to understand. A certain universality would be lost with no real corresponding gain. The efficiency advantage of C that is usually cited is only really

significant in a relatively small number of places. It would be relevant, for example, if one wanted to implement low level drawing primitives, but that is not what this book is about.

(3) C programmers who want to see C code can look at the GM and SPACE programs, which are written in C++.

Cupertino, California Max K. Agoston

Contents

Preface . . . . . **v**

I Basic Computer Graphics **1**

1 Introduction **3**
1.1 Overview 3
1.2 The Basic Graphics Pipeline 4
1.3 Hardware Basics 7
1.4 Graphics Standards and Primitives 10
1.5 From Window to Viewport 12
1.6 Programming Notes 13
1.7 Exercises 16
1.8 Programming Projects 16

2 Raster Algorithms **22**
2.1 Introduction 22
2.2 Discrete Topology 22
2.3 Border Following Algorithms 26
2.4 Fill Algorithms 28
2.5 Generating Discrete Curves 35
2.5.1 Digital Differential Analyzers 36
2.5.2 The Bresenham Line-Drawing Algorithm 38
2.5.3 The Midpoint Line-Drawing Algorithm 40
2.6 The Aliasing Problem 44
2.7 Halftoning, Thresholding, and Dithering 48
2.8 Choosing the Coordinates of a Pixel 49
2.9 More Drawing Algorithms 49
2.9.1 Scan Converting Polygons 49
2.9.2 Drawing Circles 54
2.9.3 Drawing Ellipses and Other Conics 57
2.10 Bit Map Graphics 59
2.11 2D Animation 61

2.12 Exercises 66
2.13 Programming Projects 67

3 Clipping 69
3.1 Introduction 69
3.2 Line Clipping Algorithms 71
3.2.1 Cohen-Sutherland Line Clipping 71
3.2.2 Cyrus-Beck Line Clipping 73
3.2.3 Liang-Barsky Line Clipping 77
3.2.4 Nicholl-Lee-Nicholl Line Clipping 81
3.3 Polygon Clipping Algorithms 84
3.3.1 Sutherland-Hodgman Polygon Clipping 84
3.3.2 Weiler Polygon Clipping 85
3.3.3 Liang-Barsky Polygon Clipping 86
3.3.4 Maillot Polygon Clipping 89
3.3.5 Vatti Polygon Clipping 98
3.3.6 Greiner-Hormann Polygon Clipping 106
3.4 Text Clipping 109
3.5 Exercises 110
3.6 Programming Projects 110

4 Transformations and the Graphics Pipeline 111
4.1 Introduction 111
4.2 From Shape to Camera Coordinates 112
4.3 Vanishing Points 117
4.4 Windows and Viewports Revisited 119
4.5 The Clip Coordinate System 122
4.6 Clipping 125
4.7 Putting It All Together 130
4.8 Stereo Views 131
4.9 Parallel Projections 132
4.10 Homogeneous Coordinates: Pro and Con 134
4.11 The Projections in OpenGL 138
4.12 Reconstruction 139
4.13 Robotics and Animation 141
4.14 Quaternions and In-betweening 146
4.15 Conclusions 149
4.16 Exercises 151
4.17 Programming Projects 152

5 Approaches to Geometric Modeling 156
5.1 Introduction 156
5.2 R-sets and Regularized Set Operators 158
5.3 Representation Schemes 160
5.3.1 Early Representation Schemes 164

5.3.2 Boundary Representations 166
5.3.3 The CSG Representation 167
5.3.4 Euler Operations 171
5.3.5 Sweep Representations and Generative Modeling 174
5.3.6 Parametric Representations 178
5.3.7 Decomposition Schemes 178
5.3.8 Volume Modeling 180
5.3.9 The Medial Axis Representation 182
5.4 Modeling Natural Phenomena 188
5.5 Physically Based Modeling 190
5.6 Parametric and Feature Based Modeling 192
5.7 Functions and Algorithms 198
5.8 Data Structures 199
5.8.1 Data Structures for Boundary Representations 199
5.8.2 Data Structures for Volume Modeling 203
5.9 Converting Between Representations 205
5.10 Round-off Error and Robustness Issues 211
5.11 Algorithmic Modeling 215
5.12 Conclusions 220
5.13 Exercises 225

6 Basic Geometric Modeling Tools 227
6.1 Introduction 227
6.2 Bounding Objects and Minimax Tests 227
6.3 Surrounding Tests 232
6.4 Orientation Related Facts 238
6.5 Simple Intersection Algorithms 240
6.6 Distance Formulas 245
6.7 Area and Volume Formulas 249
6.8 Circle Formulas 252
6.9 Parametric or Implicit: Which Is Better? 258
6.10 Transforming Entities 259
6.11 Exercises 261
6.12 Programming Projects 262

7 Visible Surface Algorithms 264
7.1 Introduction 264
7.2 Back Face Elimination 267
7.3 The Schumacker List Priority Algorithm 268
7.4 Newell-Newell-Sancha Depth Sorting 269
7.5 The BSP Algorithm 270
7.6 Warnock and Weiler-Atherton Area Subdivision 273
7.7 Z-buffer Algorithms 275
7.8 The Watkins Scan Line Algorithm 278
7.9 Octree Algorithms 283

7.10 Curved Surface Algorithms 284
7.11 Adding Antialiasing 290
7.12 Conclusions 291
7.13 Programming Projects 293

8 Color 294
8.1 Introduction 294
8.2 What Is Color? 294
8.3 Perceived Color 295
8.4 Colorimetry 297
8.5 Color Models 299
8.6 Transforming Between Color Models 303
8.7 Programming Projects 307

9 Illumination and Shading 308
9.1 Introduction 308
9.2 Local Reflectance Models 310
9.3 Simple Approaches to Shading 316
9.4 Global Illumination Models 318
9.4.1 Shadows 318
9.4.2 Transparency 320
9.4.3 Ray Tracing 322
9.4.4 Radiosity Methods 323
9.5 The Rendering Equation 323
9.6 Texture and Texture Mappings 324
9.7 Environment Mappings 327
9.8 Bump Mappings 328
9.9 The Rendering Pipeline 330
9.10 Selecting a Color Palette 332
9.11 Programming Notes 333
9.12 Programming Projects 335

10 Rendering Techniques 337
10.1 Introduction 337
10.2 Ray Tracing 338
10.2.1 A Ray-Tracing Program 338
10.2.2 Ray Intersection Formulas 344
10.2.3 Ray Tracing CSG objects 348
10.3 The Radiosity Method 350
10.3.1 Form Factors: The Hemicube Method 355
10.4 Volume Rendering 358
10.4.1 Discrete Three-Dimensional Lines 362
10.4.2 The Marching Cubes Algorithm 365
10.5 Exercises 369
10.6 Programming Projects 369

II Geometric Modeling Topics **371**

11 Curves in Computer Graphics **373**
11.1 Introduction to Curves and Surfaces 374
11.2 Early Historical Developments 378
11.2.1 Lagrange Interpolation 378
11.2.2 Hermite Interpolation 381
11.2.3 Spline Interpolation 387
11.3 Cubic Curves 390
11.4 Bézier Curves 396
11.5 B-Spline Curves 404
11.5.1 The Standard B-Spline Curve Formulas 404
11.5.2 The Multiaffine Approach to B-Splines 418
11.5.3 Rational B-spline Curves 430
11.5.4 Efficient B-spline and NURBS Curve Algorithms 436
11.5.5 B-Spline Interpolation 441
11.6 Nonlinear Splines 445
11.7 Superellipses 448
11.8 Subdivision of Curves 449
11.9 Composition of Curves and Geometric Continuity 452
11.10 The Shape of a Curve 456
11.11 Hodographs 459
11.12 Fairing Curves 460
11.13 Parallel Transport Frames 461
11.14 Recursive Subdivision Curves 465
11.15 Summary 466
11.16 Exercises 468
11.17 Programming Projects 470

12 Surfaces in Computer Graphics **472**
12.1 Introduction 472
12.2 Surfaces of Revolution 474
12.3 Quadric Surfaces and Other Implicit Surfaces 480
12.4 Ruled Surfaces 482
12.5 Sweep Surfaces 484
12.6 Bilinear Surfaces 486
12.7 Coons Surfaces 487
12.8 Tensor Product Surfaces 495
12.9 The Bicubic Patch 496
12.10 Bézier Surfaces 500
12.11 Gregory Patches 502
12.12 B-spline Surfaces 504
12.12.1 The Basic B-spline Surface 504
12.12.2 Polynomial Surfaces and Multiaffine Maps 505
12.12.3 Triangular Bézier Surfaces 509
12.12.4 Rational B-spline Surfaces 512

12.12.5 Efficient B-spline and NURBS Surface Algorithms 514
12.12.6 B-spline Interpolation 516
12.13 Cyclide Surfaces 517
12.14 Subdivision of Surfaces 521
12.15 Composite Surfaces and Geometric Continuity 522
12.16 Fairing Surfaces 525
12.17 Recursive Subdivision Surfaces 526
12.18 Summary for Curves and Surfaces 530
12.19 A Little Bit of History 532
12.20 Exercises 534
12.21 Programming Projects 536

13 Intersection Algorithms 537
13.1 Overview 537
13.2 Convex Set Intersections 540
13.3 Curve Intersections 543
13.3.1 Ray-Curve Intersection 543
13.3.2 Curve-Curve Intersections 545
13.3.3 A Curve Newton-Raphson Method 546
13.3.4 Curve Recursive Subdivision Methods 547
13.3.5 Curve Algebraic Methods 551
13.4 Special Surface Intersections 552
13.4.1 Ray-Surface Intersections 552
13.4.2 Curve-Surface Intersections 552
13.4.3 Surface Sections 553
13.5 Surface-Surface Intersections 557
13.5.1 Surface Lattice Evaluation Methods 558
13.5.2 Surface Marching Methods 558
13.5.3 Surface Homotopy Method 570
13.5.4 Surface Recursive Subdivision Methods 572
13.5.5 Surface Algebraic Methods 574
13.6 Summary 578
13.7 Programming Projects 580

14 Global Geometric Modeling Topics 582
14.1 Overview 582
14.2 Distance Algorithms 582
14.3 Polygonizing Curves and Surfaces 587
14.4 Trimmed Surfaces 598
14.5 Implicit Shapes 614
14.5.1 Implicit Curves 614
14.5.2 Implicit Surfaces and Quadrics 622
14.6 Finding Contours 624
14.7 Skinning 630
14.8 Computing Arc Length 633
14.9 Offset Shapes 638
14.9.1 Offset Curves 638
14.9.2 Offset Surfaces 644

14.10 Envelopes 646
14.11 Exercises 647
14.12 Programming Projects 647

15 Local Geometric Modeling Topics 649
15.1 Overview 649
15.2 Curvature 649
15.3 Geodesics 652
15.3.1 Generating Smooth Geodesics 652
15.3.2 Generating Discrete Geodesics 657
15.4 Filament Winding and Tape Laying 667
15.5 Dropping Curves on Surfaces 670
15.6 Blending 672
15.7 Programming Projects 683

16 Intrinsic Geometric Modeling 684
16.1 Introduction 684
16.2 Virtual Reality 685
16.3 Geometrically Intelligent Modeling Systems 687
16.4 Exploring Manifolds 689
16.5 Where To From Here? 693

III More on Special Computer Graphics Topics 695

17 Computational Geometry Topics 697
17.1 Introduction 697
17.2 Range Queries 697
17.3 Interval and Segment Trees 703
17.4 Box Intersections 709
17.5 Convex Set Problems 711
17.6 Triangulating Polygons 714
17.7 Voronoi Diagrams 720
17.8 Delaunay Triangulations 722

18 Interval Analysis 726
18.1 Introduction 726
18.2 Basic Definitions 727
18.3 Inclusion Functions 731
18.4 Constraint Solutions 735
18.5 An Application: Implicit Curve Approximations 738
18.6 Constrained Minimizations 742
18.7 Conclusions 744
18.8 Exercises 744

19 The Finite Element Method 745
19.1 Introduction 745
19.2 What Is It All About? 745

19.3 The Mathematics Behind FEM . . . 747
19.4 An Example . . . 749
19.5 Summary . . . 753

20 Quaternions . . . 755
20.1 Introduction . . . 755
20.2 Basic Facts . . . 755
20.3 Quaternions as Transformations . . . 760
20.4 Exercises . . . 766

21 Digital Image Processing Topics . . . 767
21.1 Introduction . . . 767
21.2 The Ubiquitous Laplace Equation . . . 768
21.3 From Laplace to Fourier . . . 772
21.4 The L^p Function Spaces . . . 773
21.5 Fourier Series . . . 775
21.6 The Fourier Transform . . . 781
21.7 Convolution . . . 786
21.8 Signal Processing Topics . . . 788
21.9 Wavelets . . . 792
21.10 Exercises . . . 796

22 Chaos and Fractals . . . 797
22.1 Introduction . . . 797
22.2 Dynamical Systems and Chaos . . . 797
22.3 Dimension Theory and Fractals . . . 802
22.4 Iterated Function Systems . . . 806

Appendix A: Notation . . . 815

Appendix B: Abstract Program Syntax . . . 819

Appendix C: IGES . . . 822
C.1 What Is IGES? . . . 822
C.2 A Sample IGES File . . . 822
C.3 The IGES Geometric Types . . . 827
C.4 The IGES Nongeometric Types . . . 832

Bibliography . . . 835
Abbreviations . . . 835
Advanced Calculus . . . 835
Algebraic Curves and Surfaces . . . 835
Algebraic Geometry . . . 836
Algebraic Topology . . . 836
Analytic Geometry . . . 836
Antialiasing . . . 836
Blending . . . 836
Clipping . . . 837

Color 837
Computational Geometry 838
Conics 839
Constructive Solid Geometry 839
Contours 839
Convex Sets 840
Curvature 840
Curve Algorithms 840
Cyclides 841
Differential Geometry 841
Digital Image Processing 841
Engineering Applications 841
Finite Element Method 842
Fourier Series and Transforms 842
Fractals 842
General Computer Graphics 843
Geodesics 843
Geometric Modeling Books 843
Geometric Modeling Papers 845
Graphical User Interfaces 848
Graphics Pipeline 848
Graphics Standards 848
Hodographs 849
Implicit Curves and Surfaces 849
Intersection Algorithms 850
Interval Analysis 852
Mathematics for Geometric Modeling 852
Medial Axes 852
Miscellaneous 854
Numerical Methods 854
Offset Curves and Surfaces 855
PC Oriented Computer Graphics 855
Physically Based Modeling 855
Polygonization Algorithms 856
Projective Geometry and Transformations 856
Quadrics 856
Quaternions 856
Radiosity 857
Raster Algorithms 857
Ray Tracing 858
Real Analysis 859
Rendering 859
Robotics 859
Shading and Illumination (Early Work) 859
Spatial Data Structures 860
Splines 860
Subdivision Curves and Surfaces 862
Surfaces and Manifolds 862

Texture 863
Topology 863
Trimmed Surfaces 863
Virtual Reality 864
Visible Surface Detection 864
Visualization 865
Volume Rendering 865

Index **867**

Bibliographic Index **896**

Index of Algorithms **906**

PART I

BASIC COMPUTER GRAPHICS

CHAPTER 1

Introduction

1.1 Overview

This book is about constructive geometry. Our object is to study geometry, all sorts of geometry, and also to present a setting in which to carry out this investigation on a computer. The adjective "constructive" is intended to imply that we will not be satisfied with an answer to a geometric problem unless we also are given a well-defined algorithm for computing it. We need this if we are going to use a computer. However, even though our goal is a computational one, our journey will take us through some beautiful areas of pure mathematics that will provide us with elegant solutions to problems and give us new insights into the geometry of the world around us. A reader who joins us in this journey and stays with us to the end will either have implemented a sophisticated geometric modeling program in the process or at least will know how to implement one.

Figure 1.1 shows the task before us at a very top level. We have a number of representation problems. Our starting point is the "real" world and its geometry, but the only way to get our hands on that is to build a mathematical model. The standard approach is to represent "real" objects as subsets of Euclidean space. Since higher dimensional objects are also interesting, we shall not restrict ourselves to subsets of 3-space. On the other hand, we are not interested in studying all possible subsets. In this book, we concentrate on the class of objects called finite polyhedra. More exotic spaces such as fractals (the spaces one encounters when one is dealing with certain natural phenomena) will only be covered briefly. They certainly are part of what we are calling geometric modeling, but including them would involve a large amount of mathematics of a quite different flavor from that on which we wish to concentrate here. Next, after representing the world mathematically, we need to turn the (continuous) mathematical representations into finite or discrete representations to make them accessible to computers. In fact, one usually also wants to display objects on a monitor or printer, so there are further steps that involve some other implementation and computation issues before we are done.

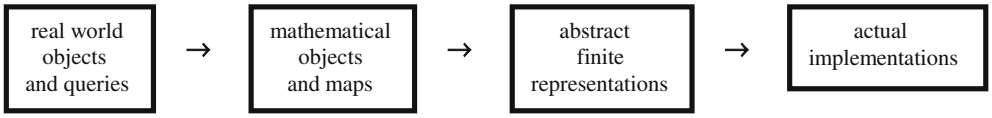


Figure 1.1. The geometric modeling representation pipeline.

As we look at the various representation problems shown in Figure 1.1, note that, although we have only mentioned objects so far, representations also need to represent the maps (operations, etc.) between them because a good and complete model of something needs to mimic everything in the original. In any case, objects and maps go hand in hand in mathematics. With every new class of objects it is fruitful to define the naturally associated maps (take vector spaces and linear transformations, for example).

To summarize, the emphasis of this book is on showing how to model finite polyhedra and the invariants associated to them on a computer and we shall show how to set up a programming environment to facilitate this investigation. One has a fairly good grip on the mathematics part of the representation pipeline, but less so on the rest, at least in terms of having a well-defined theoretical approach. The fact is that, although computer graphics is an exciting, rapidly developing field that has come a long way from the early days when people first tried to use computers for this, things are still being done in rather ad hoc ways. There is really no overall systematic approach, only a lot of isolated, special results that, neat as some of the ideas and algorithms may be, do not fit into any unifying picture. To put it another way, computer graphics today is an "art" and not a "science." There have been a few attempts to formalize the digital geometry aspect. See [Fium89] or [Herm98], for example. On the other hand, since the nonmathematical part of computer graphics depends on the current technology used for the display medium (raster devices at present) and, of course, the computer, and since this will continually evolve (with holographic displays the next dominant medium perhaps), the hardcore part of "computer" graphics may stay an art and never become a science.

All that we shall do in this chapter is get a few preliminaries out of the way. We shall introduce some basic terminology and indicate some of the mathematics we shall need. What little we have to say about hardware topics will be found in this chapter. The chapter ends with a bit of mathematics so that we can get started with some simple two-dimensional (2d) graphics.

1.2 The Basic Graphics Pipeline

Any meaningful use of a computer to study geometry implies that we ultimately want to display objects on a graphics device. Figure 1.2 shows some standard terminology for the first step of the three-dimensional (3d) graphics pipeline that takes us from the mathematical representation of an object in $\mathbf{R}^3$ to its image on the device. Objects in the world are described by the user with respect to a *world coordinate system*. The world is then projected onto a *view plane* from some *viewpoint* that we shall think of as the location of a *camera* or the *eye*. We have an associated *view plane* and *camera*

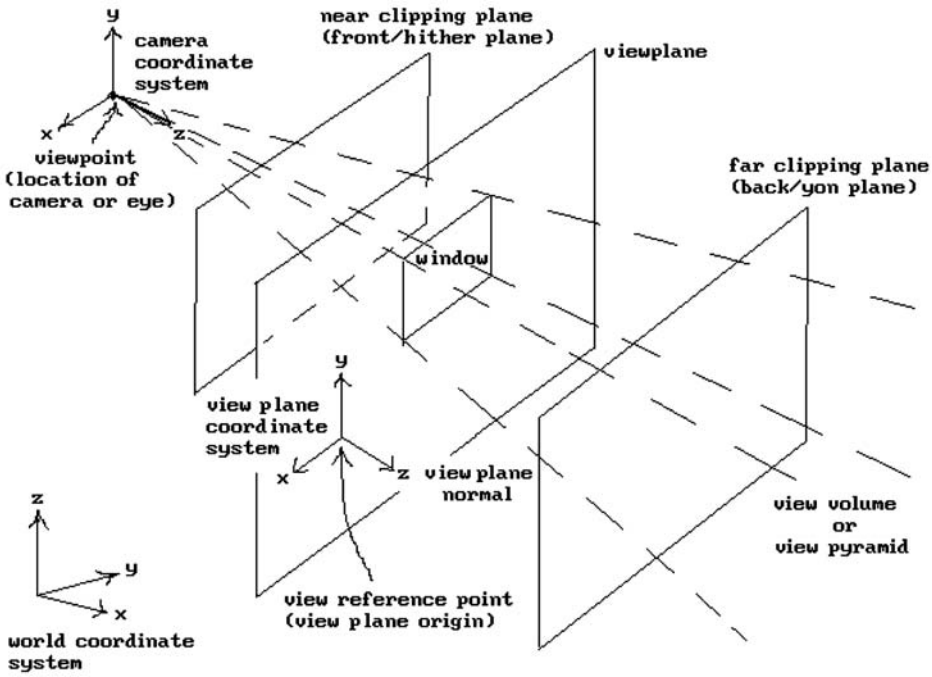


Figure 1.2. 3d graphics coordinate systems and terminology.

coordinate system. Looking from the viewpoint along the positive z-axis of the camera coordinate system specifies the *view direction*. A *window* in the view plane specifies the area of interest. The *view volume* or *view pyramid* is the infinite volume swept out by the rays starting at the viewpoint and passing through points of the window. To limit the output of objects one often uses a *near* (or *front* or *hither*) and *far* (or *back* or *yon*) *clipping plane*. The volume inside the view volume between these two planes is called the *truncated view volume* or *truncated view pyramid*. Only those parts of objects that lie in this volume and project into the window will be displayed. Finding those parts of an object is referred to as *clipping*. In principle, the three coordinate systems – the world, the camera, and the view plane coordinate system – could be distinct. In practice, however, one assumes that the coordinate axes of the camera and view plane coordinate system are parallel and the z-axes are perpendicular to the view plane. One also assumes that their x- and y-axes are parallel to the sides of the window.

The final step in mapping an object to a graphics device involves a map that transforms view plane coordinates to physical device coordinates. This is usually thought of as a two-stage process. First, an initial map transforms the window to a *viewport* that is a subrectangle of a fixed rectangle called the *logical screen*, and a second map then transforms logical screen coordinates to physical device coordinates. See Figure 1.3. Sometimes the logical screen is already defined in terms of these coordinates, so that the second map is not needed. Other times, it is set equal to a standard fixed rectangle such as the unit square $[0,1] \times [0,1]$, in which case we say that the viewport is specified in *normalized device coordinates* (NDC). The basic 3d graphics pipeline can now be summarized as shown in Figure 1.4. Chapter 4 will discuss it in great length and also fill in some missing details.

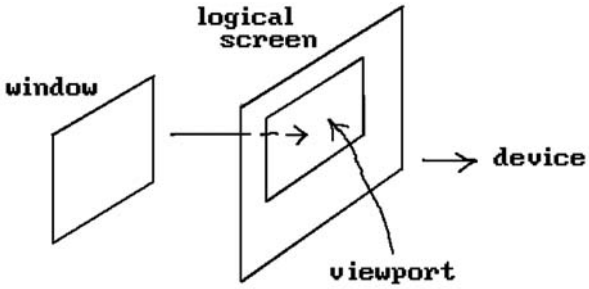


Figure 1.3. The window-to-device pipeline.

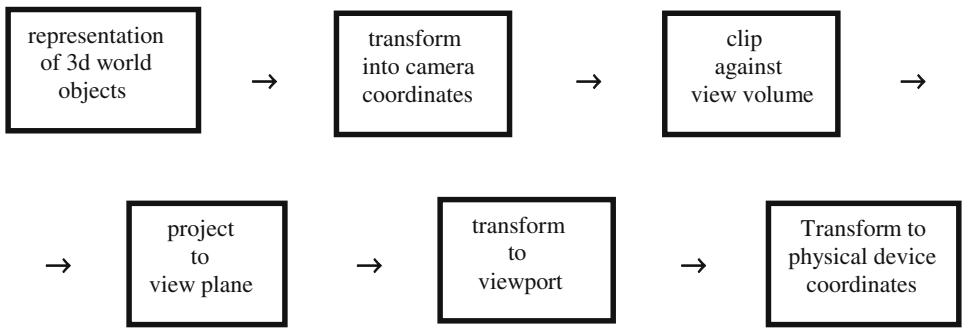


Figure 1.4. The basic 3d graphics pipeline.

The two-dimensional graphics pipeline is similar but much simpler. The window-to-device pipeline shown in Figure 1.3 stays the same, but Figures 1.2 and 1.4 get replaced by Figures 1.5 and 1.6, respectively. We have a two-dimensional *world coordinate system* and a *window* whose edges are parallel to the coordinate axes. In the case of the three-dimensional graphics pipeline, one usually assumes that the window is of a fixed size centered on the z-axis of the camera coordinate system. This is adequate to achieve most views of the world. To move the viewpoint and change the view direction we simply change the camera coordinate system. Zooming in and out is accomplished by moving the view plane further from or closer to the viewpoint. In the two-dimensional graphics case, on the other hand, one must allow the window to move and change in size. We have to be able to move the window to see different parts of the two-dimensional world and we must be able to shrink or expand the size of the window to zoom.

One word of caution is in order. The distinction between "window" and "viewport" is often blurred and, sometimes, what should be called a viewport is called a window. The terms used are not as important as the conceptual difference. One specifies **what** one sees in user coordinates and the other specifies **where** one sees it. The window, as defined above, refers to the former and the viewport, to the latter.

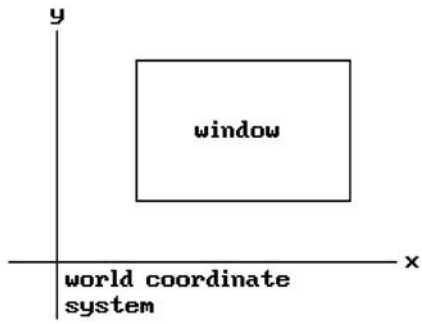


Figure 1.5. 2d graphics coordinate system and window.

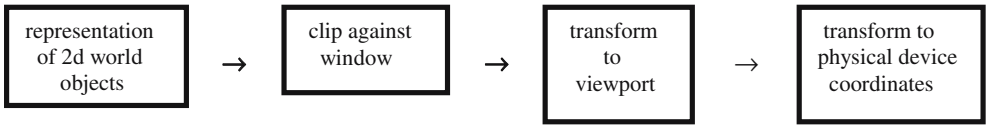


Figure 1.6. The basic 2d graphics pipeline.

1.3 Hardware Basics

Although the goal of this book is to emphasize the abstract ideas in graphics, one does need to understand a few hardware basics because that is what drives the search for efficient algorithms for the implementation of low-level graphics primitives. The most common display devices have been cathode ray tube (CRT) devices. Here an electron beam traces out an image on a phosphor-coated screen. There have been different types of CRTs, but since the early 1970s *raster scan* CRTs have been the most prevalent graphics display devices. They are refresh CRTs because the electron beam is continually rescanning the entire screen. The screen itself should be thought of as a rectangular array of dots. The image that one sees depends on how those dots are lit. The beam starts at the top of the screen and proceeds down the screen from one *scan line* to the next until it gets to the bottom. It then jumps back to the top. See Figure 1.7. The term *"horizontal retrace"* refers to the time the beam jumps from the end of a line to the beginning of the next line and *"vertical retrace"* refers to the time it jumps from the right bottom corner of the screen to the top left corner. These times, especially the latter, were often used to write to the screen to avoid flicker and knowing them was important to game developers who wanted to produce smooth animation effects.

Another display technology that has been becoming more and more popular in recent years is the *liquid crystal display* (LCD). Although there are different variants, LCDs are also raster scan devices because, for all practical purposes, they consist of a rectangular array of dots that is refreshed one row at a time. The dots themselves are the "liquid crystals," which are usually organic compounds that consist of molecules that allow light to pass through them if they are aligned properly by means of an applied voltage. The bottom line is that the liquid crystals can be individually switched on or off. LCDs have a number of advantages over the raster scan CRTs. In

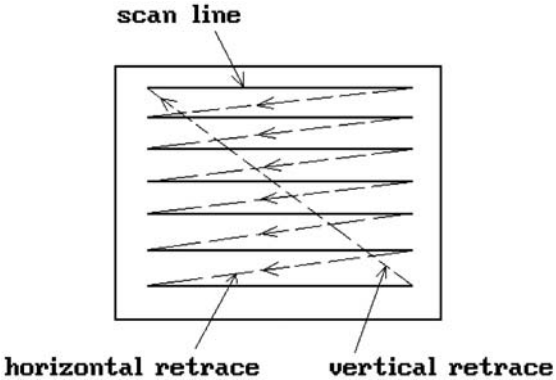


Figure 1.7. The raster scan CRT.

particular, one does not have to worry about refresh rates or flicker and they are not as bulky.

The hardware assumption made in this book, one that should apply to two-dimensional displays in the foreseeable future, is that the reader is working on a raster scan device. This assumption has an important consequence. Raster scan devices use a *refresh buffer* to specify which dots on the screen are to be lit and how. To get the picture we want, we only have to set the values in that buffer correctly. Therefore, our abstract representation problem specializes to representing subsets of Euclidean space as (discrete) subsets of a rectangle in $\mathbf{Z}^2$. Less formally, we shall talk about representing objects in a "raster." A *raster* refers to a two-dimensional rectangular array of pixels, where a *pixel* is an abbreviation for "picture element," which could, in theory, be any value. In practice, a pixel is represented in computer memory by one or more bits that specify a color. A monochrome picture is where each pixel is represented by only one bit. A row in a raster is called a *scan line*. If the raster has m columns and n rows, then we say that the *resolution* of the picture is $m \times n$.

The hardware graphics standards for computers have evolved over time. The standards for the IBM personal computer (PC) are listed in chronological order below:

Type	Resolution	Number of colors
CGA	640×200	2 (black plus one other)
Hercules	720×348	2 (black and white)
EGA	640×350	16
VGA	640×480	16
super VGA	$\geq 800 \times 600$	≥ 256

For more details about these standards see [Wilt87] or [Ferr94].

The refresh buffer of a raster scan device is usually called a *frame buffer*. In general, the term "frame buffer" refers to an array of memory (separate from main memory) thought of as a two-dimensional array of pixels (a raster). Frame buffers serve two functions:

(1) as a place where the image is stored as it is computed
(2) as a refresh buffer from which the image is displayed on a raster device

A frame buffer is an interface between what are usually relatively slow graphics computations and the high data rate video image display. In the typical personal computer the frame buffer is located on the graphics card that manages the video subsystem of the computer. It basically used to be not much more than some extra memory. For example, the table below describes the frame buffers used by the IBM PC family of computers:

Type	Size of frame buffer	Starting memory address (in hexadecimal)
CGA	16 K	B800:0000
Hercules	64 K	B000:0000
EGA,VGA	256 K for 16 colors	accessed via a 64 K window starting at A000:0000
super VGA	1 M or more	accessed via a 64 K window starting at A000:0000

Over time the graphics subsystems of personal computers have become more powerful, and the hardware is supporting more and more of the operations that one needs to perform in graphics, such as antialiasing (Section 2.6) and the bit map operations discussed below and in Section 2.10. They also have additional buffers, such as a z-buffer (Chapter 7), texture buffers (Chapter 9), or stencil buffers (for masking operations). This support only used to be found on high-end graphics workstations.

As indicated above, displaying objects on the computer screen involves writing to the frame buffer. This amounts to storing values in memory. Ordinarily, a store operation replaces the value that was there. In the case of frame buffers one has more options. If A is a location in memory, then let [A] denote the content of A. Frame buffers typically support store operations of the form (V **op** [A]) → [A], where V is a new value and **op** is a binary logical operator that operates on a bit-by-bit basis. Typical binary logical operations on bits are **or**, **and**, **xor**, and **replace**. The statement (V **replace** [A]) → [A] corresponds to the standard store operation where the new value replaces the old one. When a frame buffer uses a store operation corresponding to an operator **op**, we shall say that it is in ***op*** *mode*. For example, we may talk about being in **xor** mode.

As a simple example of how having various modes for a frame buffer can be useful, consider how the standard quick and dirty method used to move a cursor around on the screen without destroying the background uses the **xor** mode. The method relies on **xor**'s well-known property that

$$b \textbf{ xor } (b \textbf{ xor } a) = a.$$

What this means is that if one **xor**'s the same value to a memory location twice in a row, then that memory location will hold its original value at the end. Now, a straightforward way to move a cursor on the screen without erasing what is there would be to save the area first before writing the cursor to it and then restoring the old value after the cursor has moved. This would be very time consuming. There is a much better way of using the **xor** mode. Assume that the cursor starts out at some initial position defined by a variable oldA. Now switch into **xor** mode and repeat the following three steps as often as desired:

Draw cursor at oldA (this will erase the cursor)
Draw cursor at new position newA
Replace the value in oldA with that in newA

Note that **replace** mode would cause this loop to erase everything in the cursor's path and leave behind a trail of the cursor. There is one disadvantage with the **xor** operation, however, which may not make it a viable option in certain situations. Although one can use it to move objects around on the screen without destroying the background, the objects may change color. If, for example, one wants to move a red cursor and have it stay red, then this is not possible with **xor** mode because the cursor will assume different colors as it moves over differently colored areas of the screen. Therefore, if it is important that the cursor stay red, then there is no simple alternative to first saving the area to which one is writing and restoring it afterwards.

Because the availability of logical operators in store operations simplifies and speeds up many useful graphics operations, current graphics systems have built-in hardware support for them. We will have more to say about this in Section 2.10.

We finish this section with two more terms one sees frequently in graphics. *Scan conversion* is the act of converting points, lines, other geometric figures, functions, etc., into the raster data structure used in frame buffers one scan line at a time. After a scene is modeled, it needs to be "rendered." To *render* a scene means to construct an image on a display device that is visually satisfactory. What is "satisfactory" depends firstly on the device and its constraints and secondly on what one is trying to do. To emphasize the position that rendering occupies in graphics, keep in mind that the modeling or mathematical representation comes first and then the rendering. Any given model can have many different renderings. For example, a sphere can be rendered in different colors. In trying to render scenes one runs into a number of important problems: visible line or surface determination, illumination, texturing, transparency, etc. These will all be addressed in coming chapters.

1.4 Graphics Standards and Primitives

A person who wants to develop a graphics program has to learn how to access the graphics capabilities of the system that he/she is working on. Unfortunately, there are many graphics devices out there in the world. If one wanted a program to work with all those devices and if one had to program the hardware directly, then one could easily spend all of one's time on very low-level code and never get to that in which one is really interested. Therefore, let somebody else, say the manufacturer of the system or the compiler vendor, worry about the low-level stuff so that one can concentrate on higher-level ideas. This is where software graphics standards come in. They are the interface between a high-level language and the low-level code that talks to the actual hardware. The interface is basically a specification of high-level graphics primitives. As long as one's code calls only these primitives, a program will run on any system that is supported by that particular interface. In other words, standards make code **portable** by making it **device independent**.

Lots of different standards exist with some more sophisticated than others. The early DOS operating system standards, such as the Borland Graphics Interface (BGI), were fairly primitive. Any program in Borland PASCAL or C/C++ that used the Borland PASCAL or C/C++ graphics primitives was guaranteed to run under DOS on most of

the IBM PC–compatible computers. The same was true of the corresponding interface found in the Microsoft compilers. A number of much more sophisticated standards were developed over the years such as

Core (The 3d Core Graphics System): specified by ACM SIGGRAPH committees in 1977 and 1979 ([GSPC77] and [GSPC79])
GKS (Graphics Kernel System): specified by various national and international committees in the 1980's with a 3d version becoming a standard in 1988 ([ANSI85], [ISO 88], [EnKP84], [BDDH95])
PHIGS (Programmer's Hierarchical Interactive Graphics System): a more complex standard than GKS, which was specified by ANSI (the American National Standards Institute) in 1988 ([ANSI88] and [VanD88])

See [Cars98] for a brief history. Two more recent standards are

OpenGL: see [WNDS99], [KemF97], [WriS00]
DirectX: see [Glid97], [BarD98], [Timm96]

The rise in the popularity of the Microsoft Windows operating system meant that its application programming interface (API) became a driving force for standards for that system. At first there was only the basic Windows graphics device interface (GDI). This made writing graphics programs hardware independent, but at the expense of speed. The result was that developers, especially those involved in writing games, stayed with DOS, which allowed programmer to go directly to the hardware and squeeze out the last ounce of speed essential for games. To attract developers to Windows, Microsoft next came out with WinG, which provided a few low-level bitmap functions that did speed up basic graphics operations substantially, but it was not enough. Microsoft's graphics standard successors to WinG were DirectDraw and Direct3D, which were part of the DirectX API that was intended for multimedia applications. DirectDraw provided two-dimensional graphics primitives. Direct3D was the three-dimensional counterpart. Although these allowed for high-performance graphics under Windows, DirectDraw and Direct3D were low level. A competing and higher-level graphics API is OpenGL, a graphics standard originally developed by Silicon Graphics, Inc., for its graphics workstations. Good implementations of OpenGL for Windows are built on DirectX drivers. Although native DirectX code is currently faster, the advantage of OpenGL is that it is available on many other computer and operating system platforms, a plus for Internet applications. The companion programs for this book, GM and SPACE, use OpenGL.

Having just praised standards, we also need to point out what has traditionally been their downside. If one uses a standard, then one must be willing to put up with extra overhead in the code. Furthermore, because standards are device independent, they, by definition, usually do not take advantage of any special features that a particular piece of hardware may have. What this means is that programs that use them are sometimes much slower on a particular machine than a program that accesses its hardware features directly. Software developers have often been forced to choose between device independence and speed in those cases where speed is critical. Fortunately, with DirectX and OpenGL the situation has much improved and this is no longer a serious problem.

1.5 From Window to Viewport

One of the first bits of mathematics one runs into in a graphics program is the transformation from the window to the viewport. Both the window and viewport are representable as rectangles in the plane whose sides are parallel to the coordinate axes. What we are looking for is a simple map from one of these rectangles to another. Intuitively, all this amounts to is a change of scale.

The standard representation for our rectangles is as products of intervals in the form $[a,b] \times [c,d]$. Normally, the implied assumption in the representation of an interval like $[a,b]$ is that $a \le b$; however, in our current context where we will be interested in maps from one interval to another, we do not require that. It will be useful to allow $a > b$. Returning to our discussion of windows and viewport, if one uses normalized device coordinates, the viewport is a subrectangle of $[0,1] \times [0,1]$. If one considers the viewport as a rectangle in the raster, then it has the form $[m_1,m_2] \times [n_1,n_2]$, where m_i and n_i are integers. There is one caveat, however. The (0,0) position in the raster has traditionally been associated to the top left-hand corner on the screen. That means that the y-axis has to be inverted because users always think of that axis as going up, not down. In other words, if, say, the resolution is 800×600 and the viewport is the entire screen, then the viewport should be represented by the rectangle $[0,799] \times [599,0]$.

Mathematically then, the search for the window-to-viewport transformation boils down to the following: If $\mathbf{W} = [w_a,w_b] \times [w_c,w_d]$ and $\mathbf{V} = [v_a,v_b] \times [v_c,v_d]$ are the rectangles that represent the window $\mathbf{W}$ and viewport $\mathbf{V}$, respectively, then we want a map $T\colon \mathbf{W} \to \mathbf{V}$ of the form

$$T(x,y) = (T_1(x), T_2(y)),$$

where each T_i is linear. In other words, we have two one-dimensional problems of the form:

> Given intervals $[a,b]$ and $[c,d]$, find the linear map $S\colon [a,b] \to [c,d]$ with $S(a) = c$ and $S(b) = d$.

If $S(x) = \alpha x + \beta$, then the stated boundary conditions for S lead to two equations in two unknowns α and β, which are easily solved. We get that

$$\begin{aligned} S(x) &= \frac{d-c}{b-a}x + \frac{bc-ad}{b-a} \\ &= c + \frac{x-a}{b-a}(d-c). \end{aligned}$$

The second form of the answer says that we send x to that point in $[c,d]$, which is the same percentage of the way from c to d as x is from a to b. If one remembers that intuitive fact then one has no need to solve equations because the answer is obvious. At any rate, we now have the following solution for T:

$$T(x,y) = \left(\frac{1}{w_b - w_a}((v_b - v_a)x + (w_b v_a - w_a v_b)), \frac{1}{w_d - w_c}((v_d - v_c)y + (w_d v_c - w_c v_d)) \right).$$

Later on in Chapter 4 we shall derive a more general window-to-viewport transformation, but what we have now is good enough to do some simple two-dimensional graphics programming.

1.6 Programming Notes

In the early years of the IBM PC and DOS and after there were some programming languages such as PASCAL or C that had some basic graphics primitives built into the language, it was fairly easy to describe what had to be done to write a graphics program. It was a three-stage process. First, every such program had to enter "graphics mode" when it started, then it could do all the graphics that it wanted, and finally it had to leave graphics mode at the end and restore whatever mode the system was in before the program started. Life has gotten much more complicated now that we are in an age of graphical user interfaces (GUIs) and the Microsoft Windows operating system. Describing how one programs the graphics API for Microsoft Windows would entail writing another book. However, we do want to give the reader a flavor of what is involved. To that end we present and discuss our only piece of low-level graphics code in this book. It shows how one would have used BGI code for the DOS operating system.

As we just mentioned, the first thing that needed to be done in any graphics program was to initialize both the hardware and certain global variables describing this hardware to the program. Program 1.6.1 shows a very basic sample BGI C procedure, "InitializeGraphics," which did this. The BGI procedure "initgraph" did the initialization and returned the information about the hardware in use in its parameters "graphDriver" and "graphMode." The third parameter to the procedure was a DOS path name to a directory where the BGI drivers were located. An empty string meant that they were in the current directory. The function "graphresult" returned any error that might have occurred and prevented the graphics system from being initialized. A typical error was caused by the fact that the BGI driver was not located in the current directory. The BGI drivers were files that came with the Borland programming languages. Each contained hardware-specific code for the basic graphics primitives and the one that matched one's hardware got linked into one's program.

After the graphics mode was initialized correctly, we then stored some useful constants in global variables. The functions "getmaxx" and "getmaxy" returned the maximum resolution of the screen in pixels in the horizontal and vertical direction, respectively. The "textheight" and "textwidth" functions returned the height and width of characters which one needs to determine the space required for text.

The "atexit" procedure passed the name of a procedure to call when the program was done and was about to return to DOS. We have passed the name of the "MyExitProc" procedure that calls the "closegraph" procedure. The latter switches from graphics mode back to the standard 25 line and 80 column text mode (or whatever mode the system was in before the program was called). Without the call to the "closegraph" procedure the system would have been left in graphics mode with a messed-up screen and would probably have had to be rebooted.

Assuming that the "InitializeGraphics" procedure executed without problems, one would be in graphics mode and be presented with a blank screen. As indicated earlier,

```
/* Global variables */
int  graphDriver, graphMode,    /* After call to InitGraph these variables specify the
                                   current hardware */
     numColors,                 /* maximum number of colors */
     scrnXmax, scrnYmax         /* screen resolution */
     txtHeight, txtWidth;       /* the height and width in pixels of a character in the
                                   current font */

void MyExitProc (void)
{ closegraph ();                /* Shut down the graphics system */
}

void InitializeGraphics (void)
{ int errorCode;

  graphDriver = DETECT;         /* DETECT is a BGI constant */
  initgraph (&graphDriver,&graphMode,"");
  errorCode = graphresult ();

  if  ( errorCode != grOk )     /* grOk is a BGI constant */
  { /* Error occurred during initialization */
    printf (" Graphics system error: %s\n",grapherrormsg (errorCode));
    exit (1);
  }

  atexit (MyExitProc);          /* so that we do closegraph when exiting */

  numColors = getmaxcolor () + 1;
  scrnXmax   = getmaxx ();
  scrnYmax   = getmaxy ();
  txtHeight  = textheight ("A");
  txtWidth   = textwidth ("A");
}
```

Program 1.6.1. Code for initializing DOS graphics mode.

doing a similar initialization for Microsoft Windows is much more complicated. The reason is that the user's program is now initializing one of potentially many windows on the screen. Under DOS basically only **one** window was initialized, namely, the whole screen. If a program wanted to deal with multiple windows, it would have to do that completely by itself. In other words, with Microsoft Windows we have a more complicated initialization procedure but we gain functionality. If one is using OpenGL or DirectX, then actually two initializations are required. After initializing the native Windows GDI, so that one can run the program in a standard window on the screen

and use basic windowing operations, one has to initialize OpenGL and DirectX in a separate step.

After a program has initialized the graphics hardware, the next step is to decide how to lay out the screen. Where should graphics output go? Where to put the menus? What about an area for getting feedback from the user and printing system–related messages? Books have been written on what makes for a good graphical user interface. See [Pedd92] for example. Microsoft has its own recommendations for programs that run in its Windows environment. See [Micr94].

One thing is clear though about today's GUIs. They take an awful lot of code and time to develop. Even if one does not start from scratch and one uses the typical APIs one gets when developing for an environment like Windows, it still requires quite a bit of understanding about the underlying architecture to use them effectively. For that reason, when this author has taught graphics classes he always, since the days of DOS, provided the students with a program similar to the current GM program that can be found on the accompanying CD. Its interface, described in the document GmGUI which is also on the CD, allowed both mouse and keyboard input and made it easy for students to add menus. In this way the students did not have to spend any time developing this interface and could concentrate on implementing the various algorithms described in the book. The current Windows version of GM is also such that students do not need to have any prior knowledge of Windows or C++. (They obviously do have to know how to program in C.) A couple of lectures at the beginning of the semester and another one or two later on to describe some additional features was all that was necessary. Of course, if one wants to make use of OpenGL, then this takes extra time.

The GM program already comes with quite a bit of functionality built into it. This means that some of the programming projects at the end of the chapters in this book, in particular some of the basic ones in the early chapters such as this one, have already been implemented either implicitly or explicitly. Readers who are new to graphics programming should ignore this fact and will find it very instructive to develop the required code on their own. They can then compare their solutions with the ones in the GM program. It is when one gets to more advanced projects that building on the GM program would be appropriate.

Finally, this book will not get into any device-dependent issues and all the algorithms and code will be analyzed and presented at a higher level than that. Except for the BGI example above, we present no actual code but shall use a fairly standard abstract program syntax (pseudocode) described in Appendix B. We shall also not use any actual API when discussing abstract algorithms, but, if needed, use the following set of graphics primitives:

(The points and rectangles below are assumed to be in the raster, that is, they are specified by **integer** coordinates. Rectangles are specified by two points, the top left and bottom right corner. By a rectangle we mean the border only, not the interior.)

SetMode (MODE)	(sets "current" mode to MODE, where MODE is a bit operation such as **xor** or **replace**)
SetColor (COLOR)	(sets "current" color to COLOR)
Draw (point or rectangle)	(in "current" color and mode)
Erase (point or rectangle)	(draws in "background" color)

Draw (point or rectangle, attribute)	(attribute is typically a color but could be something more general)
DrawLine (point, point)	(draws raster version of line segment from first point to second in "current" color and mode)
Write (string, point)	(write a string into the raster at pixel location point)

Note that erasing a point or rectangle is really the same as drawing it in the "background" color. We list the "Erase" procedure separately only for conceptual reasons. Also, drawing a rectangle or a line segment could be implemented with the "Draw (point)" procedure, but all current graphics APIs have efficient procedures for drawing rectangles and line segments directly and that is why we list that procedure separately. We shall show how the DrawLine procedure is implemented in terms of "Draw (point)" in Chapter 2. Of course, drawing lines is a very basic operation that typically is called many times. It is a place where optimization really pays off and is best implemented in assembly language, or better yet, in hardware. Therefore, when coding programs one should use the line drawing procedure that comes with the software.

The primitives above can easily be implemented using any given API. We believe, however, that they will make our abstract code more readable. In any case, whatever system the reader is working on, it is assumed that he/she can implement these procedures. These primitives are all that we shall need to describe all of the algorithms in this book.

1.7 Exercises

Section 1.5

1.5.1 Find the equations for the following transformations:

(a) T: $[-1,3] \to [5,6]$
(b) T: $[2,7] \to [3,1]$
(c) T: $[-1,2] \times [3,5] \to [5,7] \times [-3,-4]$
(d) T: $[7,-2] \times [1,2] \to [3,2] \times [0,3]$

1.8 Programming Projects

Section 1.5

In these programming assignments assume that the user's world is the plane. We also assume the reader has a basic windowing program with an easily extensible menu system. The GM program is one such and the user interface in the projects below fits naturally into that program. Furthermore, the term "screen" in the projects below will mean the window on the real screen in which the program is running. All projects after the first one (Project 1.5.1) assume that a window-to-viewport transformation has been implemented.

1.5.1 A window-to-viewport transformation

The goal of this first project is simply to write and test the window-to-viewport transformation. The main menu should add another item to the list:

```
MAIN
  :
Utils
```

Activating the Utils item should bring up the menu

```
UTILS
Window
Viewport
Dimension
```

that allows the user to change the current dimensions of the window, to change the location of the viewport on the screen, and to toggle the display of the window's dimensions, respectively. If the window dimension display has been toggled to be on, then it should stay on the screen no matter which menu is the current one. Let the default window be $[-10,10] \times [-10,10]$. Keep in mind though that the window dimensions can be arbitrary real numbers. It is the viewport dimensions that are always integers. One way to display the window dimensions would be as follows:

```
+10


-10
-10                +10
```

In this project there are no objects to display but it can be tested by drawing viewports with a background that has a color different from the rest of the screen and checking the dimensions that are displayed.

1.5.2 Graphing functions

The object of this project is to try out some line drawing commands. Specifically, you are to draw a linear approximation to the graph of some functions. Add another item to the main menu:

```
MAIN
  :
Graphs
```

The idea is to evaluate a given function at a finite set of values in its domain and then to draw the polygonal curve through the corresponding points on its graph. See Figure 1.8. Let a user specify the following:

(1) The interval [a,b] over which the function is to be graphed.

(2) The "resolution" n, meaning that the function will be evaluated at $a + i\,(b - a)/n$, $0 \le i \le n$.

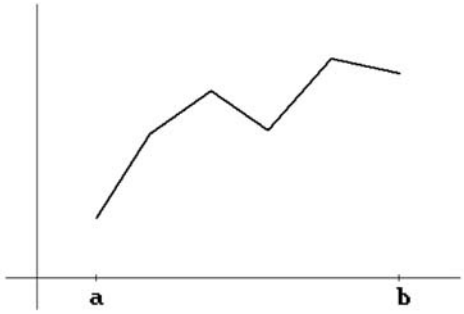


Figure 1.8. A sample function graph.

Because the values of a function may change substantially from one domain to the next one it is important that one choose the window well; otherwise, the graph may look very tiny or not show up at all. A simple scheme would scale the x-direction to be not much bigger than the domain of the function and the y-direction to cover only that range of values needed for the graph. To do the latter one should first evaluate all the points, find the maximum and minimum of the y-value, and then adjust the y-dimension of the window to those values. Such a scheme does have the disadvantage, however, that the window will keep changing if one changes the domain. To avoid this one could leave it to the user to decide on the window or pick some fixed default window that only changes if the graph moves outside it. To test whether a graph lies entirely in the window check that all the points on it lie in the window.

Finally, to make the graph more readable it would help to show the coordinate axes with ticks for some representative values.

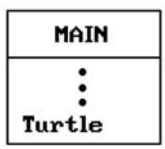
1.5.3 Turtle graphics

This is another project to try out line drawing commands. Assume that a "turtle" is crawling around **in the plane** ($\mathbf{R}^2$). A turtle is an object that is defined by a position and a direction (in which it is looking). The standard basic commands that a turtle understands are

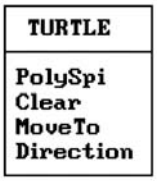
Forward (dist)
MoveTo (x,y)
Turn (θ)
TurnTo (θ)
Right (θ)

The "**Forward**" procedure draws a line from the current position of the turtle to the new one, which is a distance "dist" from the old one in the direction in which the turtle is looking. The "**MoveTo**" procedure does not, but simply repositions the turtle to the real point (x,y). The "**Turn**" procedure turns the turtle by the angle θ specified **relative** to the current direction it is looking. The "**TurnTo**" procedure is the absolute version of "**Turn**." It turns the turtle to look in the direction that makes an angle θ with the x-axis. You will be surprised at what interesting figures can be created by a turtle. For lots more on turtle geometry and interesting graphics that can be generated with it see [AbeD81].

Add an item to the main menu so that it now looks like:



Activating the Turtle item in the main menu should enter the user into "turtle graphics" mode, show a turtle at its current position, and show the menu



A simple drawing of a turtle would be a small square with a line segment emanating from it to show the direction in which it is looking. Activating "**PolySpi**" in the menu should (starting with the current position and direction of the turtle) draw the path taken by the turtle according to the following algorithm:

```
real dist, turnAngle, incr;
integer numSteps;

{Draws numSteps segments of a spiral with given exterior angle
  (measured in degrees)}
for i:=1 to numSteps do
  begin
    Forward (dist);
    Right (turnAngle);
    dist := dist + incr;
  end;
```

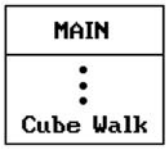
Draw the spiral after asking the user to input values for the four parameters. Some values to try are num = 100 and (dist,angle,incr) = (.1,144,.1), (.05,89.5,.05), (.05,170,.05), and (.05,60,.05). The "**Clear**" command should clear the viewport except for the turtle. The "**MoveTo**" and "**Direction**" command should have the obvious effect on the turtle.

When entering the turtle menu for the first time, the turtle should be initialized to "sit" at the center of an empty viewport "looking" right. After that the program should not reinitialize the turtle or clear the screen on its own, except that the screen should be cleared whenever the Turtle menu is exited. The turtle should only be visible whenever one is inside the Turtle menu. When outside the turtle menu, the graphics area should always be blank except for the possible dimension values.

Note: You do not have to worry about clipping the turtle's path to the window. In this program it is the user's responsibility to ensure that the path lies entirely inside the window.

1.5.4 Turtle crawling on a cube ([AbeD81])

For this project change the main menu to



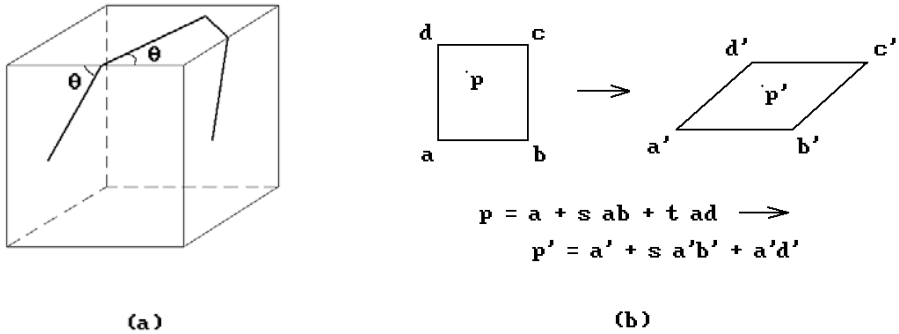


Figure 1.9. Turtle crawling on cube.

This project is more advanced and requires familiarity with vectors. It also needs the formula for the intersection of two segments discussed in Section 6.5. To display a turtle crawling on a cube we use a parallel projection of a three-dimensional cube into the plane. In this way, the path of the turtle can be described via linear combinations of planar vectors without involving any knowledge of transformations from $\mathbf{R}^3$ to $\mathbf{R}^2$. See Figure 1.9(a) for what one should see. Note that walking in a straight line preserves the angle the path makes with an edge as the edge is crossed. Ignore the case where a path meets a vertex of the cube. One can let the turtle disappear there.

The key idea is that, at any time, the turtle is in a face of the cube which one can identify with a fixed square. The parallel projection then maps this square onto a parallelogram. See Figure 1.9(b) where we map the square **A** with vertices **a**, **b**, **c**, and **d** onto the parallelogram **A′** with vertices **a′**, **b′**, **c′**, and **d′**, respectively. If **p** is an arbitrary point of **A**, write **p** in the form

$$\mathbf{a} + s\ \mathbf{ab} + t\ \mathbf{ad}.$$

The parallel projection will then map **p** to

$$\mathbf{a'} + s\ \mathbf{a'b'} + t\ \mathbf{a'd'}.$$

Therefore, the basic steps are:

(1) Pick points $\mathbf{a}_i$ in the plane onto which the vertices of the cube (one may as well use the standard unit cube $[0,1] \times [0,1] \times [0,1]$) get mapped.

(2) Keep track of the face the turtle is on along with the identification of its vertices with the given vertices in the plane.

(3) When moving forward a certain distance d from a point **p**, check if this entails crossing an edge. If yes, then move only to the point on the edge and move the remaining distance in the new face in the next step. Let **q** be the end point of the current segment through which we moved.

(4) Find the segment **p′q′** which is the image of the segment **pq** and draw it.

(5) Repeat steps (3) and (4) until one has moved forward the distance d. If we crossed an edge, then we may have to update the face we are on.

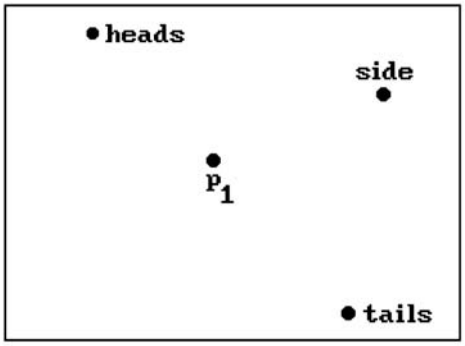
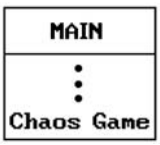


Figure 1.10. The chaos game.

To make the picture look nicer, draw the segments on the back faces of the cube with dashed lines. To generate paths use a procedure like the polyspiral procedure in project 1.3.

1.5.5 The Chaos game ([Barn87])

To play this game add the following item to the main menu:



Let the user pick four points on the screen. For example, Figure 1.10 shows points marked "heads," "tails," "side," and "$\mathbf{p}_1$." Now generate points $\mathbf{p}_i$, $i \geq 2$, as follows: "Toss a coin." If the coin comes up heads, $\mathbf{p}_i$ is the point half way from $\mathbf{p}_{i-1}$ to the point marked "heads." If the coin comes up tails, $\mathbf{p}_i$ is the point half way from $\mathbf{p}_{i-1}$ to the point marked "tails." If the coin ends up on its side, $\mathbf{p}_i$ is the point half way from $\mathbf{p}_{i-1}$ to the point marked "side." Analyze the patterns of points that are generated in this fashion. Tossing a coin simply translates into generating a random integer from {0,1,2}.

CHAPTER 2

Raster Algorithms

Prerequisites: Sections 4.2, 5.2 in [AgoM05] (to define and motivate concepts in Section 2.2), Section 21.8 (for Section 2.6)

2.1 Introduction

As pointed out in our introductory chapter, the only real implementation constraint that the hardware places on us is that all geometric objects eventually need to be represented by a collection of points in a two-dimensional grid (a raster). The subject matter of this chapter is to analyze the geometry of discrete sets and to describe some important algorithms that map continuous planar objects to discrete ones. Insofar as it is possible, one would like the discrete world to be a mirror image of the continuous one.

Section 2.2 starts the chapter by introducing some discrete world terminology. Sections 2.3, 2.4, and 2.9.1 describe a border-following algorithm and several region-filling algorithms. Sections 2.5 and 2.9 deal with discrete curves – how to generate them and work with them efficiently. Sections 2.6–2.8 discuss some problems caused by discretization and some ways to deal with them. Hardware issues that are involved in the optimization of low-level graphics primitives are ignored in this book, but Section 2.10 does briefly discuss how the existence of certain bit map operations helps out. We finish with a brief discussion in Section 2.11 of a few basic techniques in 2d animation.

2.2 Discrete Topology

This section defines the discrete analogs of a number of important continuous concepts. Probably the most basic of these is the idea of continuity itself, and central to that is the idea of a neighborhood of a point. Neighborhoods define the "topology" of a space. Now a raster can be modeled in an obvious way as a subset of $\mathbf{Z}^2$ and so this leads us to describe some possible definitions of a neighborhood of a point in $\mathbf{Z}^2$, or

more generally in $\mathbf{Z}^n$. Although we are only interested in the case n = 2 in this chapter, there is nothing special about that case (except for the terminology), and it is useful to see what one can do in general. In fact, the case n = 3 will be needed to define discrete lines for volume rendering in Chapter 10. This book will not delve into the concept of curve rasterization in dimensions larger than 3, but the subject has been studied. See, for example, [Wüth98] or [Herm98].

Definition. In $\mathbf{Z}^2$, the *4-neighbors* of (i,j) are the four large grid points adjacent to (i,j) shown in Figure 2.1(a). The *8-neighbors* of (i,j) are the eight large grid points adjacent to (i,j) in Figure 2.1(b). More precisely, the 4-neighbors of (i,j) are the points (i,j + 1), (i − 1,j), (i,j − 1), and (i + 1,j). The 8-neighbors can be listed in a similar way.

In order to generalize this definition to higher dimensions, think of the plane as tiled with 1 × 1 squares that are centered on the grid points of $\mathbf{Z}^2$ and whose sides are parallel to the coordinate axes (see Figure 2.1 again). Then, another way to define the neighbors of a point (i,j) is to say that the 4-neighbors are the centers of those squares in the tiling that share an edge with the square centered on (i,j) and the 8-neighbors are the centers of those squares in the tiling that share **either** an edge **or** a vertex with that square. Now think of $\mathbf{R}^n$ as tiled with n-dimensional unit cubes whose centers are the points of $\mathbf{Z}^n$ and whose faces are parallel to coordinate planes.

Definition. In $\mathbf{Z}^3$, the *6-neighbors* of (i,j,k) are the grid points whose cubes meet the cube centered at (i,j,k) in a face. The *18-neighbors* of (i,j,k) are the grid points whose cubes meet that cube in either a face **or** an edge. The *26-neighbors* of (i,j,k) are the grid points whose cubes meet that cube in either a face **or** an edge **or** a point.

Figure 2.2(a) shows the cubes of the 6-neighbors of the center point. Figure 2.2(b) shows those of the 18-neighbors and Figure 2.2(c), those of the 26-neighbors. More generally,

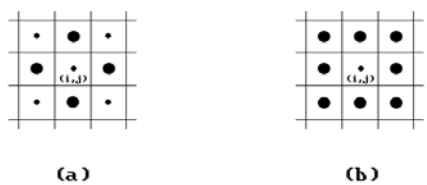


Figure 2.1. The 4- and 8-neighbors of a point.

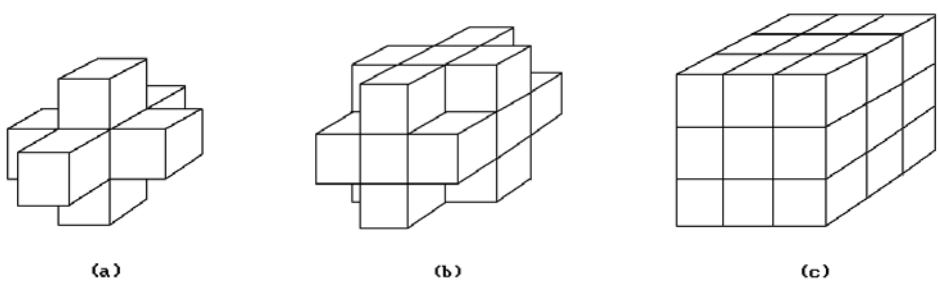


Figure 2.2. The 6-, 18-, and 26-neighbors of a point.

Definition. Let $\mathbf{p} \in \mathbf{Z}^n$ and let d be a fixed integer satisfying $0 \le d \le n - 1$. Suppose that k is the number of points of $\mathbf{Z}^n$ that are the centers of cubes that meet the cube with center $\mathbf{p}$ in a face of dimension larger than or equal to d. Each of those points will be called a *k-neighbor* of $\mathbf{p}$ in $\mathbf{Z}^n$.

Note: The general definition for k-neighbor is not very satisfying because it is relatively complicated. It would have made more sense to call the point a "d-neighbor." Unfortunately, the terminology as stated is too well established for the two- and three-dimensional case to be able to change it now.

Definition. Two points in $\mathbf{Z}^n$ are said to be *k-adjacent* if they are k-neighbors.

k-adjacency is the key topological concept in the discrete world. All the terms defined below have an implicit "k-" prefix. However, to simplify the notation this prefix will be dropped. For example, we shall simply refer to "adjacent" points rather than "k-adjacent" points. It must be emphasized though that everything depends on the notion of adjacency that is chosen, that is, for example, whether the intended k is 4 or 8, in the case of $\mathbf{Z}^2$, or 6, 18, or 26, in the case of $\mathbf{Z}^3$. To make this dependency explicit, one only needs to restore the missing "k-" prefix.

There is a nice alternate characterization of k-adjacency in two special cases that could have been used as the definition in those cases.

Alternate Definition. Let $\mathbf{p} = (p_1, p_2, \ldots, p_n)$, $\mathbf{q} = (q_1, q_2, \ldots, q_n) \in \mathbf{Z}^n$. The points $\mathbf{p}$ and $\mathbf{q}$ are *2n-adjacent* in $\mathbf{Z}^n$ if and only if

$$\sum_{i=1}^{n} |q_i - p_i| = 1$$

They are *$(3^n - 1)$-adjacent* in $\mathbf{Z}^n$ if and only if $\mathbf{p} \ne \mathbf{q}$ and $|q_i - p_i| \le 1$ for $1 \le i \le n$.

Properties of 2n- and $(3^n - 1)$-adjacency are studied extensively in [Herm98].

Definition. A (*discrete* or *digital*) *curve* from point $\mathbf{p}$ to point $\mathbf{q}$ in $\mathbf{Z}^n$ is a sequence $\mathbf{r}_s$, $1 \le s \le k$, of points such that $\mathbf{p} = \mathbf{r}_1$, $\mathbf{q} = \mathbf{r}_k$, and $\mathbf{r}_s$ is adjacent to $\mathbf{r}_{s+1}$, $1 \le s \le k - 1$. Furthermore, with this notation, we define the *length* of the curve to be $k - 1$.

For example, the points $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and $\mathbf{p}_4$ in Figure 2.3 form a discrete curve of length 3 with respect to 8-adjacency but not with respect to 4-adjacency because $\mathbf{p}_1$ and $\mathbf{p}_2$ are not 4-adjacent.

Definition. A set $\mathbf{S}$ is *connected* if for any two points $\mathbf{p}$ and $\mathbf{q}$ in $\mathbf{S}$ there is a curve from $\mathbf{p}$ to $\mathbf{q}$ that lies entirely in $\mathbf{S}$. A maximal connected subset of $\mathbf{S}$ is called a *component*.

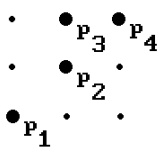


Figure 2.3. An 8-connected discrete curve.

Because of the difference between 4- and 8-connected, note the difference between a 4-component and an 8-component. It is easy to show that every 8-component is the union of 4-components (Exercise 2.2.2). A similar comment holds for the components of sets in $\mathbf{Z}^3$.

Some Definitions. We assume that the sets **S** below are subsets of some fixed set **P** in $\mathbf{Z}^n$. In practice, **P** is usually a large but finite solid rectangular set representing the whole *picture* for a scene, but it could be all of $\mathbf{Z}^n$.

The *complement* of **S** in **P**, denoted by $\mathbf{S}^c$, is, $\mathbf{P} - \mathbf{S}$.
The *border* of **S**, B(**S**), consists of those points of **S** that have neighbors belonging to $\mathbf{S}^c$ if $\mathbf{S} \neq \mathbf{P}$ or neighbors in $\mathbf{Z}^n - \mathbf{P}$ if $\mathbf{S} = \mathbf{P}$.
The *background* of **S** is the union of those components of $\mathbf{S}^c$ that are either unbounded in $\mathbf{Z}^n$ or that contain a point of the border of the picture **P**.
The *holes* of **S** are all the components of $\mathbf{S}^c$ that are not contained in the background of **S**.
S is said to be *simply connected* if **S** is connected and has no holes.
The *interior* of **S**, i**S**, is the set **S** – B(**S**).
An *isolated point* of **S** is a point of **S** that has no neighbors in **S**.
If **S** is a finite set, then the *area* of **S** is the number of points in **S**.

See Figure 2.4 for some examples.

Definition. There are several ways to define the *distance* d between two points (i,j) and (k,l) in $\mathbf{Z}^2$, or, more generally, between points **p** and **q** in $\mathbf{Z}^n$:

(a) *Euclidean distance*: $d = \sqrt{(i-k)^2 + (j-l)^2}$ or $d = |\mathbf{pq}|$.

(b) *taxicab distance*: $d = |k - i| + |j - l|$ or $d = \sum_{i=1}^{n} |q_i - p_i|$.

This distance function gets its name from the fact that a taxi driving from one location to another along orthogonal streets would drive that distance.

(c) *max distance*: $d = \max(|k - i|, |j - l|)$ or $d = \max_{1 \le i \le n}\{|q_i - p_i|\}$.

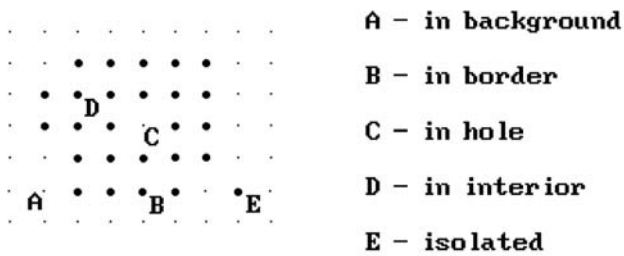


Figure 2.4. Examples of discrete concepts.

All three of these distance functions define a metric on $\mathbf{Z}^n$, called the *Euclidean*, *taxicab*, and *max metric*, respectively. (They actually also define a metric on $\mathbf{R}^n$.) For example, consider the points $\mathbf{p} = (2,1)$ and $\mathbf{q} = (5,3)$. Then the distances between these two points are:

Euclidean distance:	$\sqrt{13}$
Taxicab distance:	5
Max distance:	3

The points that are a distance of 1 from a given point are its 4-neighbors when we use the taxicab distance and the 8-neighbors when we use the max distance.

Finally, the rest of this chapter deals with two-dimensional sets and, unless stated otherwise, all our sets will be (discrete) subsets of some given picture $\mathbf{P}$ in $\mathbf{Z}^2$. We shall use the terminology above.

2.3 Border-Following Algorithms

Algorithms that can compute the borders of regions in a picture are important in a variety of places, in particular in animation. We describe one such algorithm here to give the reader a flavor of what they are like. See [RosK76] for more details. Other contour-following algorithms are described in [Pavl82]. See also [Herm98].

Assume that each point of a picture has a value associated to it and that in our case this is either 0 or 1, with the region of interest in it being the points with value 1. We shall show pictures by showing the values at their points and, to simplify the discussion, we often identify the point with its value. ("Setting a point to 3" will mean setting its value to 3.)

Definition. If $\mathbf{C}$ is a component of $\mathbf{S}$, $\mathbf{D}$ a component of $\mathbf{S}^c$, then the ***D**-border of **C*** is the set of points of $\mathbf{C}$ that are adjacent to a point of $\mathbf{D}$.

For example, consider the connected set $\mathbf{S}$ of 1's below:

$$\begin{array}{ccccc} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \end{array}$$

Let $\mathbf{D}_1$ and $\mathbf{D}_2$ be the components of $\mathbf{S}^c$ that contain the left and right "0," respectively. Then

$$\begin{array}{ccc} \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array} \qquad \text{and} \qquad \begin{array}{cccc} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & & & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{array}$$

are the $\mathbf{D}_1$-border and $\mathbf{D}_2$-border of $\mathbf{S}$, respectively.

Algorithm 2.3.1 is one *border-following* algorithm.

Input : an 8-component **C** of **S**, a 4-component **D** of $\mathbf{S}^c$
4-adjacency
adjacent points $c \in \mathbf{C}$, $d \in \mathbf{D}$

Output: all points of **D-**border of **C**

Follow the steps below:

(1) If **C** is an isolated point, then the **D**-border is just c. Change c to 4 and stop. Otherwise, change c to 3 and d to 2.

(2) List the 8-neighbors of c in clockwise order starting with d and ending with first occurrence of 1, 3, or 4:

$$e_1 , e_2 , \ldots , e_k$$

(3) If c is 3, e_k is 4, and e_h is 2 for some $h < k$, then set c to 4, e_h to 0, and stop; otherwise, set c to 4 (only if it was 1 and not 3), take e_k as the new c, take e_{k-1} as the new d, and go back to (2).

When the algorithm stops, the 4's will be the **D-**border of **C.**

Algorithm 2.3.1. A border-following algorithm.

2.3.1 Example. We show the various stages of Algorithm 2.3.1 for a set **S**. The values of points in the pictures below are shown in boldface and **S** starts off as the points marked with 1's. The current c and d are shown in parentheses to the left of the point to which they refer. The numbering of the 8-neighbors of c is shown in parentheses to the right of the point. We also show the value of k that we get in step (2) of the algorithm.

(d)**0**(1)	**0**(2)	**0**		**2**	(d)**0**(1)	**0**(2)		**2**	**0**		**0**	**0**		**2**		**0**	**0**		
(c)**1**	**1**(3)	**0**	→	**3**	(c)**1**	**0**(3)	→	**3**	**4**(8)		(d)**0**(1)	**0**(2)	→	**3**		(c)**4**	**0**	→
1	**0**	**1**		**1**	**0**	**1**(4)		**1**	**0**(7)		(c)**1**	**0**(3)		**1**(2)		(d)**0**(1)	**4**	
								0	**0**(6)		**0**(5)	**0**(4)		**0**		**0**	**0**	
k = 3				k = 4				k = 8						k = 2				

0	**2**	**0**	**0**		**0**(2)	**2**(3)	**0**(4)	**0**		**0**	**0**	**0**	
0(6)	**3**(7)	**4**	**0**	→	(d)**0**(1)	(c)**3**	**4**(5)	**0**	→	**4**	**4**	**0**	
0(5)	(c)**1**	(d)**0**(1)	**4**		**0**	**4**	**0**	**4**		**4**	**0**	**4**	
0(4)	**0**(3)	**0**(2)	**0**										
	k = 7					k = 5, h = 3							

Finally, the configuration

$$\begin{array}{ccccc} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ & & & \mathbf{1} & \end{array}$$

shows that the choice of adjacency is important. The algorithm fails if **C** is a 4-component and must be changed. See [RosK76].

Algorithm 2.3.1 can be used to find **all** border points of a set **S**. It provides a way of marking its border points so that one can then fill the interior of **S** using a fill algorithm of the type discussed in the next section.

2.4 Fill Algorithms

Contour-filling algorithms are used in many places. For example, in pattern recognition problems integrals may have to be computed over areas; in photo typesetting, fonts are described by contours that are later filled; in animation, the cel painter who fills figures has the next most time-consuming job after the animator.

There are two broad classes of such algorithms – *polygon-based* (*edge-filling*) algorithms and *pixel-based* algorithms. The former can be used in the case where the regions to be filled are defined by polygons and we can use the equations for the edges. The latter are, in a sense, more general because they can be used both for polygonal regions and also arbitrary regions whose boundaries are defined on the pixel level.

There is also a distinction as to how the algorithm decides whether a point is in the interior of a region. Some use a *parity check* that is based on the fact that lines intersect a closed curve an even number of times (if one counts intersections at certain special points such as at points of tangency correctly). This test is always used in case of polygon-based algorithms, but can also be used for pixel-based ones. Other algorithms, called *seed fill* algorithms, use connectivity methods. Here it is assumed that one is given a starting point or seed. Then one sees which pixels can be reached from this one without crossing the contour. The bounding curves can be quite general. This approach applies only to pixel-based algorithms. Also, one needs to know an interior point. This is okay in interactive situations (where one picks one using a mouse, for example), but if one wants to automate the process, note how border-following algorithms become relevant.

In this section we shall describe the pixel-based seed fill algorithms. Section 2.9.1 will look at polygon-based fill algorithms.

The Flood Fill Problem: Given **distinct** colors c and c′, a set **A** of the same color c bounded by points whose colors are **different** from c and c′, find an algorithm that changes all points of **A** and **only** those to the color c′.

An algorithm that solves this problem is called a *flood fill* algorithm. There are a number of related fill problems and associated algorithms. For example, *boundary fill* algorithms assume that all points of the boundary have the **same** color, which is **dif-**

```
procedure BFA (integer x, y)
if Inside (x,y) then
    begin
        Set (x,y);
        BFA (x,y − 1);  BFA (x,y + 1);
        BFA (x − 1,y);  BFA (x + 1,y);
    end;
```

Algorithm 2.4.1. The basic fill algorithm.

ferent from the color inside the region, where the *boundary* of a set S means here the set of points of S^c that are adjacent to S.

In the algorithms of this section, the Boolean-valued function **Inside**(x,y) determines whether or not the pixel at (x,y) has the property one wants. The procedure **Set**(x,y) sets the value of the pixel at (x,y) to its desired value. For example, to get a flood fill algorithm let **Inside**(x,y) be true if the value of the pixel at (x,y) agrees with the value of the pixels in the region and let **Set**(x,y) set the pixel value to its new value (the same as **Draw**(x,y,c′)). Using the functions **Inside** and **Set** will make our algorithms more general and applicable to a variety of fill algorithms. There is one constraint on the **Inside** function however: **Inside**(x,y) **must** return **false** after an operation **Set**(x,y).

Assume 4-adjacency is chosen and that our regions are 4-connected. The BFA procedure in Algorithm 2.4.1 shows that the basic idea behind a fill algorithm is very simple. Notice that 4-connected is important and that the algorithm will not work if the region is not 4-connected.

Although the BFA algorithm is simple, the recursion is expensive. One of the earliest nonrecursive algorithms is due to Smith ([Smit79]). It is not very efficient because pixels are visited twice, but many of the better algorithms are based on it. It will be worthwhile to describe Smith's algorithm, Algorithm 2.4.2, first before we present the one due to [Fish90b]. In this algorithm and the next, the constants XMIN, XMAX, YMIN, and YMAX define the minimum and maximum values for the x- and y-coordinates of pixels in the viewport. The procedures **Push** and **Pop** push and pop a pair (x,y) onto and from a stack, respectively. The function **StackNotEmpty** tests whether this stack is empty or not. The procedures **Inside** and **Set** are as described above.

For example, suppose that in Figure 2.5 our starting point is (7,3). After the first **FillRight** command the two-pixel segment from (7,3) to (8,3) would have been filled. The **FillLeft** command would fill (6,3). The **ScanHi** command would place the pixel coordinates (6,4) and (8,4) on the stack in that order. The **ScanLo** command would add (6,2). The segments of the region that (6,4), (8,4), and (6,2) belong to are usually called "shadows." The point of the **ScanHi** and **ScanLo** procedures is to find these shadows that still need to be filled. We now return to the beginning of the main **while** loop, pop (6,2), and make that our new starting point. The next **FillRight** and **FillLeft** would fill the segment from (2,2) to (8,2). The **ScanHi** and **ScanLo** would

```
{ Global variables }
integer x, y, lx, rx;
a stack of pixel coordinates (x,y);

procedure Fill (integer seedx, seedy)
begin
    x := seedx;   y := seedy;
    if not (Inside (x,y)) then Exit;
    Push (x,y);
    while StackNotEmpty () do
        begin
            PopXY ();
            if Inside (x,y) then
                begin
                    FillRight ();  FillLeft ();      { Fill segment containing pixel }
                    ScanHi ();  ScanLo ();          { Scan above and below current segment }
                end
        end
end;

procedure FillRight ()
begin
    integer tx;

    tx := x;
    { Move right setting all pixels of segment as we go }
    while Inside (tx,y) and (tx ≤ XMAX) do
        begin
            Set (tx,y);  tx := tx + 1;
        end;
    rx := tx − 1;  { Save index of right most pixel in segment }
end;

procedure FillLeft ()
begin
    integer tx;

    tx := x;
    { Move left setting all pixels of segment as we go }
    while Inside (tx,y) and (tx ≥ XMIN) do
        begin
            Set (tx,y);  tx := tx − 1;
        end;
    lx := tx + 1;  { Save index of left most pixel in segment }
end;
```

Algorithm 2.4.2. The Smith seed fill algorithm.

```
procedure ScanHi ()
{ Scan the pixels between lx and rx in the scan line above the current one.
  Stack the left most of these for any segment of our region that we find.
  We do not set any pixels in this pass. }
begin
    integer tx;

    if  y + 1 > YMAX  then  Exit;
    tx := lx;
    while  tx ≤ rx  do
        begin
            { Scan past any pixels not in region }
            while  not (Inside (tx,y + 1)) and (tx ≤ rx)  do  tx := tx + 1;
            if  tx ≤ rx  then
                begin
                    Push (tx,y + 1);
                    { We just saved the first point of a segment in region.
                      Now scan past the rest of the pixels in this segment. }
                    while  Inside (tx,y + 1) and (tx ≤ rx)  do  tx := tx + 1;
                end;
        end
end;

procedure ScanLo ()
{ Scan the pixels between lx and rx in the scan line below the current one.
  Stack the left most of these for any segment of our region that we find.
  We do not set any pixels in this pass. }
begin
    integer tx;

    if  y − 1 < YMIN  then  Exit;
    tx := lx;
    while  tx ≤ rx  do
        begin
            { Scan past any pixels not in region }
            while  not (Inside (tx,y − 1)) and (tx ≤ rx)  do  tx := tx + 1;
            if  tx ≤ rx  then
                begin
                    Push (tx,y − 1);
                    { We just saved the first point of a segment in region.
                      Now scan past the rest of the pixels in this segment. }
                    while  Inside (tx,y − 1) and (tx ≤ rx)  do  tx := tx + 1;
                end;
        end
end;
```

Algorithm 2.4.2. *Continued*

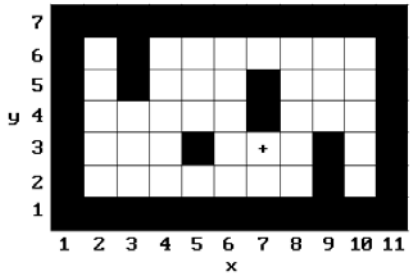


Figure 2.5. A fill algorithm example.

Figure 2.6. Pixel shadows.

put (2,3) and (6,3) on the stack. The loop would start over and pop (6,3). This time, since (6,3) has already been filled, we immediately jump back to the beginning and pop (2,3), and so on.

The problem with Smith's basic algorithm is that we look at some pixels twice, as we saw in the case of (2,3) in the previous example. This happens because we automatically put coordinates from both the line above and the line below the current one on the stack. When we then, say, deal with the line above, the algorithm will have us look at the current line again because it will be the line below that one. For a fast algorithm we need to prevent this duplicate effort. Algorithm 2.4.3 from [Fish90b] involves more bookkeeping because it differentiates between the three different types of possible shadows shown in Figure 2.6, but it will read each pixel only slightly more than once on the average and also has good worst-case behavior. Fishkin points out that it is optimal if the region has no holes.

An alternative improvement to Smith's seed fill algorithm is described by Heckbert in [Heck90b].

Finally, another distinction that is made between flood fill algorithms is whether we are dealing with *hard* or *soft* area flooding. The algorithms we have described so far were hard area flooding, which basically assumed that the region to be filled was demarcated by a "solid" boundary, that is, a curve of pixels all of the same color. Such a boundary would be a jagged curve. To get a smoother looking boundary one typically would blur or "shade" it by assigning a gradation of colors in a neighborhood of it. (The causes of the "jaggies" and solutions to the aliasing problem are discussed later in Section 2.6.) If boundaries are shaded, then we would like filling algorithms to maintain this shading. Soft area flooding refers to algorithms that do this and leave any "shading" intact. Smith's paper [Smit79] is a good reference for both hard and soft area flooding. The *tint fill* algorithm he describes in that paper is a soft area flooding algorithm.

There are other types of pixel-based fill algorithms. Pavlidis [Pavl82] describes a parity check type algorithm. Rogers [Roge98] describes various algorithms for filling regions bounded by polygons that he calls "edge fill" algorithms.

```
direction = (−1,+1);
stackRec = record                { a stackRec records the data for one shadow }
    integer    myLx, myRx,       { endpoints of this shadow }
               dadLx, dadRx,     { endpoints of my parent }
               myY;              { scan line of shadow }
    direction  myDirection;      { −1 means below parent, +1 means above }
end;

{ Global variable }
stack of stackRec shadowStack;

procedure Fill (integer seedx, seedy)
begin
    label      1, 2;
    integer    x, lx, rx, dadLx, dadRx, y;
    direction  dir;
    boolean    wasIn;

    Initialize shadowStack to empty;

    Find the span [lx,rx] containing the seed point;

    Push (lx,rx,lx,rx,seedy+1,1);
    Push (lx,rx,lx,rx,seedy−1,−1);

    while StackNotEmpty () do
        begin
1:          Pop ();
            if (y < YMIN) or (y > YMAX) then Goto 1;
            x := lx + 1;
            wasIn := Inside (lx,y);
            if wasIn then
                begin
                    Set (lx,y);   lx := lx − 1;

                    { If the left edge of the shadow touches a span, then move to its
                      left end setting pixels as we go }
                    while Inside (lx,y) and lx ≥ XMIN do
                        begin
                            Set (lx,y);   lx := lx − 1;
                        end
                end;

            { Start the main loop. Moving to the right starting from the current position x,
              if wasIn is true, then we are inside a span whose left edge is at lx. }
```

Algorithm 2.4.3. The Fishkin seed fill algorithm.

```
            while x ≤ XMAX do
                if wasIn
                    then
                        begin
                            if Inside (x,y)
                                then Set (x,y) { was inside and still inside }
                                else
                                    begin
                                        { was inside but not anymore, i.e., we
                                          just passed the right edge of a span }
                                        Stack (dadLx,dadRx,lx,x−1,y,dir);
                                        wasIn := false;
                                    end
                        end
                    else
                        begin
                            if x > rx then Goto 2;
                            if Inside (x,y) then
                                begin
                                    { we weren't inside but are now, i.e.,
                                      we just found the left edge of a new span }
                                    Set (x,y);
                                    wasIn := true;
                                    lx := x;
                                end
                            x := x + 1;
                        end;

2:          if wasIn then
                { we just hit the edge of the viewport while in a span }
                Stack (dadLx,dadRx,lx,x−1,y,dir);
        end
end;

boolean function StackNotEmpty ()
{ Returns true if shadowStack is empty and false otherwise }

procedure Push (integer myl, myr, dadl, dadr, y; direction dir)
{ Pushes record onto shadowStack }

procedure Pop ()
{ Pops top of shadowStack into local variables lx, rx, dadLx, dadRx, y, dir }

procedure Stack (integer dadLx, dadRx, lx, rx, y; direction dir)
{ Pushes an extra shadow onto shadowStack, given a newly discovered span
  and its parent. This is where the three types of shadows are differentiated. }
begin
```

Algorithm 2.4.3. *Continued*

```
    integer pushrx, pushlx;

    pushrx := rx + 1;   pushlx := lx − 1;
    Push (lx,rx,pushlx,pushrx,y+dir,dir);
    if  rx > dadRx  then  Push (dadRx+1,rx,pushlx,pushrx,y−dir,dir);
    if  lx < dadLx  then  Push (lx,dadLx−1,pushlx,pushrx,y−dir,dir);
end;
```

Algorithm 2.4.3. *Continued*

2.5 Generating Discrete Curves

Now we start a central topic of this chapter, namely, curves and the problem that one runs into when one tries to represent them with a discrete set of points. Clearly, we want any mapping of continuous structures into discrete ones to preserve the visual shape properties, such as smoothness and uniform thickness, as much as possible but this is not easy. We shall look at the problem of defining and generating discrete lines first and then conics.

Lines, or more accurately segments, are the most basic of computer graphics objects because most modeling systems use linear approximations to all objects so that displaying them reduces to drawing lots of lines. It is possible to actually give a formal definition of a discrete "straight" line (see [ArcM75] and [BoLZ75]). Not surprisingly, such definitions get complicated, but from a practical point of view we are not really interested in a definition. Rather, we are happy with an algorithm that generates a satisfactory set of points for a line. What is satisfactory? Well, that is not very precise, but some attributes that we want the generated discrete lines to have are:

(1) Visually, the line should appear as straight as possible.
(2) The line should start and end accurately, so that, for example, if several contiguous line segment are drawn, then there is no gap between them.
(3) Each line should appear to have an even visual thickness, that is, it should have as constant a density as possible, and this thickness should be independent of its length and slope.
(4) The conversion process must be fast.

In Sections 2.5.1–2.5.3 we look at line-drawing algorithms for the monochrome case, that is, where the raster is an array of 0's and 1's and the line consists of those pixels that are set to 1. Section 2.6 looks at some deeper problems that one encounters in the process of discretizing continuous objects and making them look smooth. Section 2.9.1 looks at a scan line algorithm for lists of lines and fill algorithm for polygons.

Conics are the next most common curve after the "straight" line. The circle is one obvious such curve, but the other conics are also encountered frequently. Their geo-

metric properties and relatively low degree (when compared with the popular cubic splines) make them attractive for use in designing shapes such as fonts. Because of this, a great deal of effort has been spent on devising efficient algorithms for computing them. We shall look at a few of these in Sections 2.9.2 and 2.9.3.

Because one common theme of some of the algorithms that generate discrete curves is derived from the geometric approach to solving differential equations, we start with that subject.

2.5.1 Digital Differential Analyzers

Consider the basic first order differential equation of the form

$$\frac{dy}{dx} = f(x,y) = \frac{g(x,y)}{h(x,y)}. \tag{2.1}$$

If y(x) is any solution, then f(x,y(x)) specifies the slope of the graph of y(x) at the point (x,y(x)). In other words, if one thinks of the function f as specifying a *vector field* over the entire plane (to (x,y) in the plane we associate the vector (1,f(x,y))), then solving equation (2.1) corresponds to finding a parameterized curve $x \to (x,y(x))$ whose tangent vectors agree with the vectors from this vector field. Mathematicians call such curves "integral curves." In general, given a vector field, a curve whose tangent vectors agree with the vectors of that vector field at every point on the curve is called an *integral curve* for that vector field. See Figure 2.7. The reason for this nomenclature is that solving for the curve basically involves an integration process.

This idea of vector fields and integral curves leads to the following approach to finding numerical solutions to differential equations called *Euler's method*. Suppose that we want the solution to pass through $\mathbf{p}_0 = (x_0,y_0)$. Since we know the tangent vector to the solution curve there and since the tangent line is a good approximation to the curve, moving a small distance along the tangent, say by $\varepsilon(h(x_0,y_0),g(x_0,y_0))$, where ε is a small positive constant, will put us at a point $\mathbf{p}_1 = (x_1,y_1)$, which hopefully is not too far away from an actual point on the curve. Next, starting at $\mathbf{p}_1$ we repeat this process. In general, let

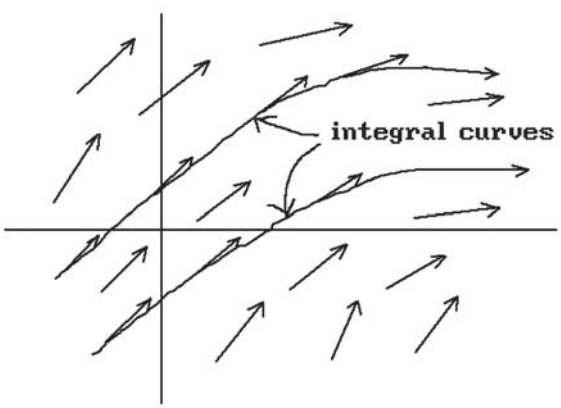


Figure 2.7. Integral curves of a vector field.

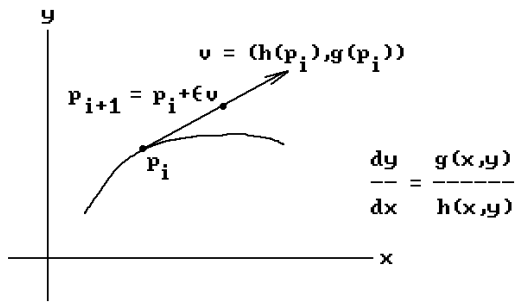


Figure 2.8. Generating an integral curve approximation.

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \varepsilon(h(x_i, y_i), g(x_i, y_i)). \tag{2.2}$$

See Figure 2.8. The sequence of points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ obtained in this way becomes our approximation to the actual integral curve passing through $\mathbf{p}_0$.

Unfortunately, as we move from point to point we start drifting away from the actual curve and so our approximation will, in general, get further and further away from the true solution. To make the method work we need to compensate for any possible error as we move along. There are some very good algorithms that solve differential equations with basically this approach by using some fancy error-correcting terms. For more information see a text on numerical analysis such as [ConD72] or [DahB74].

Discrete curve-drawing algorithms that are based on the qualitative solutions to differential equations as described above are called *digital differential analyzer* or *DDA* type algorithms. Let us see what we get in the special case of straight lines.

The differential equation for the straight line that passes through the points (x_0,y_0) and (x_1,y_1) is

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = \frac{\varepsilon \Delta y}{\varepsilon \Delta x},$$

where $\Delta y = y_1 - y_0$, $\Delta x = x_1 - x_0$, and ε is any positive real number. Specializing the approximation formula, equation (2.2), to this differential equation gives us a sequence of points $\mathbf{p}_i$ defined by

$$\mathbf{p}_{i+1} = \mathbf{p}_i + (\varepsilon \Delta x, \varepsilon \Delta y). \tag{2.3}$$

In fact, the points $\mathbf{p}_i$ we generate will actually fall on the line, so that we do not have to worry about compensating for any errors. Although this may seem like overkill in the case of continuous lines, it does motivate an approach to generating discrete lines that leads to an extremely efficient such algorithm (the Bresenham algorithm). Note that if $\mathbf{q}_i$ is the point with integer coordinates that is gotten from $\mathbf{p}_i$ by rounding each real coordinate of $\mathbf{p}_i$ to its nearest integer, then the points $\mathbf{q}_i$ define a discrete curve that is an approximation to the continuous one. The key to getting an efficient line-drawing algorithm is to be able to compute the $\mathbf{q}_i$ efficiently.

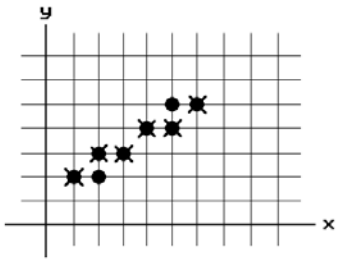


Figure 2.9. Simple and symmetric DDA generated lines.

× simple DDA • symmetric DDA

In the continuous case one always generates points on the line no matter what ε is chosen but the choice of ε does matter when generating discrete lines. We now look at two possible choices for ε. These give rise to what are called the *simple* and *symmetric DDA*, respectively.

Let $m = \max(|\Delta x|,|\Delta y|)$.

The simple DDA: Choose $\varepsilon = 1/m$.
The symmetric DDA: Choose $\varepsilon = 2^{-n}$, where $2^{n-1} \le m < 2^n$.

2.5.1.1 Example. Suppose that we want to generate the discrete line from (1,2) to (6,5).

Solution. In this case $(\Delta x,\Delta y) = (5,3)$. For the simple DDA we have

$$\varepsilon = 1/5,\ \varepsilon\Delta x = 1,\ \varepsilon\Delta y = 3/5, \quad \text{and} \quad \mathbf{p}_{i+1} = \mathbf{p}_i + (1,3/5).$$

In the case of the symmetric DDA, we have

$$\varepsilon = 1/8,\ \varepsilon\Delta x = 5/8,\ \varepsilon\Delta y = 3/8, \quad \text{and} \quad \mathbf{p}_{i+1} = \mathbf{p}_i + (5/8,3/8).$$

The points that are generated are shown in Figure 2.9. The points of the simple DDA are shown as ×'s and those of the symmetric DDA are shown as solid circles.

2.5.2 The Bresenham Line-Drawing Algorithm

Although the DDA algorithms for drawing straight lines are simple, they involve **real** arithmetic. Some simple modifications result in an algorithm that does only **integer** arithmetic, and only additions at that.

Note that in the case of the simple DDA, either x or y will always be incremented by 1. For simplicity, assume that the start point of our line is the origin. If we also restrict ourselves to lines whose endpoint is in the first octant in the plane, then it will be the x that always increases by 1. Therefore, we only need to worry about computing the y coordinates efficiently.

Suppose therefore that we want to draw a line from (0,0) to (a,b), where a and b are integers and $0 \le b \le a$ (which puts (a,b) into the first octant). Using equation (2.3), the points $\mathbf{p}_i$, $0 \le i \le a$, generated by the simple DDA are then defined by

$$\mathbf{p}_i = \mathbf{p}_{i-1} + (1, b/a) = (i, i\, b/a),$$

and the discrete line consists of the points (i,y_i), where y_i is the real number i(a/b) rounded to the nearest integer.

Now the y coordinates start at 0. At what point does y_i become 1? To answer this question, we must compute b/a, 2b/a, 3b/a, . . . , and watch for that place where these values become bigger than 1/2. Furthermore, the y_i will then stay 1 until these values become bigger than 3/2, at which time y_i will become 2. Since we want to avoid doing real arithmetic, note that we do not really care what the actual values are but only care about when they get bigger than 1/2, 3/2, 5/2, ... This means that we can multiply through by 2a and the answer to the question as to when y_i increases by 1 is determined by when 2b, 4b, 6b, . . . become bigger than a, 3a, 5a, ... Since computers can compare a number to 0 in less time than it takes to compare it to some other number, we shall start off by subtracting a. Our first question now is

"When does 2b – a, 4b – a, 6b – a, . . . become bigger than 0?"

and only involves repeated integer additions of 2b to an initial sum d = 2b – a. After the sum d has become bigger than 0 and y has switched to 1, we need to check when the sum becomes bigger than 2a. By subtracting 2a, we again only need to keep checking for when the sum gets to be bigger than 0 by successive additions of 2b. In general, whenever y is incremented by 1, we subtract 2a from the current sum d. In that way we always need to check d simply against 0. For example, suppose we want to draw the line from (0,0) to (15,3). In this case, 2a = 30, 2b = 6, and the initial d is 6 – 15 = –9. The table below shows the points (x_i,y_i) that are drawn and the changes to the sum d as i ranges from 0 to 8:

i	0	1	2	3	4	5	6	7	8
d		–9	–3	–27	–21	–15	–9	–3	–27
(x_i,y_i)	(0,0)	(1,0)	(2,0)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,2)

The code in Algorithm 2.5.2.1 implements the algorithm we have been describing.

In our discussion above we have restricted ourselves to lines that start at the origin and end in the first octant. Starting at another point simply amounts to adding a constant offset to all the points. Lines that end in a different octant can be handled in a similar way to the first octant case – basically by interchanging the x and y. What this boils down to is that an algorithm which handles all lines is not much harder, involving only a case statement to separate between the case where the absolute value of the slope is either larger or less than or equal to 1.

We have just described the basis for the *Bresenham line-drawing algorithm* ([Bres65]). It, or some variation of it, is the algorithm that is usually used for drawing straight lines. Bresenham showed in [Bres77] that his algorithm generated the best-fit discrete approximation to a continuous line. The variation that is Algorithm 2.5.2.2

Code for drawing the discrete line from (0,0) to the point (a,b) in the first octant:

```
begin
    integer d, x, y;

    d := 2*b – a;
    x := 0;
    y := 0;
    while true do
        begin
            Draw (x,y);
            if x = a then Exit;
            if d ≥ 0 then
                begin
                    y := y + 1;
                    d := d – 2*a;
                end;
            x := x + 1;
            d := d + 2*b;
        end
end
```

Algorithm 2.5.2.1. Basic line-drawing algorithm.

comes from [Heck90c] and works for all lines. It generates the same points as the original Bresenham line-drawing algorithm but is slightly more efficient.

To further improve the efficiency of DDA-based algorithms, there are n-step algorithms that compute several pixels of a line at a time. The first of these was based on the idea of double stepping. See [RoWW90] or [Wyvi90]. There are also algorithms that use a 3- or 4-step process. See [BoyB00] for an n-step algorithm that automatically uses the optimal n and claims to be at least twice as fast as earlier ones.

2.5.3 The Midpoint Line-Drawing Algorithm

Because drawing lines efficiently is so important to graphics, one is always on the lookout for better algorithms. Another well-known line-drawing algorithm is the so-called *midpoint line-drawing algorithm*. It produces the same pixels as the Bresenham algorithm, but is better suited for generalizing to other implicitly defined curves such as conics and also to lines in three dimensions (see Section 10.4.1). The general idea was first described in [Pitt67] and is discussed in greater detail by [VanN85].

Assume that a **nonvertical** line **L** is defined by an equation of the form

$$f(x,y) = ax + by + c = 0,$$

```
procedure DrawLine (integer x0, y0, x1, y1)
begin
    integer dx, ax, sgnx, dy, ay, sgny, x, y, d;

    dx := x1 – x0;   ax := abs (dx)*2;   sgnx := Sign (dx);
    dy := y1 – y0;   ay := abs (dy)*2;   sgny := Sign (dy);
    x  := x0;   y := y0;
    if  ax > ay
        then              { x increases faster than y }
            begin
                d := ay – ax/2;
                while  true  do
                    begin
                        Draw (x,y);
                        if  x = x1  then  Exit;
                        if  d ≥ 0  then
                            begin
                                y := y + sgny;   d := d – ax;
                            end;
                        x := x + sgnx;   d := d + ay;
                    end
            end
        else              { y increases faster than x }
            begin
                d := ax – ay/2;
                while  true  do
                    begin
                        Draw (x,y);
                        if  y = y1  then  Exit;
                        if  d ≥ 0  then
                            begin
                                x := x + sgnx;   d := d – ay;
                            end;
                        y := y + sgny;   d := d + ax;
                    end
            end
end;

integer function Sign (real x)

if   x < 0   then   return (–1)
             else   return (+1);
```

Algorithm 2.5.2.2. A Bresenham line-drawing algorithm.

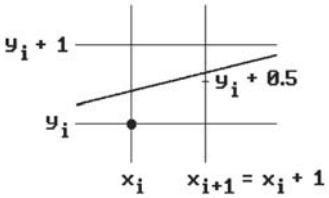


Figure 2.10. The midpoint line-drawing algorithm decision.

where $-b \geq a \geq 0$. This assumption implies that our line has slope between 0 and 1. Lines with other slopes are handled in a symmetric way like in Bresenham's algorithm. Vertical lines are a very special case that would be handled separately. Another important consequence of our assumptions is that f(x,y) will be positive for points (x,y) below the line and negative for points above the line. Also like in the Bresenham algorithm, the points $\mathbf{p}_i = (x_i,y_i)$ that we will generate for the line will have the property that the x-coordinate will be incremented by 1 each time, $x_{i+1} = x_i + 1$, so that we only have to determine the change in the y coordinate.

See Figure 2.10. The only possible value for y_{i+1} is y_i or $y_i + 1$. The decision will be based on the sign of

$$d_i = f(x_i + 1, y_i + 0.5).$$

If $d_i > 0$, then the line **L** crosses the line $x = x_i + 1$ above the point $(x_i + 1, y_i + 0.5)$ and we need to let y_{i+1} be $y_i + 1$. If $d_i < 0$, then we should let y_{i+1} be y_i. If $d_i = 0$, then either y value would be satisfactory. We shall choose y_i in that case. Choosing our points p_i in this way is what constitutes the basic idea behind the midpoint line-drawing algorithm. The only thing that is left is to describe some optimizations that can be made in the computations. First, the d_i can be computed efficiently in an incremental way. By definition, if $d_i \leq 0$, then

$$\begin{aligned} d_{i+1} &= f(x_i + 2, y_i + 0.5) \\ &= a(x_i + 2) + b(y_i + 0.5) + c \\ &= d_i + a. \end{aligned}$$

On the other hand, if $d_i > 0$, then

$$\begin{aligned} d_{i+1} &= f(x_i + 2, y_i + 1.5) \\ &= a(x_i + 2) + b(y_i + 1.5) + c \\ &= d_i + a + b. \end{aligned}$$

This shows that the next value of the decision variable can be computed by simple additions to the previous value. If our line **L** starts at the point $\mathbf{p}_0 = (x_0,y_0)$, then

$$d_0 = f(x_0 + 1, y_0 + 0.5) = f(x_0, y_0) + a + b/2.$$

This gives us the starting value for the decision variable and all the rest are computed incrementally. We can avoid the fraction in the starting value and do all our compu-

tations using purely integer arithmetic by multiplying the equation for our line by 2, that is, let us use

$$F(x,y) = 2f(x,y) = 2(ax + by + c) = 0.$$

This has the effect of multiplying our starting value for the decision variable and its increments by 2. Since only the sign of the variable was important and not its value, we have lost nothing. Putting all this together leads to Algorithm 2.5.3.1.

Finally, before leaving the subject of line-drawing algorithms, we should point out that there are other such algorithms other than the ones mentioned here. For example, there are run-based line drawing algorithms. See [SteL00]. One thing to keep in mind

```
procedure DrawLine (integer x0, y0, x1, y1)
{ We have chosen the equation

        f (x, y)  =  (dy) x – (dx) y + c  =  0

  as the equation for the line.}
begin
    integer dx, dy, d, posInc, negInc, x, y;

    dx := x1 – x0;
    dy := y1 – y0;
    d   := 2*dy – dx;          { Initial value of decision variable }
    posInc := 2*dy;            { The increment for d when  d ≥ 0  }
    negInc := 2*(dy – dx);     { The increment for d when  d < 0  }
    x  := x0;   y := y0;
    Draw (x,y);
    while  x < x1  do
        begin
            if  (d ≤ 0)
                then  d := d + posInc
                else
                    begin
                        d := d + negInc;    y := y + 1;
                    end;
            x := x + 1;
            Draw (x,y);
        end
end;
```

Algorithm 2.5.3.1. The midpoint-line drawing algorithm.

though is that the time spent in line drawing algorithms is often dominated by the operation of setting pixels in the frame buffer, so that software improvements alone may be less important.

2.6 The Aliasing Problem

No matter how good a line drawing algorithm is, it is impossible to avoid giving most discrete lines a staircase effect (the "jaggies"). They just will not look "straight." Increasing the resolution of the raster helps but does not resolve the problem entirely. In order to draw the best looking straight lines one has to first understand the "real" underlying problem which is one of **sampling**.

The geometric curves and surfaces one is typically trying to display are continuous and consist of an infinite number of points. Since a computer can only show a finite (discrete) set of points, how one chooses this finite set that is to represent the object is clearly important. Consider the sinusoidal curve in Figure 2.11. If we sample such a sine wave badly, say at the points **A**, **B**, **C**, and **D**, then it will look like a straight line. If we had sampled at the points **A**, **E**, **F**, and **D**, then we would think that it has a different frequency.

> **The basic problem in sampling theory:** How many samples does one have to take so that no information is lost?

This is a question that is studied in the field of signal processing. The theory of the *Fourier transform* plays a big role in the analysis. Chapter 21, in particular Section 21.6, gives an overview of some of the relevant mathematics. For more details of the mathematics involved in answering the sampling problem see [GonW87], [RosK76], or [Glas95]. We shall only summarize a few of the main findings here and indicate some practical solutions that are consequences of the theory.

Definition. A function whose Fourier transform vanishes outside a finite interval is called a *band-limited* function.

One of the basic theorems in sampling theory is the following:

The Whittaker-Shannon Sampling Theorem. Let f(x) be a band-limited function and assume that its Fourier transform vanishes outside [–w,w]. Then f(x) can be reconstructed exactly from its samples provided that the sampling interval is no bigger than 1/(2w).

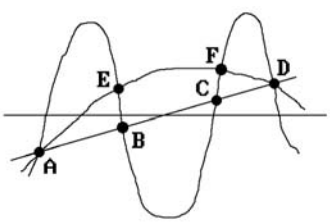


Figure 2.11. Aliasing caused by bad sampling.

If T is a sampling interval, then 1/T is called the *sampling frequency* and 1/(2T) is called the *Nyquist limit*. The Whittaker-Shannon Theorem says that if a function is sampled less often than its Nyquist limit, then a complete recovery is impossible. One says that the function is *undersampled* in that case. Undersampling leads to a phenomenon referred to as *aliasing*, where fake frequencies or patterns appear that were not in the original object. The two-dimensional situation is similar, but in practice one must sample a lot more because of limitations of available reconstruction algorithms.

Now in the discussion above, it was assumed that we were taking an infinite number of samples, something that we obviously cannot do in practice. What happens if we only take a finite number of samples? Mathematically, this corresponds to where we multiply the sampled result by a function that vanishes outside a finite interval. The main result is that it is in general **impossible** to faithfully reconstruct a function that has only been sampled over a finite range. To put it in another way, no function that is nonzero over only a finite interval can be band-limited and conversely, any band-limited function is nonzero over an unbounded set.

The practical consequences of the theory sketched above can be seen in lots of places. Aliasing is most apparent along edges, near small objects, along skinny highlights, and in textured regions. Ad hoc schemes for dealing with the problem may be disappointing because of the human visual system's extreme sensitivity to edge discontinuities (*vernier acuity*). Aliasing is also a problem in animation. The best-known example of temporal aliasing is the case of the wagon wheel appearing to reverse its direction of motion as it spins faster and faster. Other examples are small objects flashing off and on the screen, slightly larger objects appearing to change shape and size randomly, and simple horizontal lines jumping from one raster line to another as they move vertically. See Figure 2.12. This happens because objects fall sometimes on and sometimes between sampled points.

Jaggies do not seem to appear in television because the signal generated by a television camera, which is sampled only in the vertical direction, is already band-limited before sampling. A slightly out of focus television camera will extract image samples that can be successfully reconstructed on the home television set. People working in computer graphics usually have no control over the reconstruction process. This is part of the display hardware. In practice, antialiasing techniques are imbedded in algorithms (like line-drawing or visible surface determination algorithms). The approaches distinguish between the case of drawing isolated lines, lines that come from borders of polygons, and the interior of polygons.

There are essentially two methods used to lessen the aliasing problem. Intuitively speaking, one method treats pixels as having area and the other involves sampling at a higher rate. The obvious approach to the aliasing problem where one simply

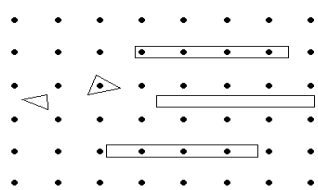


Figure 2.12. Objects appearing, disappearing, changing size.

increases the resolution of the display device is a special case of the latter. Mathematically, the two methods are

(1) prefiltering, and
(2) supersampling or postfiltering

Prefiltering. This amounts to treating each sample point as representing a finite area rather than simply a dot. Because lines often fall between pixels, this would avoid concentrating everything at a pixel in a hit-or-miss fashion. Mathematically, the process corresponds to applying a convolutional filter before sampling. One must make sure that the highest frequency of a signal in a scene does not exceed one-half the sampling rate.

Two widely used models for computing the area subtended by a pixel are

(1) One considers the image a square grid as in Figure 2.13 with the pixels in the centers of the squares.
(2) One computes the area using a weighting function similar to a Gaussian function. This in fact models the effect of the electron beam of a CRT and printing processes more closely. The pixels are larger and overlap. Details near the center now count more heavily than those near the edge.

Model (1) is easier than (2), but (2) produces better pictures. Internal details, such as highlights, are harder to handle.

In the case of boundaries of polygons we can use shading to suggest the position of the edges and can make the picture look as if it had higher resolution than it in fact has. Therefore, associate to each pixel an intensity proportional to the percentage of its area that is covered by the polygon. For example, if the intensity values ranged from 0 to 15, then we might assign pixel **A** in Figure 2.13 a value of 2 and pixel **B**, a value of 8. This approach could obviously substantially increase the amount of computation one has to do. However, by using an efficient approximation of the area that is needed, it turns out that all it takes is a slight modification to the Bresenham algorithm to get an efficient implementation of it, namely, the Pitteway-Watkinson algorithm. See [PitW80] or [Roge98].

Another approach for drawing antialiased lines treats the lines as having a thickness. An algorithm of this type is the Gupta-Sproull algorithm. See [GupS81], [Thom90], or [FVFH90]. It also starts with the standard Bresenham algorithm and then adds some checks for nearby pixels above and below each pixel that would be drawn by that algorithm.

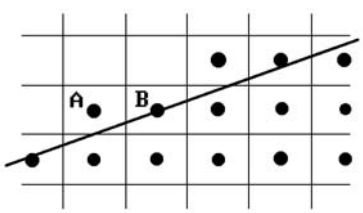


Figure 2.13. Pixel intensities based on percentage of area covered.

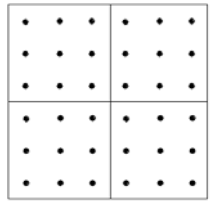


Figure 2.14. Supersampling with scaling factor 3.

Supersampling. Here we sample at more points than will actually be displayed. More precisely, we sample at n uniformly situated points within the region associated to each pixel and then assign the average of these values to the pixel. One usually oversamples the same amount in each direction so that $n = s^2$ for some *scaling factor* s. For example, to create a 512×512 image we would sample at 1536×1536 points if s is 3. The samples would be taken 1/3 of a pixel width apart. In Figure 2.14, each square corresponds to a pixel in the final image and the dots show the location of the nine samples per pixel.

Postfiltering. In supersampling the sample values for each pixel are **averaged**. This gives each sample the same weight. Postfiltering uses the same approach but allows each sample to have a different weight. Supersampling is therefore a special case of postfiltering. Different weighting or "window" functions can be used. For example, if we represent the weighting operation in matrix form with the ij'th entry being the weighting factor for the ij'th sample, then rather than using the supersampling matrix

$$(1/9)\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

we could use

$$(1/8)\begin{pmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{or} \quad (1/16)\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$

Mathematically, postfiltering corresponds to a convolution and filtering operation on the samples. The cost of generating an image with supersampling and postfiltering is proportional to the number of scan lines. The cost of calculations involving shading is proportional to the square of the number of scan lines. This means that the algorithm is particularly expensive for visible surface determination algorithms.

In conclusion, antialiasing techniques add a large amount of computation time to any algorithm that uses them. To minimize this extra work, one tries to do it only for areas where problems occur and makes no special computations for the rest. Of course, this assumes that one knows all about the picture, say a jar defined via many polygons. For lots more about antialiasing techniques see [FVFH90].

2.7 Halftoning, Thresholding, and Dithering

In contrast to antialiasing where we use multiple intensity levels to increase the resolution, *halftoning* (or *patterning*) is a technique for obtaining increased **visual** resolution with a minimum number of intensity levels. Basically, rectangular grids of pixels are treated as single pixels. This is how photographs are usually reproduced for magazines and books. For example, using a 2×2 grid we can get five different intensities. See Figure 2.15(a). Not all of the possible combinations are used (basically symmetric patterns are to be avoided) in order not to introduce unwanted patterns into the picture. Using the patterns in Figure 2.15(b) could easily introduce unwanted horizontal or vertical lines in a picture. Normally 2×2 or 3×3 grids are used.

Halftoning reduces the overall **spatial** resolution of a system. For example, the resolution of a 1024×1024 monitor would be reduced to 512×512 with 2×2 grids. This means that such a technique is best applied when the resolution of the original scene is less than that of the output device.

Another technique called *thresholding* deals with the problem where we have a digital image with the same resolution as our monochrome display device but with more intensity levels. The simplest form of thresholding is to use a fixed threshold for each pixel. If the intensity exceeds that value, the pixel is drawn white, otherwise it is drawn black. Such a simple scheme can lose a lot of detail. A more refined algorithm of this type is due to Floyd and Steinberg. See [Roge98].

Finally, *dithering* is a technique applying to monochrome displays that is used with halftoning or thresholding methods to smooth edges of objects by introducing random noise into the picture. It increases the visual resolution without reducing the spatial resolution. One adds a random error to each pixel value before comparing to the threshold value (if any has been selected). Good error patterns have to be chosen carefully. *Ordered dithering* is where a square **dither matrix** is added to the picture. Alternatively, rather than adding noise using the same threshold for each pixel one can vary the threshold. With this approach, an optimum 2×2 matrix has been shown to be

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}.$$

The entries of the matrix are used as the threshold for the corresponding pixel. There are recursive formulas for higher dimensional dither matrices. See [Roge98].

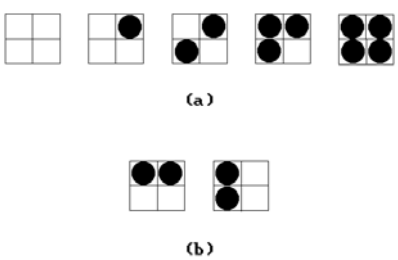


Figure 2.15. Halftone patterns.

2.8 Choosing the Coordinates of a Pixel

Before going on to discuss another scan conversion algorithm we pause to take up a subject that probably did not occur to the reader as being an issue. However, since pixels should be treated as having area, if we consider our image as corresponding to a grid as we have, where should the pixels be placed? Should they be at the intersection of the grid lines or in the center of the grid squares? Equivalently, when we consider scan lines, do their y-coordinates fall on integers or half-integers? Whatever choice one makes, it **does** matter. We summarize the conclusions of the excellent article by Heckbert [Heck90a].

The real issue here is how one maps reals to integers. Should one round or truncate? Rounding corresponds to placing pixels at the integers because the whole interval $[n - 0.5, n + 0.5)$ will map to n. Truncating corresponds to placing the pixels at half-integers because the whole interval $[n, n + 1)$ will map to n. To use an example, if one rounds, then the interval $[-.5, 2.5)$ maps to $\{0,1,2\}$, whereas if one truncates, then $[0,3)$ maps to $\{0,1,2\}$. The second approach is a cleaner choice because there are no .5's to worry about. By truncating one simplifies some mathematics. We shall therefore use the following correspondence if we need to map back and forth between the continuous and discrete world:

real c → **integer** n = Floor (c)
integer n → **real** (n + 0.5)

(Mathematically it is the Floor function that returns an integer whereas the Trunc function returns a real.) In two dimensions this means that when we have a pixel with coordinates (x,y), its center will be at continuous coordinates (x + 0.5,y + 0.5). Note that this was the choice we made when discussing antialiasing. Now we know why.

In the future, whenever we scan a **continuous** object the scan lines will fall on half-integers.

2.9 More Drawing Algorithms

2.9.1 Scan Converting Polygons

The Bresenham line-drawing algorithm discussed in Section 2.5.2 dealt with scan converting a **single** segment. There may be several segments we want to scan convert such as the boundary of a polygon. In that case one can use the **coherence** inherent in that problem and use an algorithm that is more efficient that simply scan converting each bounding edge separately.

Consider the edges in Figure 2.16. As we move down from the top of the picture one scan line at a time we do not need to compute the intersections of the edges with the scan line each time. These can be computed incrementally. Since not every edge will intersect current scan line, by using an *active edge list* (AEL), we do not have to look at every edge each time. Here are the steps to scan convert these edges efficiently:

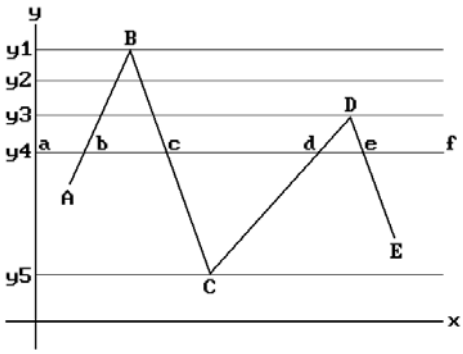


Figure 2.16. Scan converting a polygon.

Step 1. Associate a "bucket" to each scan line and initialize it to empty.
Step 2. Find the largest y value for each edge and put the edge into the corresponding scan line's bucket.
Step 3. For each edge **e** maintain the following information:

x – initially the x-coordinate of the highest point of the edge **e** (in general the x-coordinate x_e of the intersection of **e** with the current scan line)
dx – change in x from line to line (the reciprocal of the slope of the line)
dy – initially the number of scan lines crossed by **e**

Step 4. Initialize the active edge list to empty. Set y to the height of the top scan line.
Step 5. Add any edges in the bucket for y to the active edge list.
Step 6. For each edge in the active edge list draw (x,y), change the x to x + dx, and decrement dy. If dy becomes 0, then remove that edge from the list.
Step 7. Decrement y. If y is 0, then quit; otherwise, go back to Step 5.

In Figure 2.16, when we reach scan line y1, the edges **AB** and **BC** will be added to the active edge list. At scan line y2 nothing special happens. When we get to scan line y3, the edges **CD** and **DE** will be added to the list. Finally, at scan line y5 there are only the two edges **BC** and **CD** on the list and they will now be removed.

To avoid having fixed bucket sizes and limiting the amount of data for each scan line, one stores pointers only and stores all information sequentially in an array. Alternatively, one can use a **linked** list to be able to add and delete easily.

A problem related to scan converting lists of edges which is of more practical importance is scan converting **solid** polygons. This leads to polygon based fill algorithms. The pixel-based analog was already discussed earlier in Section 2.4.

Assume that XMIN, XMAX, YMIN, and YMAX are the minimum and maximum values for the x- and y-coordinates of pixels. The basic idea here is the following:

```
for i:=YMIN to YMAX do
    for j:=XMIN to XMAX do
        if Inside (polygon,j,i) then Draw (j,i);
```

The Boolean-valued function "Inside" counts the intersections of the line from (j,i) to (–∞,i) with the polygon. If this number is odd, then the function returns **true**, other-

wise it returns **false**. Of course, this algorithm is too slow. One needs to take **scan line coherence** into account. This leads us to what are called *ordered edge list* fill algorithms. They have the following general form:

```
for each scan line do
    begin
      Find all intersections of edges with the scan line;
      Sort the intersections by increasing x;
      Fill alternate segments;
    end;
```

For example, consider the scan line y4 in Figure 2.16. Notice how filling the alternate segments [b,c] and [d,e] does in fact fill what we want. That this works is justified by a parity type argument. An active edge list again helps. Algorithm 2.9.1.1 shows a more detailed version.

```
linerec = record
    real     x, dx;
    integer dy;
end;

linerecs = linerec list;

begin
    linerecs array [0.. ] edges;    { the edges of the polygons }
    linerecs   ael;                 { the active edge list }
    integer    y;

    Scan the polygon and set up the edges table;
    ael := nil;

    for  y:=YMIN to YMAX  do
        begin
            Add all edges in  edges [y]  to ael;
            if  ael ≠ nil  then
                begin
                    Sort ael by increasing x;
                    Fill pixels along y by scanning ael and filling alternate x segments;
                    Delete from ael edges for which  dy = 0 ;
                    Update each x in ael by dx;
                end
        end
end;
```

Algorithm 2.9.1.1. An ordered edge list fill algorithm.

The following points need to be made about Algorithm 2.9.1.1:

(1) The polygon is assumed to lie entirely in window.
(2) Horizontal edges need not be considered because they get filled automatically.
(3) There is a problem with parity at vertices unless one takes precautions.

To understand the parity problem at vertices consider Figure 2.16 again. At vertices, their x values would be listed twice in the active edge list. In the case of a local maximum like vertex **B** = ($x_{\mathbf{B}}$,$y_{\mathbf{B}}$), the algorithm would fill the segments [XMIN,$x_{\mathbf{B}}$], [$x_{\mathbf{B}}$,$x_{\mathbf{B}}$], and [$x_{\mathbf{B}}$,XMAX] on the scan line y = $y_{\mathbf{B}}$ to the background color, the color of the polygon, and the background color, respectively. This is as we would want it. On the other hand, when the algorithm gets to vertex **A** = ($x_{\mathbf{A}}$,$y_{\mathbf{A}}$), assuming that there was another edge below this vertex, it would fill [XMIN,$x_{\mathbf{A}}$] to the background color, [$x_{\mathbf{A}}$,$x_{\mathbf{BA}}$] to the color of the polygon, and [$x_{\mathbf{BA}}$, $x_{\mathbf{BC}}$] to the background color, etc. This is **not** correct. Furthermore, we cannot simply skip duplicate x-coordinates as we scan the active edge list. If we did, then vertices like **A** would be handled correctly, but the algorithm would now fail at local maxima and minima like **B**. The way that this parity problem is usually resolved is to shorten one of the (two) edges that meet in a vertex that is **not** at a local extremum. For example, change the lower vertex (x,y) of the upper edge to (x,y + 1) (leaving the upper vertex of the lower edge in tact). **No** shortening takes place at vertices that are local extrema. With this change to the edges, Algorithm 2.9.1.1 will now work correctly, but we need a test for when vertices are local extrema. Here is one:

```
if adjacent edges have the same sign for their slope
  then  the common vertex is not a local extremum
  else  test the opposite endpoints for whether or not they lie on the
        same side of the scan line as the vertex: if they do, then the
        vertex is a local extremum, otherwise, not
```

To see that a further test is required in the **else** case, consider Figure 2.17 and the two pairs of segments ([(–1,–1),(0,0)],[(0,0),(1,–1)]) and ([(–1,–1),(0,0)],[(0,0),(–1,1)]). In both pairs, the segments have opposite slopes, but (0,0) is a local extremum for the first pair but not for the second. One can tell the two apart however because the endpoints (–1,–1) and (–1,1) for the first pair lie on opposite sides of the scan line for (0,0), whereas the endpoints (–1,–1) and (1,–1) both lie on the same side of the scan line.

Finally, note that the ordered edge list fill algorithm "works" for polygons with self-intersections and/or holes. See Figure 2.18. One needs to understand what "works"

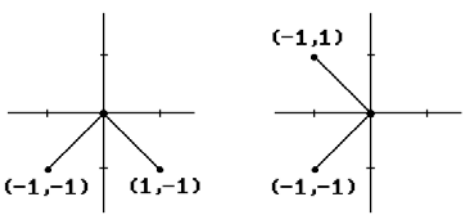
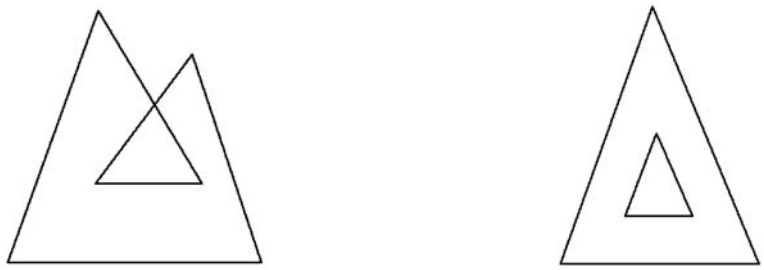


Figure 2.17. Testing for local extrema.



(a) Self-intersecting polygon (b) Polygon with hole

Figure 2.18. Various types of polygon.

means though. For example, the inside of the inner loop of Figure 2.18(a) will be drawn in the background color.

Looking ahead to Chapter 7, which is on visible surface determination, we can deal with multiple polygons here if we have an associated priority number with each, where having a higher priority means being in front of or not obscuring. In the algorithm above, as we go along we must now keep track of the polygon to which the "current" segment "belongs." One way to do this is to maintain the following additional data:

(1) covers – a Boolean array so that covers[i] is **true** for the ith polygon if it covers the current segment
(2) numcover – the number of polygons covering the current segment
(3) visiblePoly – a pointer to the foremost polygon, if any

As we move from segment to segment in a scan line, numcover is incremented or decremented appropriately. The array covers is initialized to **false** and every time that one runs into an edge of the ith polygon, covers[i] is negated. The pointer visiblePoly tells us the color of the current segment.

In conclusion, here are some points to consider when deciding on a fill algorithm. The main advantages of ordered edge list algorithms are that pixels are visited only once and they are well suited for shading algorithms since both ends of a span are computed before the span is drawn so that one can interpolate intensities. The main disadvantage is the large amount of processing required maintaining and sorting various lists. The main advantage to seed fill algorithms is that they can fill arbitrary planar contours, not just those bounded by polygonal curves. The main disadvantages are that some pixels are visited many times and one requires an initial interior point. The latter is not a problem in interactive situations but would be in a fully automated one. One would then have to invoke another algorithm to find such a point. See [AckW81] for some conclusions based on performance tests. Basically, fill time tends to be dominated by the time required to set pixels making the ordered edge list algorithms the most attractive overall. [FisB85] compares various specific seed fill algorithms. An antialiased scan conversion algorithm is described in [Morr90].

2.9.2 Drawing Circles

Probably the most straightforward approach to generating points on a circle is to use a polar coordinate parameterization. If, for simplicity, we restrict the discussion to circles of radius r centered at the origin, then this map is given by the formula

$$\theta \to (r\cos\theta, r\sin\theta).$$

The only problem with this is that the sine and cosine functions are relatively complicated to evaluate. We want a speedier algorithm. One can use rational functions. For example, there is the following rational function parameterization of the right half of the unit circle

$$t \to \left(\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}\right), \quad -1 \le t \le 1.$$

These rational polynomials can be evaluated rather efficiently using a method of forward differences, but the problem now is that equally spaced t's do not give rise to equally spaced points on the circle.

The DDA approach that led to a good algorithm in the case of lines is also applicable to circles. By differentiating the equation

$$x^2 + y^2 = r^2$$

for a circle of radius r about the origin implicitly with respect to x, one sees that

$$\frac{dy}{dx} = -\frac{x}{y}$$

is the differential equation for that circle. This means that a circle-generating DDA can be written in the form

$$x_{n+1} = x_n + \varepsilon\, y_n$$
$$y_{n+1} = y_n - \varepsilon\, x_n.$$

A natural choice for ε is 2^{-n}, where $2^{n-1} \le r < 2^n$. Unfortunately, if one were to plot the points that are generated by these equations, one would find that they spiral outward. From a mathematical standpoint one should not have been surprised because the determinant of the matrix

$$\begin{pmatrix} 1 & -\varepsilon \\ \varepsilon & 1 \end{pmatrix}$$

for this linear transformation is $1 + \varepsilon^2$. An ad hoc way to correct this determinant problem is to make a slight change and note that the matrix

$$\begin{pmatrix} 1 & -\varepsilon \\ \varepsilon & 1-\varepsilon^2 \end{pmatrix}$$

has determinant equal to 1. The points that are generated by the corresponding transformation produce a much better result. The equations for this transformation are

$$x_{n+1} = x_n + \varepsilon\, y_n$$
$$y_{n+1} = (1-\varepsilon^2)y_n - \varepsilon\, x_n,$$

which reduces to

$$x_{n+1} = x_n + \varepsilon\, y_n$$
$$y_{n+1} = y_n - \varepsilon\, x_{n+1}.$$

Notice how the equation for the (n + 1)st value for y now also involves the (n + 1)st value of x. The coordinates of the new point have to be computed sequentially, first the x value, then the y value. Before we could compute them in parallel. Furthermore, what we have here is a simple example of an error-correcting term. All good methods for numerical solutions to differential equations have this, something that was alluded to in Section 2.5.1.

Returning to our problem of generating points on a circle, our new system of equations produces points that no longer spiral out. However, having determinant equal to 1 is only a necessary requirement for a transformation to preserve distance. It is not a sufficient one. In fact, the points generated by our new transformation form a slight ellipse.

To get a better circle-generating algorithm we start over from scratch with a new approach. Assume that the radius r is an integer and define the "error" function E by

$$E = r^2 - x^2 - y^2.$$

This function measures how close the point (x,y) is to lying on the circle of radius r. As we generate points on the circle we obviously want to minimize this error. Let us restrict ourselves to the octant of the circle in the first quadrant, which starts at (0,r) and ends at $(r/\sqrt{2}, r/\sqrt{2})$. Note that in this octant as we move from one point to the next the x-coordinate will always increase by 1 and the y-coordinate will either stay the same or decrease by 1. Any other choice would not minimize E. The two cases are illustrated in Figure 2.19. We shall call the two possible moves an **R-move** or a **D-move**, respectively.

As we move from point to point, we choose that new point which minimizes E. However, we can save computation time by computing the new E incrementally. To see this, suppose that we are at (x,y). Then the current E, E_{cur}, is given by

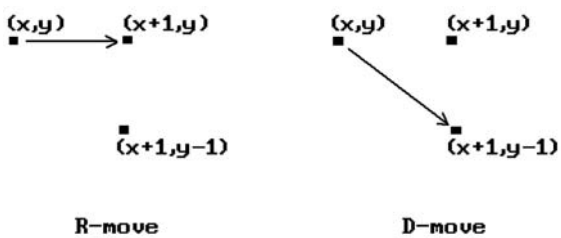


Figure 2.19. Moves in circle-generating algorithm.

$$E_{cur} = r^2 - x^2 - y^2.$$

After an R-move the new E, call it E_R, is

$$\begin{aligned} E_R &= r^2 - (x+1)^2 - y^2 \\ &= E_{cur} - (2x+1). \end{aligned}$$

After a D-move the new E, call it E_D, is

$$\begin{aligned} E_D &= r^2 - (x+1)^2 - (y-1)^2 \\ &= E_{cur} - (2x+1) + (2y-1). \end{aligned}$$

One algorithm for drawing a circle is to choose that move which makes our new error, either E_R or E_D, have the opposite sign of our current one. The idea is that if we find ourselves outside the circle we should move as quickly as possible back into the circle and vice versa. This leads to Algorithm 2.9.2.1, the Bresenham circle-drawing algorithm.

The only problem with Algorithm 2.9.2.1 is that we were using a heuristic that does not always minimize the real error E (Exercise 2.9.2.1). To get a better algorithm, we have to make that our goal. Choosing the move that minimizes the error can be done by testing the sign of $|E_D| - |E_R|$. To gain efficiency we want to avoid having to compute absolute values of numbers. Consider the possible outcomes shown in following table:

E_D	E_R	$\lvert E_D\rvert - \lvert E_R\rvert$
+	+	$E_D - E_R = 2y - 1$, always positive
+	−	$E_D + E_R$
−	+	this case never happens
−	−	$-E_D + E_R = -(2y - 1)$, always negative

```
x := 0;  y := r;  E := 0;
while x ≤ y do
    begin
        if E < 0 then
            begin
                E := E + y + y − 1;
                y := y − 1;
            end;
        E := E − x − x − 1;
        x := x + 1;
        Draw (x,y);
    end;
```

Algorithm 2.9.2.1. The Bresenham circle-drawing algorithm (one octant).

This table shows that the sign of $|E_D| - |E_R|$ always agrees with the sign of the auxiliary variable

$$G = E_D + E_R.$$

Furthermore, G can also be computed incrementally. Let G_R and G_D denote the values of G after an R-move or D-move, respectively. If G_{cur} is the current G value, then

$$G_R = G_{cur} - 4x - 6 \tag{2.4}$$

and

$$G_D = G_{cur} - 4x + 4y - 10. \tag{2.5}$$

It is easy to derive formulas (2.4) and (2.5). We prove formula (2.4) for G_R in case we move right. Recall that

$$\begin{aligned} E_R &= E_{cur} - (2x+1), \\ E_D &= E_{cur} - (2x+1) + (2y-1), \text{ and} \\ G_{cur} &= E_D + E_R = 2E_{cur} - (4x+2) + (2y-1). \end{aligned}$$

On an R-move,

$$\begin{aligned} E_{new} &= E_{cur} - (2x+1), \\ \text{new}E_R &= E_{new} - [2(x+1)+1], \\ \text{new}E_D &= E_{new} - [2(x+1)+1] + (2y-1), \text{ and} \\ G_R &= \text{new}E_D + \text{new}E_R \\ &= 2E_{new} - (4x+2) - 2 + (2y-1) \\ &= 2E_{cur} - 2(2x+1) - (4x+2) - 2 + (2y-1) \\ &= G_{cur} - 4x - 6. \end{aligned}$$

The other cases are proved in a similar fashion.

Finally, going one step further, the increments to G_R and G_D themselves can be computed incrementally, producing an improved Algorithm 2.9.2.2. It can be shown that the algorithm produces a best-fit curve for the circle when either the radius r or its square is an integer, but that may not be the case if one tries the same approach when r^2 is not an integer.

See [Blin87] for a more complete overview of circle drawing algorithms. For a version of the midpoint line-drawing algorithm that works for circles see [VanN85].

2.9.3 Drawing Ellipses and Other Conics

The equation for the standard ellipse centered at the origin is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, \tag{2.6}$$

```
x := 0;
y := r;
g := 2*r − 3;
dgr := − 6;  dgd := 4*r − 10;
while x ≤ y do
    begin
        if g < 0
            then                        { go diagonal }
                begin
                    g   := g + dgd;
                    dgd := dgd − 8;  { −4 (x+1) + 4 (y−1) − 10 = −4x + 4y − 10 − 8 }
                    y   := y − 1;
                end
            else                        { go right }
                begin
                    g   := g + dgr;
                    dgd := dgd − 4;  { y stays the same, x increases by 1 }
                end;

        dgr := dgr − 4;      { x always gets incremented: −4 (x+1) − 6 = −4x − 6 − 4 }
        x   := x + 1;
        Draw (x,y);
    end;
```

Algorithm 2.9.2.2. An improved Bresenham circle-drawing algorithm (one octant).

and we can generate points on it using the standard parameterization

$$\theta \to (a \cos \theta, b \sin \theta).$$

Differentiating equation (2.6) implicitly gives the differential equation

$$\frac{dy}{dx} = -\frac{x}{y}\frac{b^2}{a^2}.$$

This leads to a DDA approach similar to the case of a circle.

Because ellipses are an important class of curves, many algorithms exist to draw them. The Bresenham approach to drawing a circle can be extended to ellipses, but that algorithm will no longer guarantee a minimum linear error between the ellipse and the chosen pixels. See [Mcil92] for a correct Bresenham-type algorithm. Alternatively, a version of the midpoint algorithm described in Section 2.5.3 produces an efficient and more accurate algorithm. For more details and other references see [VanN85] and [Roge98].

Special algorithms for generating parabolas and hyperbolas are also known. We shall not describe any here but instead jump directly to the case of a general conic. Every conic can be defined implicitly by an equation of the form

$$ax^2 + bxy + cy^2 + dx + ey + f = 0.$$

Given a starting pixel **on** the conic, one can determine which adjacent pixel to pick next similar to what we did in the case of circles by looking at an error function. Each possible move will have an error associated to it and we simply choose the move with the least error. It is easy to show that the error functions have the same form as the equation of the conic and that they can be computed incrementally. See [Blin88a] and [Chan88] for more details.

Conics are a special case of implicitly defined curves and general algorithms for generating such curves will be presented in Section 14.5.1.

2.10 Bit Map Graphics

There are lots of situations in graphics where one needs to map blocks of bits from one location to another. Today's graphical user interfaces present a user with many "windows" that pop on and off the screen. Animation techniques usually achieve their motion effect by mapping a saved block of pixels to the current location of the moving figure (thereby erasing it and restoring the background) and then mapping another block containing the figure to its new location. This section looks at some rectangular bit map basics. The discussion is specific for the IBM PC, although most of it is generic. For a more extensive discussion see [Miel91], [FVFH90], and [DeFL87].

First of all, what does it take to define a bit map? Rectangles are specified by the upper left and lower right corner:

```
rectangle = record
  integer x0, y0,  { upper left corner }
          x1, y1;  { lower right corner }
  end;
```

A bit map specifies a rectangle in a possibly larger rectangle:

```
bitMap = record
  pointer  base;   { start address in memory }
  integer  width;  { width in number of words }
  rectangle rect;
end;
```

Bit maps are stored in row major form in memory. The width field refers to a possible larger bitmap that contains this rectangle. For example, the rectangle may be properly contained in a frame buffer whose pixels are stored in row major form. In that case, the pixels of the rectangle do not form a contiguous sequence and one needs the width of the bigger frame buffer to be able to access the pixels of the rectangle as one moves from one row to the next. See Figure 2.20. Note that each row of a rectangle is assumed to specify a contiguous chunk of bits in memory however.

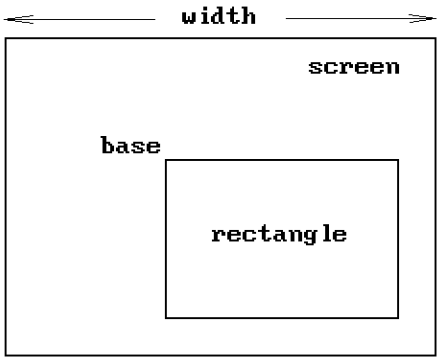


Figure 2.20. Specifying bit maps.

Several modes are allowed in bit map copying:

```
opMode = ( XORmode,ORmode,ANDmode,REPLACEmode );
```

One often allows a "texture" so that one can generate patterns.

```
texture = byte array [1..texlen];
```

The value of texlen depends on the graphics hardware. Each bit in the texture array indicates whether the corresponding bit in the source rectangle is to be copied.

Here are the basic parameters of a bitBlt (*bit block transfer*) procedure:

```
procedure BitBlt (bitMap      source;
                  integer     x0, y0;  { start point of source rectangle }
                  texture     tex;
                  bitMap      destination;
                  rectangle   rect;    { target rectangle }
                  opMode      mode);
```

Sometimes the source rectangle is specified instead of the target rectangle. In any case, both the source and target rectangle are the same size and may need to be clipped against their associated bit maps. The bit maps themselves may overlap so that the copy operation must be careful not to overwrite essential data. Here is an outline for the procedure:

```
Clip the source and target rectangle to their bit maps;
If either the width or height of the clipped rectangles is negative, then exit;
addr1 := address of start of clipped source rectangle;
addr2 := address of start of clipped target rectangle;
if  addr1 < addr2
  then
    begin
      Reset addr1 to address of end of clipped source rectangle;
      Reset addr2 to address of end of clipped target rectangle;
      Copy the rows of the source rectangle to the target in bottom to top order
    end
  else  Copy the rows of the source rectangle to the target in top to bottom order
```

Before each bit is copied, one checks the appropriate bit in the texture array. A "1" means that the bit is to be copied, a "0," that it is not. If a source bit sB is to be copied, then the target bit tB is replaced by sB **op** tB using the current copying mode **op**.

Although the abstract code for a bitBlt operation is straightforward, the tricky part is to implement it as efficiently as possible. For that reason, such operations are usually implemented in assembly language. A complicating factor is when addresses do not fall on word boundaries. Efficient coding is absolutely essential here and can speed up the speed of the procedure by orders of magnitude!

An important application of BitBlt procedures is managing a *cursor* on the screen. What this involves is mapping predefined bit maps to specified locations on the screen. A cursor corresponds to a small rectangle of bits. In Section 1.3 we already described a simple way to move a cursor is using **xor** mode along with the advantages and disadvantages of this method.

2.11 2D Animation

The object of this section is to describe a few simple aspects of two-dimensional animation. The general topic is much too large to do more than that here, especially since much of it is not really a topic in geometric modeling per se. On the other hand, many of the techniques used in two-dimensional computer animation belong in a discussion of raster graphics. This is certainly true of that part of animation which deals with showing moving objects and which lends itself to a lot of fun programming exercises. Keep in mind though that animation techniques have changed along with hardware. All we shall basically do here is describe a few interesting tricks that deal with Boolean operations on bits. These tricks were especially important in the early days of graphics where one had only a single frame buffer and not the multiple buffers that one finds in graphics systems today. We shall have a little to say about animation of three-dimensional objects in Sections 4.13 and 4.14.

Showing moving objects is accomplished by showing a sequence of still objects, where each individual picture shows the objects moved slightly from their previous position. Here are some simple methods by which one can perform such animation, starting with the most basic:

(1) Redraw the whole screen for each new image.

(2) If the objects consist of lines, then simply erase the current lines and redraw them at the new location.

(3) Erase/draw objects using block write commands.

(4) Use an approach similar to (3), except let each block have "trailing" blanks, so that each new block write erases and draws simultaneously. This reduces the number of block writes. An example of this is shown in Figure 2.21. If we wanted to show a ball moving from left to right, we could show the ball at the sequence of locations shown in (a). Rather than writing blocks of pixels the size of the ball, we could write a block shown in (b) which simultaneously erases the previous ball as it writes the new one.

(5) Use bit operations such as **xor**, **and**, or more general BitBlt procedures. We have already discussed in Section 1.3 how **xor** mode is useful in moving a cursor

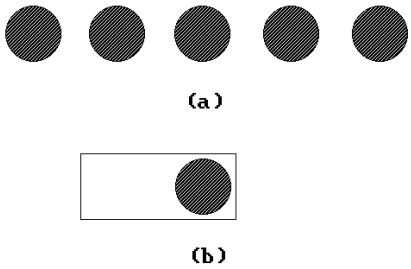


Figure 2.21. Animating with "trailing blank" blocks.

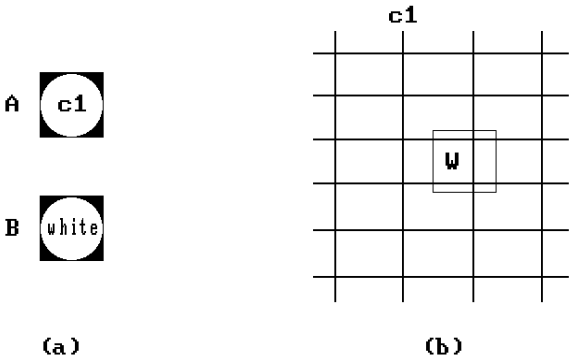


Figure 2.22. **or**/**xor** animation.

without disturbing the background. One can also use **xor** to do "rubber banding." For example, to drag a line anchored to a point around on the screen with the mouse, one would perform the following two instructions in a loop:

Erase the current line segment.
Draw the line segment from the fixed point to the new location of the mouse.

The advantage of **xor**ing is speed but its disadvantages are

The background can bleed through because if it is nonzero, then **xor** operation will add bits to the image being drawn.
One cannot **xor** a zero.

One can also use combinations of bit operations. Here is an example using the **or**/**xor** combination. See Figure 2.22. Suppose that one wants to move a ball around on a grid. Assume that the ball has color c1, the grid has color c2, and the background is black. Define two blocks **A** and **B** as follows: **A** contains a ball of color c1 on a black background and **B** contains a white ball on a black background. See Figure 2.22(a). If we want to write the ball to block **W** on the screen, we first save **W** and then replace **W** by (**B or W**) **xor A**. See Figure 2.22(b). To make this work, we need to assume that c1 and c2 are complimentary colors in the sense that

(a) c1 **or** c2 = white (or all 1s)
(b) c1 **and** c2 = black (or 0)

Actually, the only thing we really need for this technique to work in general is that the objects in the world have a different color from the object we are trying to move. The reason is that we have been talking about color **numbers** and the association between actual colors and their numbers is quite arbitrary and can be changed at will. This will become clear later when we talk about color lookup tables. One can play similar tricks with other bit operations such as **or**/**and**.

(6) Maintain several "pages" of information and "flip" between the pages. For example, to show a walking man, one could precompute several different walking positions of the man, and then cycle between them. Having predrawn images in memory may very well be faster than computing them dynamically.

The speed of objects depends on how far apart they are drawn in succession. A basic way to keep track of multiple moving figures is to cycle through them. One can keep track of them using arrays. If objects overlap but one knows their relative distances from the viewer one can prioritize them by distance and use a *painter's algorithm* (see Chapter 7) and draw the nearer objects last.

One big problem with multiple objects is collision detection. Bounding boxes (see Chapter 6) may be helpful in cutting down on the amount of computation if the objects are complicated. Sometimes one knows where collisions **can** occur and then one only has to check in those places. One potential problem is shown in Figure 2.23 where two objects crossed paths but no collision would have been detected since one only checks for that in their final position. To handle that one could keep track of the paths of objects and check their intersections.

Now, the methods above have lots of constraints and will only work in certain situations. In (2) and (3), each figure has to fit into a block with a **fixed** background in that block. The methods also tend to be slow. However, in the early days of the personal computer the graphics system did not give developers much to work with. There is another issue in the case of CRTs that was alluded to briefly in Section 1.3. Writing a block of memory to the frame buffer while the hardware is scanning it and displaying the picture will usually cause flicker in the image. This is caused by the fact that the scan is much faster than the memory writes and during one scan only part of the memory block may have been written. To get flicker-free images a programmer would have to be very careful about the timing of the writes, which would usually be done during the vertical retrace of the beam. Furthermore, everything would have to be done in assembly language to get maximum speed. Fortunately, by using APIs like

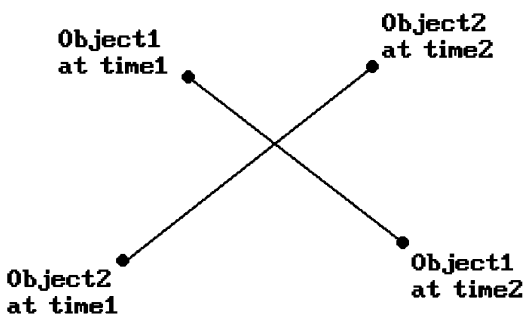


Figure 2.23. A collision-detection problem.

OpenGL and DirectX, most programmers no longer have to worry about such low-level issues because those were dealt with by the implementers of those APIs.

Next, we look at another aspect of frame buffers that is very helpful for animation. Today's frame buffers have hardware support for

(1) lookup tables
(2) panning, and
(3) zooming

Lookup Tables. Users and programmers can reference colors in different ways even though, at the hardware level, any representation eventually needs to be translated into RGB values. (We shall have more to say about color in Chapter 8.) One standard way to reference a color is by means of a number that really is an index into a table. This table is initialized by the operating system to certain default values. The size of the table depends on the number of colors that can be displayed simultaneously by the graphics system. For example, the number 0 is the standard representative for black. For a 256-color table, the standard number for white would be 255. The actual color that the hardware can associate to an index is quite arbitrary however and can be changed by a programmer. In that way, a relatively small table can access a large number of colors. For example, even if we only have a table of size 256 (8 bits), we would be able to access an actual 24 bit worth of colors (but only 256 at a time).

Panning. The frame buffer in the graphics hardware might actually be much larger than the number of pixels on the screen. By using "origin registers" that specify the location where the electron beam starts its scan in the buffer, the hardware makes panning easy. One can quickly scroll the image up or down, or left or right, simply by changing the values in those registers.

Zooming. The zoom feature allows one to display a portion of the image in magnified form. The magnification power is typically a power of 2. What happens is that a zoom factor of, say 2, would repeat each pixel and scan line twice.

The above-mentioned features allow three interesting forms of animation.

(1) **Animation using lookup tables:**

We explain this technique with an example. To show a red ball bouncing on a gray floor and a black background we could set up a picture as shown in Figure 2.24.

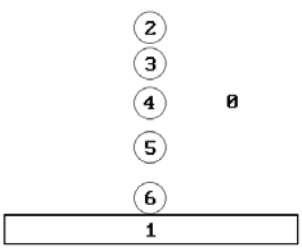


Figure 2.24. Bouncing ball using lookup table.

The numbers show the color numbers associated to the indicated regions. We start by associating the colors gray and red to 1 and 2, respectively, and black to all the other numbers. For the second frame we change 2 to black and 3 to red. For the third frame, we change 3 to black and 4 to red, and so on. In this way, making the ith ball the red ball as i changes from 2 to 6 and then from 6 back to 2, the ball will appear to bounce.

(2) **Animation using color cycling and bit plane extraction:**

Color cycling means that if we have a color table T with n entries, then we keep changing the colors of items via the instructions

$$T[2] := T[1]; T[3] := T[2]; \ldots T[i+1] := T[i]; \ldots T[n] := T[n-1]; T[1] := T[n];$$

If the colors in the image are designed appropriately, then these changes can create the illusion of motion.

Bit plane extraction relies on the fact that individual bits of a pixel can be thought of as corresponding to individual image bit planes. Frame buffers are then just a collection of k bit planes, where k is the number of bits in a pixel. See Figure 2.25. One trick we can now play is to create k different image frames with each frame using a subset of all colors. Then animation can be achieved by setting the lookup values for all values except the current frame value to the background color. Such updating of lookup values causes the picture to cycle through the frames. For example, assume that we have 3 bit pixels. We first set all color numbers **except** 4 to black, then we set all color numbers **except** 2 to black, and finally we set all color numbers **except** 1 to black. This will cycle through three single color images.

(3) **Animation using the pan and zoom features:**

One divides the frame buffer into k smaller areas. Typical values for k are 4 and 16. Next, one creates a reduced resolution image for a frame of animation in each area. The animation is produced by zooming each area appropri-

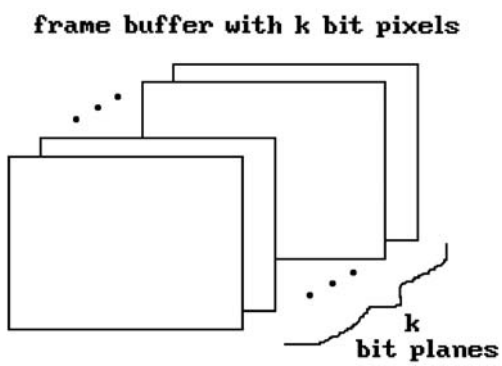


Figure 2.25. The bit planes of a frame buffer.

ately and then cycling through the areas by changing the origin registers appropriately.

Comparing the second and third methods, we see that the second has full resolution but allows a limited number of colors whereas the third has low resolution but offers full use of colors.

Some graphics systems supported small fixed-size rectangular pixel blocks that are maintained separately from the frame buffer but can be superimposed on it by the hardware. These are called *sprites* and were common in the hardware for video games. Using sprites it is very fast to move small images around on the screen without disturbing the background simply by changing some registers that specify the address in the frame buffer at which to show the sprite. Some simple video games consisted entirely of sprites moving over a fixed background. Collisions can be checked very easily. Sprites can also be implemented in software using bitBlt operations, although this will obviously not be as fast.

Finally, to get a smooth animation sequence, one needs to generate a minimum of 10–15 frames per second but this depends on the complexity of the scene and one probably wants more like 24–30 frames per second. Aside from the fact that writing large blocks of data takes time and would slow down any animation, there is an additional problem for CRTs that we have mentioned before. One has to be very careful about when and what one writes to the frame buffer if one wants to avoid flicker. It might be much better to create a complete image in an auxiliary buffer and then copy it to the frame buffer in one fell swoop. The only problem is that the copy operation would involve so much memory that it would not be fast enough and there would still be flicker. To avoid this copying most graphics systems on PCs now support what is called *double buffering*, that is, the auxiliary buffer one wants to write to is actually part of the graphics system (not part of main memory) and rather than copying it to the frame buffer, the hardware allows one to switch the scanning of the electron beam between it and the initial frame buffer. The changeover becomes almost instantaneous and the flicker will be gone. Of course, if it takes too long to compute each image, then the animation would be jerky.

Graphics systems nowadays have lots of memory on board and support all kinds of extra buffers. We shall have some more to say about this later in Chapter 9.

2.12 Exercises

Section 2.2

2.2.1 Find the various integers k so that there are k-neighbors in $\mathbf{Z}^4$.

2.2.2 Prove that every 8-component of a subset of $\mathbf{Z}^2$ is the union of 4-components.

Section 2.5.1

2.5.1.1 Determine the points generated by the simple and symmetric DDA for the line segment from (2,6) to (–1,1). Make a table and also plot these points (both the real **and** integer ones).

Section 2.9.1

2.9.2.1 Give an example showing that the first Bresenham circle-drawing algorithm (Algorithm 2.9.2.1) is **not** optimal but that the improved Bresenham algorithm (Algorithm 2.9.2.2) gives the correct results in this case.

2.13 Programming Projects

For some of the projects below you will first have to have a program, like GM, which lets a user define boundaries for arbitrary regions by dragging the cursor across the screen (either with a mouse or by using the arrow keys on the keyboard). Alternatively, deal only with polygonal regions that the user specifies by picking its vertices.

Section 2.4

2.4.1 Fill the region with a user selected color using the Fishkin algorithm.

Section 2.9.1

2.9.1.1 Implement the algorithm for scan converting polygons described in this section. More generally, implement this algorithm for multiple polygons that admit a back-to-front ordering.

Section 2.10

2.10.1 Implement a program showing a bouncing ball using the color lookup table technique as indicated in Figure 2.24.

2.10.2 A billiard ball game

The object of this project is to simulate the motion of a billiard ball and a cue stick. See Figure 2.26. Specifically, you should

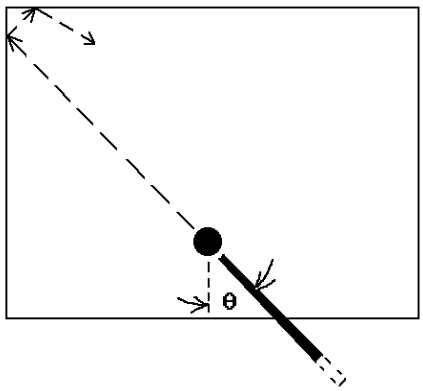


Figure 2.26. A billiard ball game.

(1) display a rectangular "table" with a single solid colored ball at some initial position

(2) show a cue stick (represented as a long, thin rectangular object) hitting the ball at various angles specified by the user

(3) show the movement of the ball after it is hit as it bounces from wall to wall

(4) have the balls slow down and finally come to a halt to make things look somewhat realistic

Experiment with different animation techniques on your own, such as double buffering. Hand in the one that you think gives the best results. *Be sure to explain* why it was chosen over other approaches.

Optionally, have one or more other balls on the table and

(5) if a moving ball hits another one, then it should come to a stop and the other one should move in the appropriate direction with the first one's velocity (if a ball hits $k > 1$ balls with velocity v, then they should start with velocity v/k)

Your program should have suitable menus and output explanations as to what is happening.

CHAPTER 3

Clipping

Prerequisites: Basic vector algebra

3.1 Introduction

Planar clipping algorithms rank as probably the second most important type of algorithm in computer graphics, following right behind line-drawing algorithms in importance. Mathematically, to *clip* one set against another means to find their intersection. In practice, one usually wants also to get this intersection in terms of some predefined data structure.

This chapter discusses some of the well-known clipping algorithms along with some newer and more efficient ones. The algorithms fall into two types: the *line-clipping* algorithms, which clip single line segments against rectangular or convex regions, and *polygon-clipping* algorithms, which clip whole polygons against other polygons. The following terminology is used:

Definition. The polygon being clipped is called the *subject* polygon and the polygon that one is clipping against is called the *clip* polygon.

The choice of algorithms to discuss was motivated by the following considerations:

(1) It is currently one of the best algorithms of its type.
(2) It is not the best algorithm but still used a lot.
(3) The algorithm was interesting for historical reasons and easy to describe.
(4) It involved the use of some interesting techniques, even though it itself is no longer a recommended method.

Below we list the algorithms described in this chapter and categorize them by considerations (1)–(4) above. We also state any assumption regarding their clip polygon and make some comments about them. Some of the algorithms will be discussed in great detail. Others are only described very briefly, especially if they fall under heading (3) or (4) above.

Line-clipping algorithms:

	Category	Clip Polygon	Comments
Cohen-Sutherland	(2)	rectangular	**The** classic line-clipping algorithm. Still popular because it is so easy to implement.
Cyrus-Beck	(4)	convex	
Liang-Barsky	(2)	rectangular	Faster than Cohen-Sutherland. Still popular. Easy to implement.
Nicholl-Lee-Nicholl	(1)	rectangular	Purely two-dimensional.

Polygon-clipping algorithms:

	Category	Clip Polygon	Comments
Sutherland-Hodgman	(3)	convex	
Weiler	(3), (4)	arbitrary	
Liang-Barsky	(4)	rectangular	
Maillot	(1)	rectangular	
Vatti	(1)	arbitrary	Fast, versatile, and can generate a trapezoidal decomposition of the intersection.
Greiner-Hormann	(1)	arbitrary	As general as Vatti. Simpler and potentially faster, but no trapezoidal decomposition.

Line-clipping algorithms fall into two types: those that use encoding of the endpoints of the segment (Cohen-Sutherland) and those that use a parameterization of the line determined by the segment (Cyrus-Beck, Liang-Barsky, and Nicholl-Lee-Nicholl). In Section 4.6 we discuss a hybrid of the two approaches that works well for the clipping needed in the graphics pipeline.

Frequently, one needs to clip more than one edge at a time, as is the case when one wants to clip one polygon against another. One could try to reduce this problem to a sequence of line-clipping problems, but that is not necessarily the most efficient way to do it, because, at the very least, there would be additional bookkeeping involved. The clipped figure may involve introducing some vertices that were not present in the original polygon. In Figure 3.1 we see that the corners **A** and **B**

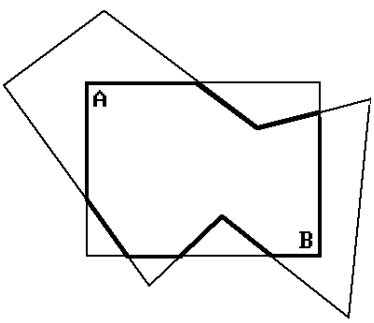


Figure 3.1. Turning points in polygon clipping.

of the window need to be added. These corners are called *turning points*. The term was introduced in [LiaB83] and refers to the point at the intersection of two clipping region edges that has to be added to preserve the connectivity of the original polygon. This is the reason that polygon clipping is treated separately from line clipping.

Polygon-clipping algorithms fall into roughly two categories: *turning-point-based algorithms* like the Liang-Barsky and Maillot algorithms, which rely on quickly being able to find turning points explicitly, and the rest. Turning-point-type algorithms scan the segments of the subject polygon and basically clip each against the **whole** window. The rest tend to find turning points implicitly, in the sense that one does not look for them directly but that they are generated "automatically" as the algorithm proceeds. The Sutherland-Hodgman algorithm treats the clip polygon as the intersection of halfplanes and clips the **whole** subject polygon against each of these halfplanes one at a time. The Weiler, Vatti, and Greiner-Hormann algorithms find the turning points from the clip polygon in the process of tracing out the bounding curves of the components of the polygon intersection, although they trace the boundaries in different ways.

The chapter ends with some comments on clipping text. Some additional comments on clipping when homogeneous coordinates are used can be found in the next chapter in Sections 4.6 and 4.10.

3.2 Line-Clipping Algorithms

3.2.1 Cohen-Sutherland Line Clipping

This section describes an algorithm that solves the following planar clipping problem:

> Given a segment $[\mathbf{P}_1, \mathbf{P}_2]$, clip it against a rectangular window and return the clipped segment $[\mathbf{Q}_1, \mathbf{Q}_2]$ (which may be empty if the original segment lies entirely outside the window).

The Cohen-Sutherland line-clipping algorithm is probably the most popular of such algorithms because of its simplicity. It starts out by encoding the nine regions into which the boundary lines of the window divide the whole plane with a 4-bit binary code. See Figure 3.2. If $\mathbf{P}$ is an arbitrary point, then let $c(\mathbf{P}) = x_3x_2x_1x_0$, where x_i is either 0 or 1, define this encoding. The bits x_i have the following meaning:

0110	0010	0011
0100	0000	0001
1100	1000	1001

Figure 3.2. Cohen-Sutherland point codes.

$x_0 = 1$ if and only if **P** lies strictly to the right of the right boundary line.
$x_1 = 1$ if and only if **P** lies strictly above the top boundary line.
$x_2 = 1$ if and only if **P** lies strictly to the left of the left boundary line.
$x_3 = 1$ if and only if **P** lies strictly below the bottom boundary line.

The algorithm now has three steps:

Step 1. Encode $\mathbf{P}_1$ and $\mathbf{P}_2$. Let $c_1 = c(\mathbf{P}_1)$ and $c_2 = c(\mathbf{P}_2)$.

Step 2. Check if the segment can be trivially rejected, that is, using the bitwise logical **or** and **and** operators, test whether

(a) c_1 **or** $c_2 = 0$, or
(b) c_1 **and** $c_2 \neq 0$.

In case (a), the segment is entirely contained in the window since both endpoints are and the window is convex. Return $\mathbf{Q}_1 = \mathbf{P}_1$ and $\mathbf{Q}_2 = \mathbf{P}_2$.
In case (b), the segment is entirely outside the window. This follows because the endpoints will then lie in the halfplane determined by a boundary line that is on the other side from the window and halfplanes are also convex. Return the empty segment.

Step 3. If the segment cannot be trivially rejected, then we must subdivide the segment. We clip it against an appropriate boundary line and then start over with Step 1 using the new segment. Do the following to accomplish this:

(a) First find the endpoint **P** that will determine the line to clip against.

If $c_1 = 0000$, then $\mathbf{P}_1$ does not have to be clipped and we let **P** be $\mathbf{P}_2$ and **Q** be $\mathbf{P}_1$.
If $c_1 \neq 0000$, then let **P** be $\mathbf{P}_1$ and **Q** be $\mathbf{P}_2$.

(b) The line to clip against is determined by the left-most nonzero bit in $c(\mathbf{P})$. For the example in Figure 3.3, $\mathbf{P} = \mathbf{P}_1$, $\mathbf{Q} = \mathbf{P}_2$, and the line to clip against is the left boundary line of the window. Let **A** be the intersection of the segment [**P**,**Q**] with this line.
(c) Repeat Steps 1–3 for the segment [**A**,**Q**].

The algorithm will eventually exit in Step 2.

With respect to the efficiency of the Cohen-Sutherland algorithm, note that the encoding is easy since it simply involves comparing a number to some constant (the

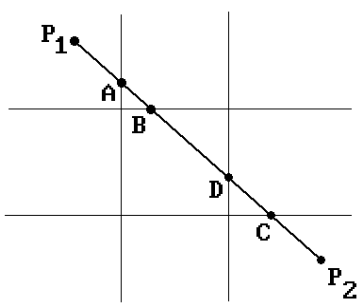


Figure 3.3. Cohen-Sutherland line-clipping example.

boundary lines of the window are assumed to be horizontal and vertical). Step 3 is where the real work might have to be done. We shall have to clip four times in the worst case. One such worst case is shown in Figure 3.3 where we end up having to clip successively against each one of the window boundary lines generating the intersection points **A**, **B**, **C**, and **D**.

Algorithm 3.2.1.1 is an implementation of the just-described algorithm. To be more efficient, all function calls should be replaced by inline code.

Finally, note that the encoding found in the Cohen-Sutherland line-clipping algorithm is driven by determining whether a point belongs to a halfplane. One can easily generalize this to the case where one is clipping a segment against an arbitrary convex set **X**. Assume that **X** is the intersection of halfplanes $\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_k$. The encoding of a point **P** is now a k-bit number $c(\mathbf{P}) = X_K X_{K-1} \ldots X_1$, where

$$X_i \text{ is 1 if } \mathbf{P} \text{ lies in } \mathbf{H}_i \text{ and 0 otherwise.}$$

Using this encoding one can define a clipping algorithm that consists of essentially the same steps as those in the Cohen-Sutherland algorithm. One can also extend this idea to higher dimensions and use it to clip segments against cubes. See Section 4.6.

3.2.2 Cyrus-Beck Line Clipping

The Cyrus-Beck line-clipping algorithm ([CyrB78]) clips a segment **S** against an arbitrary convex polygon **X**. Let $\mathbf{S} = [\mathbf{P}_1, \mathbf{P}_2]$ and $\mathbf{X} = \mathbf{Q}_1\mathbf{Q}_2 \ldots \mathbf{Q}_k$. Since **X** is convex, it is the intersection of halfplanes determined by its edges. More precisely, for each segment $[\mathbf{Q}_i, \mathbf{Q}_{i+1}]$, i = 1,2, . . . ,k, ($\mathbf{Q}_{k+1}$ denotes the point $\mathbf{Q}_1$) we can choose a normal vector $\mathbf{N}_i$, so that **X** can be expressed in the form

$$\mathbf{X} = \bigcap_{i=1}^{k} \mathbf{H}_i,$$

where $\mathbf{H}_i$ is the halfplane

$$\mathbf{H}_i = \{\mathbf{Q} \mid \mathbf{N}_i \bullet (\mathbf{Q} - \mathbf{Q}_i) \geq 0\}.$$

With this choice, the normals will point "into" the polygon. It follows that

$$\mathbf{S} \cap \mathbf{X} = \bigcap_{i=1}^{k} (\mathbf{S} \cap \mathbf{H}_i).$$

In other words, we can clip the segment **S** against **X** by successively clipping it against the halfplanes $\mathbf{H}_i$. This is the first basic idea behind the Cyrus-Beck clipping algorithm. The second, is to represent the line **L** determined by the segment **S** parametrically in the form $\mathbf{P}_1 + t\mathbf{P}_1\mathbf{P}_2$ and to do the clipping with respect to the parameter t.

```
{ Constants }
RIGHTBDRY    = 1;
TOPBDRY      = 2;
LEFTBDRY     = 4;
BOTTOMBDRY = 8;

boolean function CS_Clip (ref real x0, y0, x1, y1;  real xmin, ymin, xmax, ymax)
{ This function clips the segment from (x0, y0) to (x1, y1) against the window
  [xmin, xmax]×[ymin, ymax].  It returns false if the segment is entirely outside the
  window and true otherwise.  In the latter case the variables x0, y0, x1, and y1 will
  also have been modified to specify the final clipped segment. }
begin
    byte c0, c1, c;
    real x, y;

    { First encode the points }
    c0 := RegionCode (x0,y0);
    c1 := RegionCode (x1,y1);

    { Next the main loop }
    while  (c0 or c1) ≠ 0  do
        if  (c0 and c1) ≠ 0
            then    return (false);
            else
                begin
                    { Choose the first point not in the window }
                    c := c0;
                    if  c = 0  then  c := c1;

                    { Now clip against line corresponding to first nonzero bit }
                    if  (LEFTBDRY and c) ≠ 0
                        then
                            begin                    { Clip against left bdry }
                                x := xmin;
                                y := y0 + (y1 − y0)*(xmin − x0)/(x1 − x0);
                            end
                        else  if  (RIGHTBDRY and c) ≠ 0
                            then
                                begin                { Clip against right bdry }
                                    x := xmax;
                                    y := y0 + (y1 − y0)*(xmax − x0)/(x1 − x0);
                                end
                            else  if  (BOTTOMBDRY and c) ≠ 0
                                then
```

Algorithm 3.2.1.1. The Cohen-Sutherland line-clipping algorithm.

```
                                begin        { Clip against bottom bdry }
                                    x := x0 + (x1 − x0)*(ymin − y0)/(y1 − y0);
                                    y := ymin;
                                end
                            else if (TOPBDRY and c) ≠ 0
                                then
                                    begin   { Clip against top bdry }
                                      x := x0 + (x1 − x0)*(ymax − y0)/(y1 − y0);
                                      y := ymax;
                                    end;

                    { Next update the clipped endpoint and its code }
                    if c = c0
                        then
                            begin
                                x0 := x;  y0 := y;
                                c0 := RegionCode (x0,y0);
                            end
                        else
                            begin
                                x1 := x;  y1 := y;
                                c1 := RegionCode (x1,y1);
                            end
                end;  { while }

    return (true);
end;

byte function RegionCode (real x,y);
{ Return the 4-bit code for the point (x,y) }
begin
    byte c;

    c := 0;
    if x < xmin
        then c := c + LEFTBDRY
        else if x > xmax
                then c := c + RIGHTBDRY;
    if y < ymin
        then c := c + BOTTOMBDRY
        else if y > ymax
            then c := c + TOPBDRY;
    return (c);
end;        { RegionCode }
```

Algorithm 3.2.1.1. *Continued*

Let $\mathbf{L}_i$ be the line determined by the segment $[\mathbf{Q}_i,\mathbf{Q}_{i+1}]$. Define intervals $\mathbf{I}_i = [a_i,b_i]$ as follows:

Case 1: $\mathbf{L}$ is parallel to $\mathbf{L}_i$.

(a) If $\mathbf{L}$ lies entirely in $\mathbf{H}_i$, then let $\mathbf{I}_i = (-\infty, +\infty)$.
(b) If $\mathbf{L}$ lies entirely outside of $\mathbf{H}_i$, then let $\mathbf{I}_i = \phi$.

Case 2: $\mathbf{L}$ is not parallel to $\mathbf{L}_i$.

In this case $\mathbf{L}$ will intersect $\mathbf{L}_i$ in some point $\mathbf{P} = \mathbf{P}_1 + t_i\mathbf{P}_1\mathbf{P}_2$. We distinguish between the case where the line $\mathbf{L}$ "enters" the halfplane $\mathbf{H}_i$ and where the line "exits" $\mathbf{H}_i$.

(a) (Line enters) If $\mathbf{P}_1\mathbf{P}_2 \bullet \mathbf{N}_i \geq 0$, then let $\mathbf{I}_i = [t_i, +\infty)$.
(b) (Line exits) If $\mathbf{P}_1\mathbf{P}_2 \bullet \mathbf{N}_i < 0$, then let $\mathbf{I}_i = (-\infty, t_i]$.

See Figure 3.4, where a segment $\mathbf{S} = [\mathbf{P}_1,\mathbf{P}_2]$ is being clipped against a triangle $\mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3$. Note that finding the intersection point $\mathbf{P}$ in Case 2 is easy. All we have to do is solve the equation

$$\mathbf{N}_i \bullet (\mathbf{P}_1 + t\,\mathbf{P}_1\mathbf{P}_2 - \mathbf{Q}_i) = 0$$

for t.

Now let $\mathbf{I}_0 = [a_0,b_0] = [0,1]$. The interval $\mathbf{I}_0$ is the set of parameters of points which lie is $\mathbf{S}$. It is easy to see that the interval

$$\begin{aligned} I &= \bigcap_{i=0}^{k} I_i \\ &= \left[\max_{0\le i\le k} a_i, \min_{0\le i\le k} b_i\right] \\ &= [a,b] \end{aligned}$$

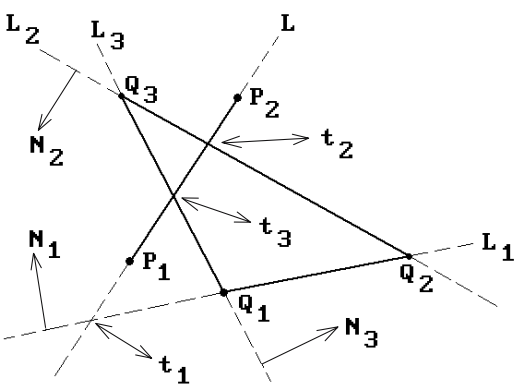


Figure 3.4. Cyrus-Beck line clipping.

is the set of parameters for the points in $\mathbf{S} \cap \mathbf{X}$. In other words, if $\mathbf{I}$ is not empty, then

$$\mathbf{S} \cap \mathbf{X} = [\mathbf{P}_1 + a\mathbf{P}_1\mathbf{P}_2, \mathbf{P}_1 + b\mathbf{P}_1\mathbf{P}_2].$$

We shall explain this process with the example in Figure 3.4. In this example,

$$\mathbf{I} = [0,1] \cap [t_1, +\infty) \cap (-\infty, t_2] \cap [t_3, +\infty) = [t_3, t_2],$$

which clearly gives the right answer.

3.2.3 Liang-Barsky Line Clipping

The Liang-Barsky line-clipping algorithm ([LiaB84]) optimizes the Cyrus-Beck line-clipping algorithm in the case where we are clipping against a rectangle. It starts by treating a segment as a parameterized set. Let $\mathbf{P}_1 = (x_1,y_1)$ and $\mathbf{P}_2 = (x_2,y_2)$. A typical point $\mathbf{P} = (x,y)$ on the oriented line $\mathbf{L}$ determined by $\mathbf{P}_1$ and $\mathbf{P}_2$ then has the form $\mathbf{P}_1 + t\mathbf{P}_1\mathbf{P}_2$. See Figure 3.5. If we let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$, then

$$\begin{aligned} x &= x_1 + \Delta x\, t \\ y &= y_1 + \Delta y\, t. \end{aligned}$$

If the window $\mathbf{W}$ we are clipping against is the rectangle $[xmin,xmax] \times [ymin,ymax]$, then $\mathbf{P}$ belongs to $\mathbf{W}$ if and only if

$$\begin{aligned} xmin &\le x_1 + \Delta x\, t \le xmax \\ ymin &\le y_1 + \Delta y\, t \le ymax \end{aligned}$$

that is,

$$\begin{aligned} -\Delta x\, t &\le x_1 - xmin \\ \Delta x\, t &\le xmax - x_1 \\ -\Delta y\, t &\le y_1 - ymin \\ \Delta y\, t &\le ymax - y_1. \end{aligned}$$

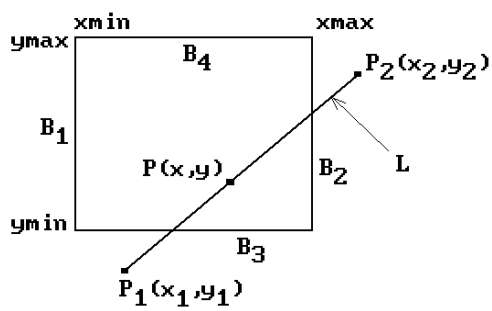


Figure 3.5. Liang-Barsky line clipping.

To simplify the notation, we introduce variables c_k and q_k and rewrite these equations as

$$c_1 t \le q_1$$
$$c_2 t \le q_2$$
$$c_3 t \le q_3$$
$$c_4 t \le q_4.$$

Set $t_k = q_k/c_k$ whenever $c_k \neq 0$. Let $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{B}_3$, and $\mathbf{B}_4$ denote the left, right, bottom, and top boundary lines of the window, respectively. With this notation we can make the following observations:

(1) If $c_k > 0$, then **L** goes from the "inside" to the "outside" of the boundary line $\mathbf{B}_k$ as t increases and we shall call t_k an *exit* value.
(2) If $c_k < 0$, then **L** goes from the "outside" to the "inside" of the boundary line $\mathbf{B}_k$ as t increases and we shall call t_k an *entry* value.
(3) If $c_k = 0$, the **L** is parallel to $\mathbf{B}_k$, which is outside the window if $q_k < 0$.

The clipping algorithm now proceeds by analyzing the three quantities q_k, c_k, and t_k. Algorithm 3.2.3.1 is a high-level version of the Liang-Barsky algorithm. Algorithm 3.2.3.2 gives the code for the actual Liang-Barsky algorithm.

3.2.3.1 Example. Consider the segment $[\mathbf{P}_1,\mathbf{P}_2]$ in Figure 3.6. We see that c_1, $c_4 < 0$ and c_2, $c_3 > 0$. In other words, t_1 and t_4 are entry values and t_2 and t_3 are exit values. The picture bears this out. One can also easily see that $t_1 < 0 < t_2 < t_4 < 1 < t_3$. Therefore, there is an entry value (t_4) that is larger than an exit value (t_2). Using the algorithm we conclude that the segment lies entirely outside the window, which is correct.

Reject the segment as soon as

an entry value is larger than 1 or
an exit value is less than 0 or
an entry value is larger than an exit value

Otherwise, the segment meets the window. We need to compute an intersection only if $t0 > 0$ and $t1 < 1$, where

$$t0 = \max(0, \max\{\text{ entry values } t_k \}), \text{ and}$$
$$t1 = \min(1, \min\{\text{ exit values } t_k \}).$$

(The case where $t0 = 0$ or $t1 = 1$ means that we should use the endpoint, that is, no clipping is necessary.)

Algorithm 3.2.3.1. High-level Liang-Barsky line-clipping algorithm.

```
boolean function LB_Clip (ref real  x0, y0, x1, y1;  real xmin, ymin, xmax, ymax)
{ This function clips the segment from (x0, y0) to (x1, y1) against the window
[xmin, xmax]×[ymin, ymax].  It returns false if the segment is entirely outside the
window and true otherwise.  In the latter case the variables x0, y0, x1, and y1 will
also have been modified to specify the final clipped segment. }
begin
    real      t0, t1, dx, dy;
    boolean more;

    t0 := 0;  t1 := 1;  dx := x1 – x0;

    Findt (–dx,x0 – xmin,t0,t1,more);          { left bdry }
    if  more  then
        begin
            Findt (dx,xmax – x0,t0,t1,more);       { right bdry }
            if  more  then
                begin
                    dy := y1  –  y0;
                    Findt (–dy,y0 – ymin,t0,t1,more);  { bottom bdry }
                    if  more  then
                        begin
                            Findt (dy,ymax – y0,t0,t1,more);  { top bdry }
                            if  more  then
                                begin                  { clip the line }
                                    if  t1 < 1  then
                                        begin          { calculate exit point }
                                            x1 := x0 + t1*dx;
                                            y1 := y0 + t1*dy;
                                        end;
                                    if  t0 > 0  then
                                        begin          { calculate entry point }
                                            x0 := x0 + t0*dx;
                                            y0 := y0 + t0*dy;
                                        end;
                                end
                        end
                end
        end;
    return (more);
end;

procedure Findt (real denom, num; ref real t0, t1; ref boolean more)
begin
    real r;

    more := true;
```

Algorithm 3.2.3.2. The Liang-Barsky line-clipping algorithm.

```
    if denom < 0
        then
            begin                    { line from outside to inside }
                r := num/denom;
                if r > t1
                    then more := false
                    else if r > t0 then t0 := r;
            end
        else if denom > 0
            then                     { line from inside to outside }
                begin
                    r := num/denom;
                    if r < t0
                        then more := false
                        else if r < t1 then t1 := r;
                end
            else if num < 0          { line parallel to boundary }
                then more := false;
end; { Findt }
```

Algorithm 3.2.3.2. *Continued*

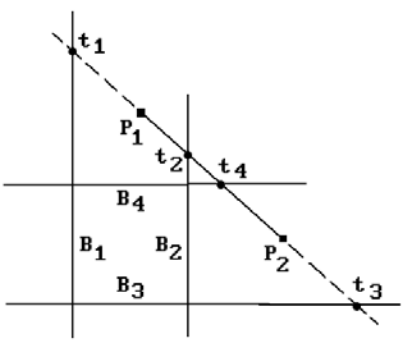


Figure 3.6. Liang-Barsky line-clipping example.

3.2.3.2 Example. Consider the segment [$\mathbf{P}_1$,$\mathbf{P}_2$] in Figure 3.7. In this example, c_1, $c_3 < 0$ and c_2, $c_4 > 0$, so that t_1 and t_3 are entry values and t_2 and t_4 are exit values. Furthermore, $t_1 < 0 < t_3 < 1 < t_2 < t_4$. This time we cannot reject the segment and must compute $t0 = \max(0, c_1, c_3) = c_3$ and $t1 = \min(1, c_2, c_4) = 1$. The algorithm tells us that we must clip the segment at the $\mathbf{P}_1$ end to get $\mathbf{Q}$ but do not need to clip at the $\mathbf{P}_2$ end. Again this is clearly what had to be done.

In conclusion, the advantage of the Liang-Barsky algorithm over the Cohen-Sutherland algorithm is that it involves less arithmetic and is therefore faster. It needs only two subtractions to get q_k and c_k and then one division.

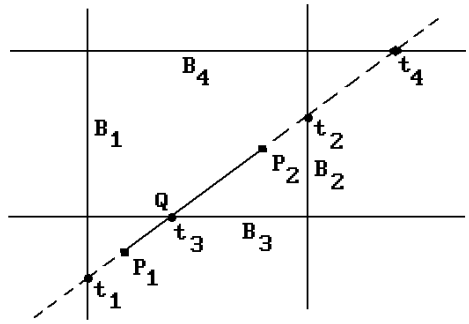


Figure 3.7. Liang-Barsky line-clipping example.

3.2.4 Nicholl-Lee-Nicholl Line Clipping

One of the problems common to both the Cohen-Sutherland and the Liang-Barsky algorithm is that more intersections are computed than necessary. For example, consider Figure 3.6 again where we are clipping line segment [$\mathbf{P}_1$,$\mathbf{P}_2$] against the window. The Cohen-Sutherland algorithm will compute the intersection of the segment with the top boundary at t_4 even though the segment is later rejected. The Liang-Barsky algorithm will actually compute **all** the parameter values corresponding to the intersection of the line with the window. Avoiding many of these wasted computations is what the Nicholl-Lee-Nicholl line-clipping algorithm ([NiLN87]) is all about. These authors also make a detailed analysis of the deficiencies of the Cohen-Sutherland and Liang-Barsky algorithms. Their final algorithm is much faster than either of these. It is not really much more complicated conceptually, but involves many cases. We describe one basic case below.

Assume that we want to clip a segment [$\mathbf{P}_1$,$\mathbf{P}_2$] against a window. The determination of the exact edges, if any, that one needs to intersect, reduces, using symmetry, to an analysis of the three possible positions of $\mathbf{P}_1$ shown in Figure 3.8. The cases are

(1) $\mathbf{P}_1$ is in the window (Figure 3.8(a)),
(2) $\mathbf{P}_1$ is in a "corner region" (Figure 3.8(b)), or
(3) $\mathbf{P}_1$ is in an "edge region" (Figure 3.8(c)).

For each of these cases one determines the regions with the property that no matter where in the region the second point $\mathbf{P}_2$ is, the segment will have to be intersected with the **same** boundaries of the window. These regions are also indicated in Figure 3.8. As one can see, these regions are determined by drawing the rays from $\mathbf{P}_1$ through the four corners of the window. The following abbreviations were used:

T – ray intersects top boundary
L – ray intersects left boundary
B – ray intersects bottom boundary
R – ray intersects right boundary

LT – ray intersects left and top boundary
LR – ray intersects left and right boundary
LB – ray intersects left and bottom boundary
TR – ray intersects top and right boundary
TB – ray intersects top and bottom boundary

For example, suppose that the segment $[\mathbf{P}_1,\mathbf{P}_2]$ is as shown in Figure 3.8(c). Here are the computations one has to perform. Let $\mathbf{P}_i = (x_i,y_i)$ and let $\mathbf{C} = (\text{xmax},\text{ymin})$ be the corner of the window also indicated in Figure 3.8(c). After checking that $y_2 < \text{ymin}$, we must determine whether the vector $\mathbf{P}_1\mathbf{P}_2$ is above or below the vector $\mathbf{P}_1\mathbf{C}$. This reduces to determining whether the ordered basis $(\mathbf{P}_1\mathbf{C},\mathbf{P}_1\mathbf{P}_2)$ determines the standard orientation of the plane or not. Since

$$\det\begin{pmatrix}\mathbf{P}_1\mathbf{C}\\ \mathbf{P}_1\mathbf{P}_2\end{pmatrix} = (\text{xmax}-x_1)(y_2-y_1)-(\text{ymin}-y_1)(x_2-x_1)<0,$$

$\mathbf{P}_1\mathbf{P}_2$ is below $\mathbf{P}_1\mathbf{C}$. We now know that we will have to compute the intersection of $[\mathbf{P}_1,\mathbf{P}_2]$ with both the left and bottom boundary of the window.

Algorithm 3.2.4.1 is an abstract program for the Nicholl-Lee-Nicholl algorithm in the edge region case ($\mathbf{P}_1$ in the region shown in Figure 3.8(c)). We assume a window $[\text{xmin},\text{xmax}] \times [\text{ymin},\text{ymax}]$.

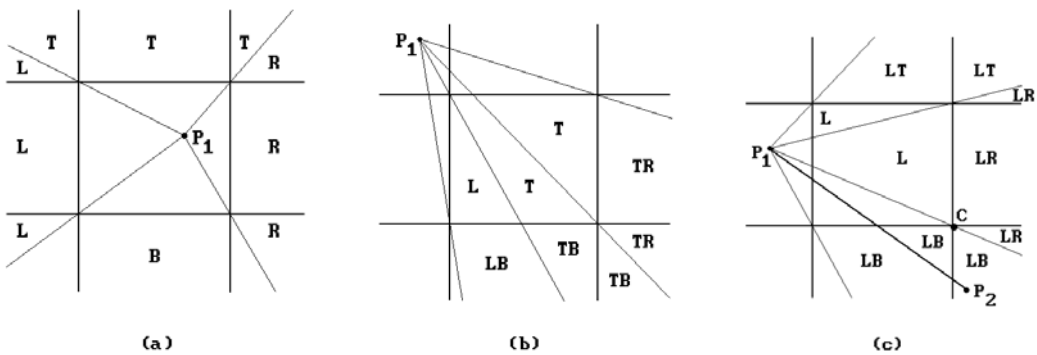


Figure 3.8. Nicholl-Lee-Nicholl line clipping.

```
procedure LeftEdgeRegionCase (ref real x1, y1, x2, y2; ref boolean visible)
begin
    real dx, dy;

    if  x2 < xmin
        then  visible := false
        else  if  y2 < ymin
            then  LeftBottom (xmin,ymin,xmax,ymax,x1,y1,x2,y2,visible)
            else  if  y2 > ymax
                then
                    begin
```

Algorithm 3.2.4.1. The edge region case of the Nicholl-Lee-Nicholl line-clipping algorithm.

```
                    { Use symmetry to reduce to LeftBottom case }
                    y1 := –y1;   y2 := –y2;   { reflect about x-axis }
                    LeftBottom (xmin,–ymax,xmax, –ymin,x1,y1,x2,y2,visible);
                    y1 := –y1;   y2 := –y2;   { reflect back }
                end
            else
                begin
                    dx := x2 – x1;   dy := y2 – y1;
                    if  x2 > xmax  then
                        begin
                            y2 := y1 + dy*(xmax – x1)/dx;    x2 := xmax;
                        end;
                    y1 := y1 + dy*(xmin – x1)/dx;    x1 := xmin;
                    visible := true;
                end
end;

procedure LeftBottom (      real  xmin, ymin, xmax, ymax;
                         ref real  x1, y1, x2, y2; ref boolean  visible)
begin
    real dx, dy, a, b, c;

    dx := x2 – x1;              dy := y2 – y1;
    a   := (xmin – x1)*dy;   b   := (ymin – y1)*dx;
    if  b > a
        then  visible := false  { (x2,y2) is below ray from (x1,y1) to bottom left corner }
        else
            begin
                visible := true;
                if  x2 < xmax
                    then
                        begin   x2 := x1 + b/dy;    y2 := ymin;    end
                    else
                        begin
                            c := (xmax – x1)*dy;
                            if  b > c
                                then   { (x2,y2) is between rays from (x1,y1) to
                                         bottom left and right corner }
                                    begin   x2 := x1 + b/dy;    y2 := ymin;    end
                                else
                                    begin   y2 := y1 + c/dx;    x2 := xmax;   end
                        end;
            end;
    y1 := y1 + a/dx;   x1 := xmin;
end;
```

Algorithm 3.2.4.1. *Continued*

To deal with symmetry only rotations through 90, 180, and 270 degrees about the origin and reflections about the lines x = –y and the x-axis are needed. These operations are extremely simple and involve only negation and assignment. See [NiLN87] for further details.

This finishes our survey of line-clipping algorithms. Next, we turn our attention to polygon-clipping algorithms.

3.3 Polygon-Clipping Algorithms

3.3.1 Sutherland-Hodgman Polygon Clipping

One of the earliest polygon-clipping algorithms is the Sutherland-Hodgman algorithm ([SutH74]). It is based on clipping the **entire** subject polygon against an edge of the window (more precisely, the halfplane determined by that edge which contains the clip polygon), then clipping the new polygon against the next edge of the window, and so on, until the polygon has been clipped against all of the four edges. An important aspect of their algorithm is that one can avoid generating a lot of intermediate data.

Representing a polygon as a sequence of vertices $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n$, suppose that we want to clip against a single edge **e**. The algorithm considers the input vertices $\mathbf{P}_i$ one at a time and generates a new sequence $\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_m$. Each $\mathbf{P}_i$ generates 0, 1, or 2 of the $\mathbf{Q}_j$, depending on the position of the input vertices with respect to **e**. If we consider each input vertex **P**, except the first, to be the terminal vertex of an edge, namely the edge defined by **P** and the immediately preceding input vertex, call it **S**, then the **Q**'s generated by **P** depend on the relationship between the edge [**S**,**P**] and the line **L** determined by **e**. There are four possible cases. See Figure 3.9. The window side of the line is marked as "inside." The circled vertices are those that are output. Figure 3.10 shows an example of how the clipping works. Clipping the polygon with vertices labeled $\mathbf{P}_i$ against edge $\mathbf{e}_1$ produces the polygon with vertices $\mathbf{Q}_i$. Clipping the new polygon against edge $\mathbf{e}_2$ produces the polygon with vertices $\mathbf{R}_i$.

Note that we may end up with some bogus edges. For example, the edge $\mathbf{R}_5\mathbf{R}_6$ in Figure 3.10 is not a part of the mathematical intersection of the subject polygon with

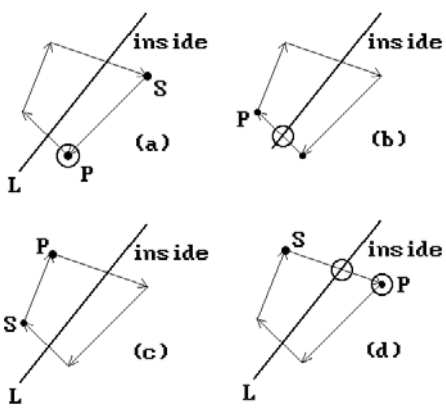


Figure 3.9. The four cases in Sutherland-Hodgman polygon clipping.

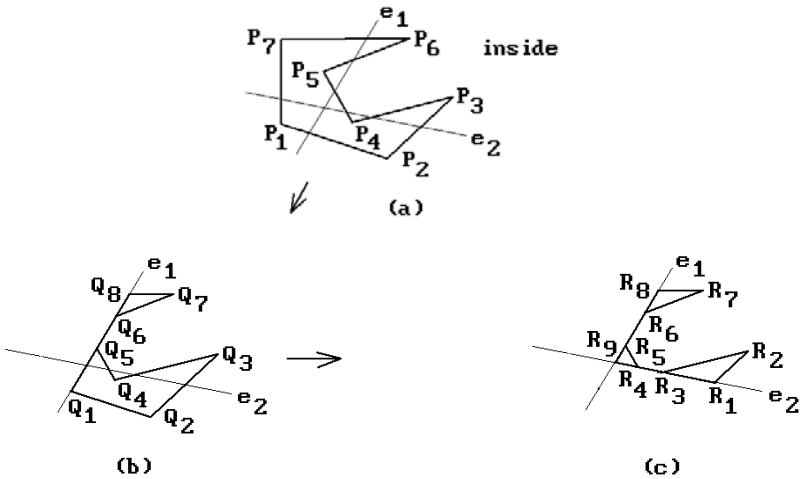


Figure 3.10. A Sutherland-Hodgman polygon-clipping example.

the clip polygon. Eliminating such edges from the final result would be a nontrivial effort, but normally they do not cause any problems. We run into this bogus edge problem with other clipping algorithms also.

An implementation of the Sutherland-Hodgman algorithm can be found in [PokG89].

3.3.2 Weiler Polygon Clipping

Another early polygon clipping algorithm was developed in the context of the visible surface determination algorithm in [WeiA77]. Weiler and Atherton needed a new algorithm because the Sutherland-Hodgman algorithm would have created too many auxiliary polygons. An improved version of the algorithm can be found in [Weil80]. Here is a very brief description of the algorithm:

> The boundaries of polygons are assumed to be oriented so that the inside of the polygon is always to the right as one traverses the boundary. Note that intersections of the subject and clip polygon, if any, occur in pairs: one where the subject *enters* the inside of the clip polygon and one where it *leaves*.
>
> **Step 1:** Compare the borders of the two polygons for intersections. Insert vertices into the polygons at the intersections.
> **Step 2:** Process the nonintersecting polygon borders, separating those contours that are outside the clip polygon and those that are inside.
> **Step 3:** Separate the intersection vertices found on all subject polygons into two lists. One is the *entering list,* consisting of those vertices where the polygon edge enters the clip polygon. The other is the *leaving list,* consisting of those vertices where the polygon edge leaves the clip polygon.
> **Step 4:** Now clip.

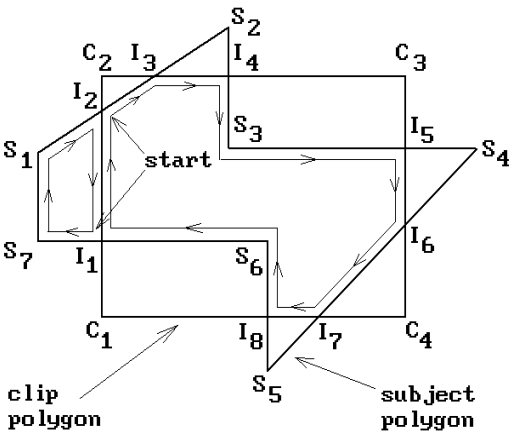


Figure 3.11. Weiler polygon clipping.

(a) Remove an intersection vertex from the entering list. If there is none, then we are done.
(b) Follow the subject polygon vertices to the next intersection.
(c) Jump to the clip polygon vertex list.
(d) Follow the clip polygon vertices to the next intersection.
(e) Jump back to the subject polygon vertex list.
(f) Repeat (b)–(e) until we are back to the starting point.

This process creates the polygons inside the clip polygon. To get those that are outside, one repeats the same steps, except that one starts with a vertex from the leaving list and the clip polygon vertex list is followed in the **reverse** direction. Finally, all holes are attached to their associated exterior contours.

3.3.2.1 Example. Consider the polygons in Figure 3.11. The subject polygon vertices are labeled $\mathbf{S}_i$, those of the clip polygon are labeled $\mathbf{C}_i$, and the intersections are labeled $\mathbf{I}_i$. The entering list consists of $\mathbf{I}_2$, $\mathbf{I}_4$, $\mathbf{I}_6$, and $\mathbf{I}_8$. The leaving list consists of $\mathbf{I}_1$, $\mathbf{I}_3$, $\mathbf{I}_5$, and $\mathbf{I}_7$. Starting Step 4(a) with the vertex $\mathbf{I}_2$ will generate the inside contour

$$\mathbf{I}_2\mathbf{I}_3\mathbf{I}_4\mathbf{S}_3\mathbf{I}_5\mathbf{I}_6\mathbf{I}_7\mathbf{I}_8\mathbf{S}_6\mathbf{I}_1\mathbf{I}_2.$$

Starting Step 4(a) with vertices $\mathbf{I}_1$, $\mathbf{I}_3$, $\mathbf{I}_5$, and $\mathbf{I}_7$ will generate the outside contours

$$\mathbf{I}_1\mathbf{S}_7\mathbf{S}_1\mathbf{I}_2\mathbf{I}_1, \mathbf{I}_3\mathbf{S}_2\mathbf{I}_4\mathbf{I}_3, \mathbf{I}_5\mathbf{S}_4\mathbf{I}_6\mathbf{I}_5, \text{and } \mathbf{I}_7\mathbf{S}_5\mathbf{I}_8\mathbf{I}_7.$$

3.3.3 Liang-Barsky Polygon Clipping

This section gives a brief outline of the Liang-Barsky polygon-clipping algorithm ([LiaB83]). The algorithm is claimed to be twice as fast as the Sutherland-Hodgman clipping algorithm. This algorithm and the next one, the Maillot algorithm, base their success on their ability to detect turning points efficiently. Before we get to the algorithm, some comments on turning points are in order.

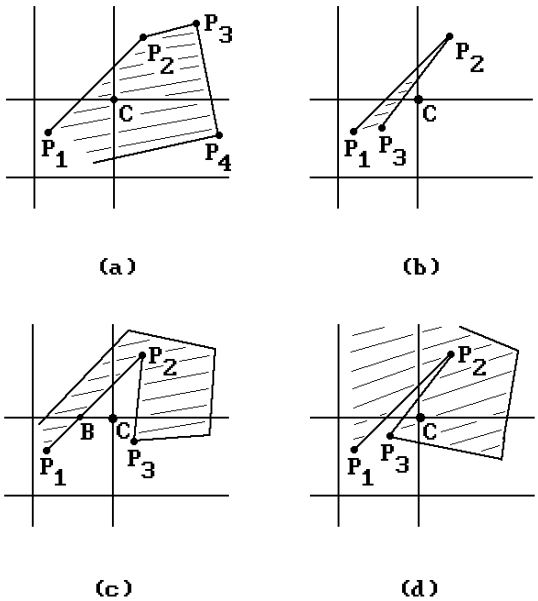


Figure 3.12. Different types of turning points.

Assume that [$\mathbf{P}_1$,$\mathbf{P}_2$] is an edge of a polygon. It is easy to see that the only time that this edge is relevant to the issue of turning points is if it enters or exits a corner region associated to the window. Figure 3.12 shows some cases of polygons (the shaded regions) and how the exiting edge [$\mathbf{P}_1$,$\mathbf{P}_2$] affects whether or not the corner **C** becomes a turning point and needs to be added to the output. One sees the following:

(1) The analysis divides into two cases: whether the polygon is to the right or left of the edge. (Figure 3.12(a,b) versus Figure 3.12(c,d))
(2) There are two subcases that depend on which side of the ray from $\mathbf{P}_2$ to **C** the segment [$\mathbf{P}_2$,$\mathbf{P}_3$] is located.
(3) The decision as to whether a turning point will be needed cannot be made on the basis of only a few edges. In principle one might have to look at all the edges of the polygon first.

It is observation (3) that complicates life for polygon-clipping algorithms that process edges sequentially in one pass. One could simplify life and generate a turning point **whenever** we run into an edge that enters, lies in, or exits a corner region. The problem with this approach is that one will generate bogus edges for our clipped polygon. The polygon in Figure 3.12(c) would generate the "dangling" edge [**B**,**C**]. Bogus edges were already encountered in Sutherland-Hodgman clipping (but for different reasons). These edges might not cause any problems, as in the case where one is simply filling two-dimensional regions. On the other hand, one would like to minimize the number of such edges, but avoiding them entirely would be very complicated with some algorithms like the Liang-Barsky and Maillot algorithm.

With this introduction, let us describe the Liang-Barsky algorithm. We shall be brief because it does not contain much in the way of new insights given the fact that

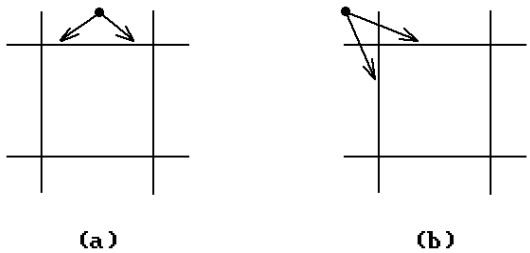


Figure 3.13. Testing for turning points.

it is built on the approach they use in their line-clipping algorithm. Furthermore, another algorithm, the Maillot algorithm, is better. Our discussion here will follow [FVFH90], who have modified the algorithm so that it is easier to follow although it has the potential disadvantage of creating more bogus edges.

Consider Figure 3.13. We extend the edges of our rectangular window to divide the plane into nine regions. If we are at a point of the polygon that lies in one of the four outside regions that meet the window in an edge, then the next edge of the polygon can only meet the window in that edge. See Figure 3.13(a). On the other hand, if the point is in one of the four corner regions, then the next edge could meet the window in one of two possible edges. Without look-ahead, we really cannot tell whether the adjacent corner of the window will become a turning point and will have to be added to the output polygon. In such a situation we shall play it safe and always include the corner point (even though this may create some of these unwanted edges that we have been talking about). This is the first point to make about the algorithm.

The other point is that the algorithm rests on the idea of entry and exit points for the edges of the polygon that correspond to the entry and exit values used in the line-clipping algorithm described in Section 3.2.3. By analyzing these points one can tell if an edge intersects the window or if it gives rise to a turning point. As we work our way through the edges of the polygon, assume that the current edge $\mathbf{e} = [\mathbf{p}_i, \mathbf{p}_{i+1}]$ is neither vertical nor horizontal. Then $\mathbf{e}$ will intersect all the boundary lines of our window. If we parameterize the line containing $\mathbf{e}$ in the form $\mathbf{p}_i + t\mathbf{p}_i\mathbf{p}_{i+1}$, then the four intersection points correspond to four parameter values and can again be classified as two entry and two exit points. We shall denote these associated parameter values by t_in1, t_in2, t_out1, and t_out2, respectively. It is easy to see that the smallest of these is an entry value and we shall let t_in1 be that one. The largest is an exit value and we shall let t_out2 be that one. Nothing can be said about the relative size of the remaining two values in general. From Section 3.2.3 we know, however, that if t_in2 < t_out1, then the line intersects the window and if t_out2 < t_in1, then the line intersects a corner region.

If a line does not intersect the window, then it must intersect three corner regions. The conditions for that are that $0 < \text{t_out1} \leq 1$ and $0 < \text{t_out2} \leq 1$. The last statement also holds if the line intersects the window. Putting all these facts together leads to Algorithm 3.3.3.1. However, we were assuming in the discussion that edges were neither horizontal nor vertical. We could deal with such lines by means of special cases, but the easiest way to deal with them and preserve the structure of Algorithm 3.3.3.1 is to use a trick and assign dummy $\pm\infty$ values to the missing entering and leaving parameters. See [FVFH90].

```
for each edge e of polygon do
    begin
        Determine the direction of e;     { Used to tell in what order the bounding
                                            lines of the clip region will be hit }
        Find exit t values;
        if t_out2 > 0 then find t_in2;
        if t_in2 > t_out1
            then                          { No visible segment }
                begin
                    if 0 < t_out1 ≤ 1 then OutputVertex (turning vertex);
                end
            else
                begin
                    if (0 < t_out1) and (t_in2 ≤ 1) then
                        begin             { Part of segment is visible }
                            if 0 ≤ t_in2
                                then OutputVertex (appropriate side intersection)
                                else OutputVertex (starting vertex);
                            if t_out1 ≤ 1
                                then OutputVertex (appropriate side intersection)
                                else OutputVertex (ending vertex);
                        end
                end;
        if 0 < t_out2 ≤ 1 then OutputVertex (appropriate corner);
    end;
```

Algorithm 3.3.3.1. Overview of a Liang-Barsky polygon-clipping algorithm.

3.3.4 Maillot Polygon Clipping

The Maillot clipping algorithm ([Mail92]) clips arbitrary polygons against a rectangular window. It uses the well-known Cohen-Sutherland clipping algorithm for segments as its basis and then finds the correct turning points for the clipped polygon by maintaining an additional bit of information. As indicated earlier, it is speedy determination of turning points that is crucial for polygon clipping, and this algorithm does it very efficiently. We shall use the same notation that was used in Section 3.2.1. We also assume that the **same** encoding of points is used. This is very important; otherwise, the tables Tcc and Cra below **must** be changed.

Let **P** be a polygon defined by the sequence of vertices $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n, \mathbf{p}_{n+1} = \mathbf{p}_0$. Algorithm 3.3.4.1 gives a top-level description of the Maillot algorithm.

In addition to the Cohen-Sutherland trivial rejection cases, Maillot's algorithm subjects all vertices of the polygon to one extra test, which he calls the "basic turning point test." This test checks for the case where the current point lies in one of the four

```
p := p0;   cp := c(p);
for  i:=1 to  n+1  do
     begin
          q := pi;   cq := c(q);

          { Clip the segment [p,q] as in Cohen-Sutherland algorithm }
          DoCSClip ();

          if  segment [p,q] is outside clipping region  then  TestForComplexCase;

          DoBasicTurningPointTest ();

          p := q;   cp := cq;
     end;
```

Algorithm 3.3.4.1. Overview of Maillot polygon-clipping algorithm.

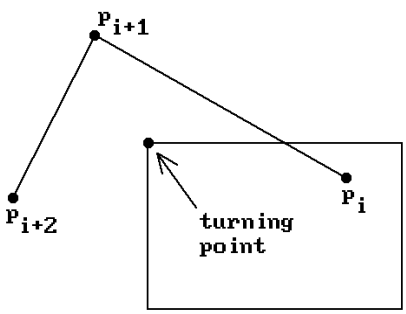


Figure 3.14. A turning point case.

corners outside the window. One probably needs to add a turning point to the clipped polygon in this case. See Figure 3.14. We said "probably" because if the current point is considered in isolation (without looking at its predecessors), then to always automatically add the point may cause us to add the same corner several times in a row. See points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, and $\mathbf{p}_{i+2}$ in Figure 3.14. In the implementation of Maillot's algorithm, we do not try to eliminate such redundancies. If this is not desired, then extra code will have to be added to avoid it.

If all of a polygon's edges meet the window, then the basic turning point test is all that is needed to clip it correctly. For polygons that have edges entirely **outside** the clipping region, one needs to do more. Figure 3.15 shows all (up to symmetry) generic cases that need to be handled in this more complex situation. The following terminology is useful for the case of edges outside the clipping region.

Notation. A point that lies in a region with code 0001, 0010, 0100, or 1000 will be called a *1-bit point*. A point that lies in a region with code 0011, 0110, 1100, or 1001

will be called a *2-bit point*. A segment is called an *x-y segment* if its start point is an x-bit point and its endpoint is a y-bit point.

Knowing the type of segment that one has is important for the algorithm. This is why an extra bit is used in the encoding of points. It is stuck at the left end of the original Cohen-Sutherland code. Below is an overview of the actions that are taken in the TestForComplexCase procedure. Refer to Figure 3.15.

The 1–1 Segment Cases (Segments **a** and **b).** Either the two points have the same code (segment **a**) and no turning point needs to be generated or they have different codes (segment **b**). In the latter case there is one turning point that can be handled by the basic turning point test. The code for the corner for this turning point is computed from the **or** of the two codes and a lookup table (the Tcc table in the code).

The 2–1 and 1–2 Segment Cases (Segments **c** and **d).** In this case one point of the segment has a 1-bit code and the other, a 2-bit code.

(a) The endpoint is the point with the 1-bit code (segment **c**): If both codes **and** to a nonzero value (segment [**P**,**R**] in Figure 3.16(a)), there is no turning point. If both

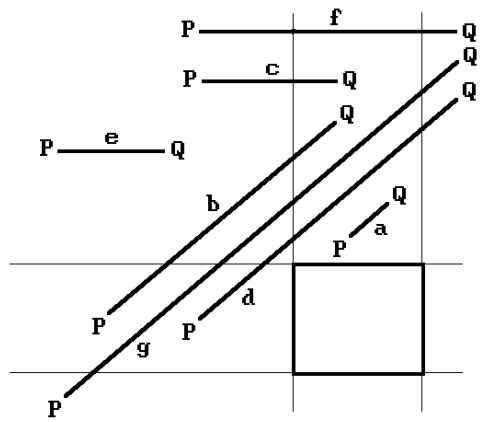


Figure 3.15. Turning point cases in Maillot algorithm.

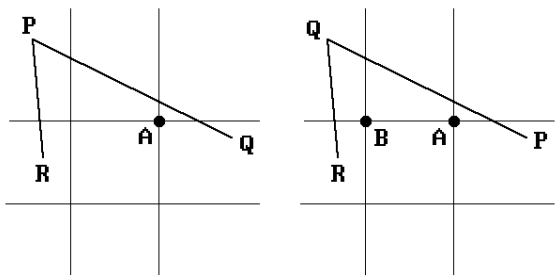


Figure 3.16. Turning point tests. (a) (b)

codes **and** to zero, then we need to generate a turning point that depends on the two codes. A lookup table (Tcc in the code) is used for this.

(b) The endpoint has the 2-bit code (segment **d**): The case where the **and** of both codes is nonzero is handled by the basic turning point test (segment [**R**,**Q**] in Figure 3.16(b). If both codes **and** to zero, we need two turning points. The first one depends on the two codes and is determined by again using a lookup table (Tcc in the code). The other is generated by the basic turning point test (segment [**P**,**Q**] in Figure 3.16(b)).

As an example of how the Tcc table is generated, consider the segment [**P**,**Q**] in Figure 3.16(b). In the figure there are two turning points **A** and **B**. The basic turning point test applied to **Q** will generate **B**. Let us see how **A** is generated. How can one compute the code, namely 3, for this turning point? Maillot defines the sixteen element Tcc table in such a way that the following formula works:

$$\text{newCode} = \text{code}(\mathbf{Q}) + \text{Tcc}[\text{code}(\mathbf{P})]$$

For the 1–1, 2–1, and 1–2 segment cases only four entries of Tcc are used in conjunction with this formula. Four other entries are set to 1 and used in the 2–2 segment case discussed below when it runs into a 1–1 segment. The remaining eight of the entries in Tcc are set to 0.

The 2–2 Segment Case (Segments **e, f** and **g).** There are three subcases.

(a) Both points have the same code (segment **e**): No turning point is needed here.

(b) Both codes **and** to a nonzero value (segment **f**): Apply the basic turning point test to the end point.

(c) Both codes **and** to a zero value (segment **g**): There will be two turning points. One of them is easily generated by the basic turning point test. For the other one we have a situation as shown in Figure 3.17 and we must decide between the two possible choices **A** or **B**. Maillot uses a midpoint subdivision approach wherein the edge is successively divided into two until it can be handled by the previous cases. The

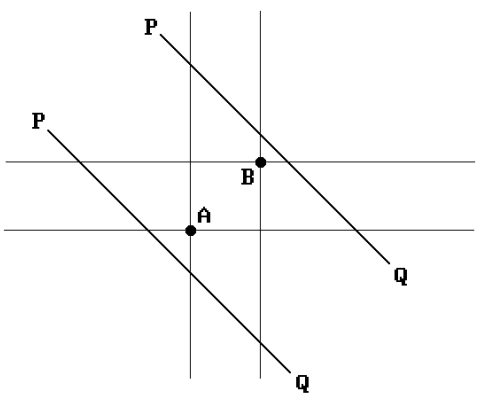


Figure 3.17. 2-2 segment case turning points.

number of subdivisions required depends on the precision used. For 32-bit integers, there will be less than 32 subdivisions.

Maillot presents a C implementation of his algorithm in [Mail92]. Our version of this algorithm is Algorithm 3.3.4.2 below. The main difference is that we tried to be as clear as possible by using extra auxiliary functions and procedures. To be efficient, however, all these calls should be eliminated and the code put inline.

As mentioned earlier, Maillot's algorithm uses the Cohen-Sutherland clipping algorithm. One can use the implementation in Section 3.2.1 for this except that the extended encoding function (ExtendedCsCode) shown in Algorithm 3.3.4.3 should be

```
{ Constants }
MAXSIZE  = 1000; { maximum size of pnt2d array }
NOSEGM   = 0;    { segment was rejected }
SEGM     = 1;    { segment is at least partially visible }
CLIP     = 2;    { segment was clipped }
TWOBITS  = $10;  { flag for 2-bit code }

{ Two lookup tables for finding turning point.
   Tcc is used to compute a correct offset.
   Cra gives an index into the clipRegion array for turning point coordinates. }
integer array [0..15] Tcc = (0, −3, −6,1,3,0,1,0,6,1,0,0,1,0,0,0);
integer array [0..15] Cra = (−1, −1, −1,3, −1, −1,2, −1, −1,1, −1, −1,0, −1, −1, −1);

pnt2d = record
    real x, y;
end;

pnt2ds = pnt2d array [0..MAXSIZE];

{ Global variables }
{ The clipping region [xmin,xmax]×[ymin,ymax] bounds listed in order:
   (xmin,ymin),(xmax,ymin),(xmin,ymax),(xmax,ymax) }
array [0..3] of pnt2d clipRegion;

pnt2d   startPt; { start point of segment }
integer startC;  { code for start point }
integer startC0; { saves startC for next call to CS_EndClip }
pnt2d   endPt;   { endpoint of segment }
integer endC;    { code for endpoint }
integer aC;      { used by procedure TwoBitEndPoint }
```

Algorithm 3.3.4.2. The Maillot polygon-clipping algorithm.

```
procedure M_Clip (ref pnt2ds inpts; integer numin);
                  ref pnt2ds outpts; ref integer numout)
{ inpts[0..numin−1] defines the input polygon with inpts[numin−1] = inpts[0] .
  The clipped polygon is returned in outpts[0..numout−1]. It is assumed that
  the array outpts is big enough. }
begin
    integer i;

    numout := 0;

    { Compute status of first point. If it is visible, it is stored in outpts array. }
    if CS_StartClip () > 0 then
        begin
            outpts[numout] := startPt;
            Inc (numout);
        end;

    { Now the rest of the points }
    for i:=1 to numin-1 do
        begin
            cflag := CS_EndClip (i);

            startC0 := endC; { endC may get changed }
            if SegMetWindow (cflag)
                then
                    begin
                        if Clipped (cflag) then
                            begin
                                outpts[numout] := startPt;
                                Inc (numout);
                            end;
                        outpts[numout] := endPt;
                        Inc (numout);
                    end
                else if TwoBitCase (endC)
                    then TwoBitEndPoint ()
                    else OneBitEndPoint ();

            { The basic turning point test }
            if TwoBitCase (endC) then
                begin
                    outpts[numout] := clipRegion[Cra[endC and not (TWOBITS)]];
                    Inc (numout);
                end;

            startPt := inpts[i];
        end;
```

Algorithm 3.3.4.2. *Continued*

```
    { Now close the output }
    if  numout > 0  then
        begin
            outpts[numout] := outpts[0];
            Inc (numout);
        end
end;  { M_Clip }


boolean function SegMetWindow (integer cflag)
return ( (cflag and SEGM) ≠ 0 );

boolean function Clipped (integer cflag)
{ Actually, this function should return true only if the first point is clipped;
  otherwise we generate redundant points. }
return ( (cflag and CLIP) ≠ 0 );

boolean function TwoBitCase (integer cflag)
return ( (cflag and TWOBITS) ≠ 0 );

procedure TwoBitEndPoint ()
{ The line has been rejected and we have a 2-bit endpoint. }
if  (startC and endC and (TWOBITS − 1)) = 0  then
    begin
        { The points have no region bits in common.  We need to generate
          an extra turning point - which one is specified by Cra table. }
        if  TwoBitCase (startC)
            then  BothAreTwoBits ()              { defines aC for this case }
            else  aC := endC + Tcc[startC];      { 1-bit start point, 2-bit endpoint }

        outpts[numout] := clipRegion[Cra[aC and not (TWOBITS)]];
        Inc (numout);
    end;  { TwoBitEndPoint }

procedure BothAreTwoBits ()
{ Determines what aC should be by doing midpoint subdivision. }
begin
    boolean notdone;
    pnt2d Pt1, Pt2, aPt;

    notdone := true;
    Pt1 := startPt;
    Pt2 := endPt;
```

Algorithm 3.3.4.2. *Continued*

```
    while notdone do
        begin
            aPt.x := (Pt1.x + Pt2.x)/2.0;
            aPt.y := (Pt1.y + Pt2.y)/2.0;
            aC := ExtendedCsCode (aPt);
            if TwoBitCase (aC)
                then
                    begin
                        if aC = endC
                            then Pt2 := aPt
                            else
                                begin
                                    if aC = startC
                                        then Pt1 := aPt
                                        else notdone := false
                                end
                    end
                else
                    begin
                        if (aC and endC) ≠ 0
                            then aC := endC + Tcc[startC and not (TWOBITS)]
                            else aC := startC + Tcc[endC and not (TWOBITS)];
                        notdone := false;
                    end
        end
end; { BothAreTwoBits }

procedure OneBitEndPoint ()
{ The line has been rejected and we have a 1-bit endpoint. }
if TwoBitCase (startC)
    then
        begin
            if (startC and endC) = 0 then
                endC := startC + Tcc[endC];
        end
    else
        begin
            endC := endC or startC;
            if Tcc[endC] = 1 then endC := endC or TWOBITS;
        end; { OneBitEndPoint }
```

Algorithm 3.3.4.2. *Continued*

```
integer function ExtendedCsCode (pnt2d p)
{ The Maillot extension of the Cohen-Sutherland encoding of points }
begin
    if p.x < clipRegion[0].x then
        begin
            if p.y > clipRegion[3].y then return (6 or TWOBITS);
            if p.y < clipRegion[0].y then return (12 or TWOBITS);
            return (4);
        end;
    if p.x > clipRegion[3].x then
        begin
            if p.y > clipRegion[3].y then return (3 or TWOBITS);
            if p.y < clipRegion[0].y then return (9 or TWOBITS);
            return (1);
        end;
    if p.y > clipRegion[3].y then return (2);
    if p.y < clipRegion[0].y then return (8);
    return (0);
end;
```

Algorithm 3.3.4.3. An extended clipping code function.

used. This function adds the extra bit (TWOBITS), which we talked about. Within the Cohen-Sutherland clipping the extra bit should be ignored.

Two functions in the Maillot algorithm, Algorithm 3.3.4.2, make use of Cohen-Sutherland clipping:

CS_StartClip: This function defines global variables

startPt – the first point of the input polygon
startC – the extended Cohen-Sutherland code for startPt

and returns values SEGM or NOSEGM, where

SEGM means that the point is inside the clipping region
NOSEGM means that the point is outside the clipping region

CS_EndClip (**integer** i): This function uses the global variables startC0 and startPt, clips the segment [startPt,ith point of polygon], and defines the global variables

startC, endC – the extended Cohen-Sutherland code for the start and endpoint, respectively
startPt, endPt – these are the original endpoints if there was no clipping or are the clipped points otherwise

The function returns values SEGM, SEGM **or** CLIP, or NOSEGM, where

SEGM means that the segment is inside the clipping region
CLIP means that the segment is only partly inside the clipping region
NOSEGM means that the segment is outside the clipping region

In conclusion, Maillot claims the following for his algorithm and implementation:

(1) It is up to eight times faster than the Sutherland-Hodgman algorithm and up to three times faster than the Liang-Barsky algorithm.
(2) It can be implemented using only integer arithmetic.
(3) It would be easy to modify so as to reduce the number of degenerate edges.

With regard to point (3), recall again that the Sutherland-Hodgman and Liang-Barsky algorithms also produce degenerate edges sometimes. The Weiler and Vatti algorithm are best in this respect.

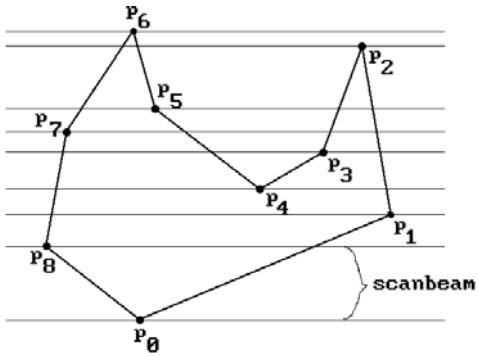
3.3.5 Vatti Polygon Clipping

Quite a few polygon-clipping algorithms have been published. We have discussed several. The Liang-Barsky and Maillot algorithms are better than the Sutherland-Hodgman algorithm, but these algorithms only clip polygons against simple rectangles. This is adequate for many situations in graphics. On the other hand, the Sutherland-Hodgman and Cyrus-Beck algorithms are more general and allow clipping against any convex polygon. The restriction to convex polygons is caused by the fact that the algorithm clips against a sequence of halfplanes and therefore only applies to sets that are the intersection of halfplanes, in other words, convex (linear) polygons. There are situations however where the convexity requirement is too restrictive. The Weiler algorithm is more general yet and works for non-convex polygons. The final two algorithms we look at, the Vatti and Greiner-Hormann algorithms, are also extremely general. Furthermore, they are the most efficient of these general algorithms. The polygons are not constrained in any way now. They can be concave or convex. They can have self-intersections. In fact, one can easily deal with lists of polygons. We begin with Vatti's algorithm ([Vatt92]).

Call an edge of a polygon a *left* or *right edge* if the **interior** of the polygon is to the right or left, respectively. Horizontal edges are considered to be both left and right edges. A key fact that is used by the Vatti algorithm is that polygons can be represented via a set of *left* and *right bounds*, which are connected lists of left and right edges, respectively, that come in pairs. Each of these bounds starts at a local minimum of the polygon and ends at a local maximum. Consider the "polygon" with vertices $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_8$ shown in Figure 3.18(a). The two left bounds have vertices $\mathbf{p}_0, \mathbf{p}_8, \mathbf{p}_7, \mathbf{p}_6$ and $\mathbf{p}_4, \mathbf{p}_3, \mathbf{p}_2$, respectively. The two right bounds have vertices $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and $\mathbf{p}_4, \mathbf{p}_5, \mathbf{p}_6$.

Note. In this section the y-axis will be pointing up (rather than down as usual for a viewport).

Here is an overview of the Vatti algorithm. The first step of the algorithm is to determine the left and right bounds of the clip and subject polygons and to store this information in a *local minima list* (LML). This list consists of a list of matching pairs



(a)

LML = ((B_1, B_2) , (B_3, B_4))

where the bounds B_i are defined by

B_1 = ($\mathbf{p}_0\mathbf{p}_8$, $\mathbf{p}_8\mathbf{p}_7$, $\mathbf{p}_7\mathbf{p}_6$) ,
B_2 = ($\mathbf{p}_0\mathbf{p}_1$, $\mathbf{p}_1\mathbf{p}_2$) ,
B_3 = ($\mathbf{p}_4\mathbf{p}_5$, $\mathbf{p}_5\mathbf{p}_6$) , and
B_4 = ($\mathbf{p}_4\mathbf{p}_3$, $\mathbf{p}_3\mathbf{p}_2$) .

(b)

Figure 3.18. Polygon bounds.

of left-right bounds and is sorted in ascending order by the y-coordinate of the corresponding local minimum. It does not matter if initial horizontal edges are put into a left or right bound. Figure 3.18(b) shows the LML for the polygon in Figure 3.18(a). The algorithm for constructing the LML is a relatively straightforward programming exercise and will not be described here. It can be done with a single pass of the clip and subject polygons.

The bounds on the LML were specified to have the property that their edges are either **all** left edges or **all** right edges. However, it is convenient to have a more general notion of a left or right bound. Therefore, from now on, a *left* or *right bound* will denote any connected sequence of edges only whose **first** edge is required to be a left or right edge, respectively. We still assume that a bound starts at a local minimum and ends at a local maximum. For example, we shall allow the polygon in Figure 3.18(a) to be described by one left bound with vertices $\mathbf{p}_0$, $\mathbf{p}_8$, $\mathbf{p}_7$, $\mathbf{p}_6$, $\mathbf{p}_5$, $\mathbf{p}_4$, $\mathbf{p}_3$, $\mathbf{p}_2$ and one right bound with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$.

The clipped or *output* polygons we are after will be built in stages from sequences of "partial" polygons, each of which is a "V-shaped" list of vertices with the vertices on the left side coming from a left bound and those on the right side coming from a right bound with the two bounds having one vertex in common, namely, the one at the bottom of the "V", which is at a local minimum. Let us use the notation $P[\mathbf{p}_0\mathbf{p}_1 \ldots \mathbf{p}_n]$ to denote the partial polygon with vertices $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$, where $\mathbf{p}_0$ is the first point and $\mathbf{p}_n$, the last. The points $\mathbf{p}_0$ and $\mathbf{p}_n$ are the top of the partial left and right bound, respectively. Some vertex $\mathbf{p}_m$ will be the vertex at a local minimum that connects the two bounds but, since it will not be used for anything, there is no need to indicate this index m in the notation. For example, one way to represent the polygon in Figure 3.18(a) would be as $P[\mathbf{p}_6\mathbf{p}_7\mathbf{p}_8\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3\mathbf{p}_4\mathbf{p}_5\mathbf{p}_6]$ (with m being 3 in this case). Notice how the edges in the left and right bounds are **not** always to the right or left of the interior of the polygon here. In the case of a "completed" polygon, $\mathbf{p}_0$ and $\mathbf{p}_n$ will be the same vertex at a local maximum, but at all the other intermediate stages in the construction of a polygon the vertices $\mathbf{p}_0$ and $\mathbf{p}_n$ may **not** be equal. However, $\mathbf{p}_0$ and $\mathbf{p}_n$ will always correspond to top vertices of the current left and right partial bounds, respectively. For example, $P[\mathbf{p}_7\mathbf{p}_8\mathbf{p}_0\mathbf{p}_1]$ (with m equal to 2) is a legitimate expression describing partial left and right bounds for the polygon in Figure 3.18(a).

A good way to implement these partial polygons is via a circularly linked list, or cycle, and a pointer that points to the last element of the list.

The algorithm now computes the bounds of the output polygons from the LML by scanning the world from the bottom to the top using what are called *scan beams*. A scan beam is a horizontal section between two scan lines (not necessarily adjacent), so that each of these scan lines contains at least one vertex from the polygons but there are **no** vertices in between them. Figure 3.18(a) shows the scan beams and the scan lines that determine them for that particular polygon. The scan beams are the regions between the horizontal lines. It should be noted here that the scan lines that determine the scan beams are not computed all at once but incrementally in a bottom-up fashion. The information about the scan beams is kept in a *scan beam list* (SBL), which is an ordered list ordered by the y-coordinates of all the scan lines that define the scan beams. This list of increasing values will be thought of as a stack. As we scan the world, we also maintain an *active edge list* (AEL), which is an ordered list consisting of all the edges intersected by the current scan beam.

When we begin processing a scan beam, the first thing we do is to check the LML to see if any of its bound pairs start at the bottom of the scan beam. These bounds correspond to local minima and may start a new output polygon or break one into two depending on whether the local minimum starts with a left-right or right-left edge pair. After any new edges from the LML are added to the AEL, we need to check for intersections of edges within a scan beam. These intersections affect the output polygons and are dealt with separately first. Finally, we process the edges on the AEL. Algorithm 3.3.5.1 summarizes this overview of the Vatti algorithm.

To understand the algorithm a little better we look at some more of its details. The interested reader can find a much more thorough discussion with abstract programs and explicit data structures in the document VattiClip on the accompanying CD. The UpdateLMLandSBL procedure in Algorithm 3.3.5.1 finds the bounds of a polygon, adds them to LML, and also updates SBL. Finding a bound involves finding the edges that make them up and initializing their data structure that maintains the information that we need as we go along. For example, we keep track of the x-coordinate of their intersection with the bottom of the current scan beam. We call this the *x-value* of the edge. The edges of the AEL are ordered by these values with ties being broken using their slope. We also record the *kind* of an edge which refers to whether it belongs to the clip or subject polygon. Two edges are called *like* edges if they are of the same kind and *unlike* edges otherwise. The partial polygons that are built and that, in the end, may become the polygons that make up the clipped polygon are called the *adjacent polygons* of their edges.

Because horizontal edges complicate matters, in order to make dealing with horizontal edges easier, one assumes that the matching left and right bound pairs in the LML list are "normalized". A *normalized* left and right bound pair satisfies the following properties:

(1) All consecutive horizontal edges are combined into one so that bounds do not have two horizontal edges in a row.
(2) No left bound has a bottom horizontal edge (any such edges are shifted to the right bound).
(3) No right bound has a top horizontal edge (any such edges are shifted to the left bound).

```
{ Global variables }
real list        SBL;   { an ordered list of distinct reals thought of as a stack}
bound pair list  LML;   { a list of pairs of matching polygon bounds }
edge list        AEL;   { a list of nonhorizontal edges ordered by x-intercept
                          with the current scan line}
polygon list     PL;    { the finished output polygons are stored here as algorithm
                          progresses }

polygon list function Vatti_Clip (polygon subjectP; polygon clipP)
{ The polygon subjectP is clipped against the polygon clipP.
  The list of polygons which are the intersection of subjectP and clipP is returned to the
  calling procedure. }
begin
    real yb, yt;

    Initialize LML, SBL to empty;

    { Define LML and the initial SBL }
    UpdateLMLandSBL (subjectP, subject); { subject and clip specify a subject }
    UpdateLMLandSBL (clipP, clip);       { or clip polygon, respectively }

    Initialize PL, AEL to empty;

    yb := PopSBL ();                     { bottom of current scan beam }
    repeat
        AddNewBoundPairs (yb);           { modifies AEL and SBL }
        yt := PopSBL ();                 { top of current scan beam }
        ProcessIntersections (yb,yt);
        ProcessEdgesInAEL (yb,yt);
        yb := yt;
    until Empty (SBL);

    return (PL);
end;
```

Algorithm 3.3.5.1. The Vatti polygon-clipping algorithm.

We introduce some more terminology. Some edges and vertices that one encounters or creates for the output polygons will belong to the bounds of the clipped polygon, others will not. Let us call a vertex or an edge a *contributing* or *noncontributing vertex* or *edge* depending on whether or not it belongs to the output polygons. With regard to vertices, if a vertex is not a local minimum or maximum, then it will be called a *left* or *right intermediate* vertex depending on whether it belongs to a left or right bound, respectively. Because the overall algorithm proceeds by taking

the appropriate action based on the vertices that are encountered, we shall see that it therefore basically reduces to a careful analysis of the following three cases:

(1) The vertex is a local minimum.
(2) The vertex is a left or right intermediate vertex.
(3) The vertex is a local maximum.

Local minima are encountered when elements on the LML become active. Intermediate vertices and local maxima are encountered when scanning the AEL. Intersections of edges also give rise to these three cases.

Returning to Algorithm 3.3.5.1, the first thing that happens in the main loop is to check for new bound pairs that start at the bottom of the current scan beams. If any such pairs exist, then we have a case of two bounds starting at a vertex **p** that is a local minimum. We add their first nonhorizontal edges to the AEL and the top y-values of these to the SBL. The edges are flagged as being a **left** or **right** edge. We determine if the edges are **contributing** by a parity test and flag them accordingly. An edge of the subject polygon is contributing if there are an odd number of edges from the clip polygon to its left in the AEL. Similarly, an edge of the clip polygon is contributing if there are an odd number of edges from the subject polygon to its left in the AEL. If the vertex is contributing, then we create a new partial polygon P[**p**] and associate this polygon to both edges. Note that to determine whether or not an edge is contributing or noncontributing we actually have to look at the geometry only for the first nonhorizontal edge of each bound. The bound's other edges will be of the same type as that one.

The central task of the main loop in the Vatti algorithm is to process the edges on the AEL. If edges intersect, we shall have to do some preprocessing (procedure ProcessIntersections), but right now let us skip that and describe the actual processing, namely, procedure ProcessEdgesInAEL. Because horizontal edges cause substantial complications, we separate the discussion into two cases. We shall discuss the case where there are no horizontal edges first.

If an edge does not end at the top of the current scan beam, then we simply update its x-value to the x-coordinate of the intersection of the edge with the scan line at the top of the scan beam. If an edge **does** end at the top of the scan beam, then the action we take is determined by the type of the top end vertex **p**. The vertex can either be an intermediate vertex or a local maximum.

If the vertex **p** is a left or right intermediate vertex, then the vertex is added at the beginning or end of the vertex list of its adjacent polygon, depending on whether it is a left or right edge, respectively. The edge is replaced on the AEL by its successor edge which inherits the adjacent polygon and left/right flag of the old edge.

If the vertex **p** is a local maximum of the original clip or subject polygons, then a pair of edges from two bounds meet in the point **p**. If **p** is a contributing vertex, then the two edges may belong either to the same or different (partial) polygons. If they have the same adjacent polygons, then this polygon will now be closed once the point **p** is added. If they belong to different polygons, say **P** and **Q**, respectively, then we need to merge these polygons. Let $\mathbf{e}_1$ and $\mathbf{e}_2$ be the top edges for **P** and $\mathbf{f}_1$ and $\mathbf{f}_2$, the top edges for **Q**, so that $\mathbf{e}_1$ and $\mathbf{f}_1$ meet in **p** with $\mathbf{f}_1$ the successor to $\mathbf{e}_1$ in the AEL. See Figure 3.19. Figures 3.19(a) and (c) show specific examples and (b) and (d) generic cases. If $\mathbf{e}_1$ is a left edge of **P** (Figures 3.19(a) and (b)), then we append the vertices of **Q** to the begin-

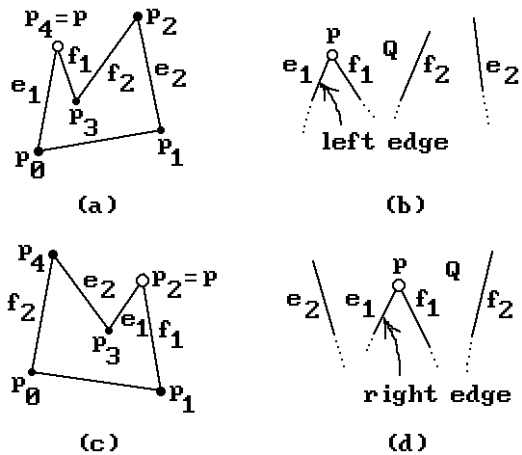


Figure 3.19. Merging polygons.

ning of the vertex list of **P**. If $\mathbf{e}_1$ is a right edge of **P** (Figures 3.19(c) and (d)), then we append the vertices of **P** to the end of the vertex list of **Q**. Note that each of the polygons has two top contributing edges. In either case, after combining the vertices of **P** and **Q**, the two edges $\mathbf{e}_1$ and $\mathbf{f}_1$ become noncontributing. If $\mathbf{e}_1$ was a left edge, then $\mathbf{f}_2$ will be contributing to **P** and the adjacent polygon of $\mathbf{f}_2$ will become **P**. If $\mathbf{e}_1$ was a right edge, then $\mathbf{e}_2$ will be contributing to **Q**. Therefore, the adjacent polygon of $\mathbf{e}_2$ will become **Q**.

When we find a local maximum we know two top edges right away, but if these have different adjacent polygons, then we need to find the other two top edges for these polygons. There are two ways to handle this. One could maintain pointers in the polygons to their current top edges, or one could do a search of the AEL. The first method gives us our edges without a search, but one will have to maintain the pointers as we move from one edge to the next. Which method is better depends on the number of edges versus the number of local maxima. Since there probably are relatively few local maxima, the second method is the recommended one.

Finally, we look at how one deals with intersections of edges within a scan beam. The way that these intersections are handled depends on whether we have like or unlike edges. Like intersections need only be considered if both edges are contributing and in that case the intersection point should be treated as both a left and right intermediate vertex. (Note that in the case of like intersections, if one edge is contributing, then the other one will be also.) Unlike intersections must always be handled. How their intersection point is handled depends on their type, side, and relative position in the AEL.

It is possible to give some precise rules on how to classify intersection points. The **classification rules** are shown in Table 3.3.5.1 in an encoded form. Edges have been specified using the following two-letter code: The first letter indicates whether the edge is a left (L) or right (R) edge, and the second letter specifies whether it belongs to the subject (S) or clip (C) polygon. The resulting vertex type is also specified by a two-letter code: local minimum (MN), local maximum (MX), left intermediate (LI), and right intermediate (RI). Edge codes are listed in the order in which their edges appear in the AEL.

For example, Rule 1 translates into the following: The intersection of a left clip edge and a left subject edge, or the intersection of a left subject edge and a left clip edge, produces

Table 3.3.5.1 Rules that Classify the Intersection Point Between Edges

Unlike edges:	Like edges:
(1) (LC ∩ LS) **or** (LS ∩ LC) → LI	(5) (LC ∩ RC) **or** (RC ∩ LC) → LI **and** RI
(2) (RC ∩ RS) **or** (RS ∩ RC) → RI	(6) (LS ∩ RS) **or** (RS ∩ LS) → LI **and** RI
(3) (LS ∩ RC) **or** (LC ∩ RS) → MX	
(4) (RS ∩ LC) **or** (RC ∩ LS) → MN	

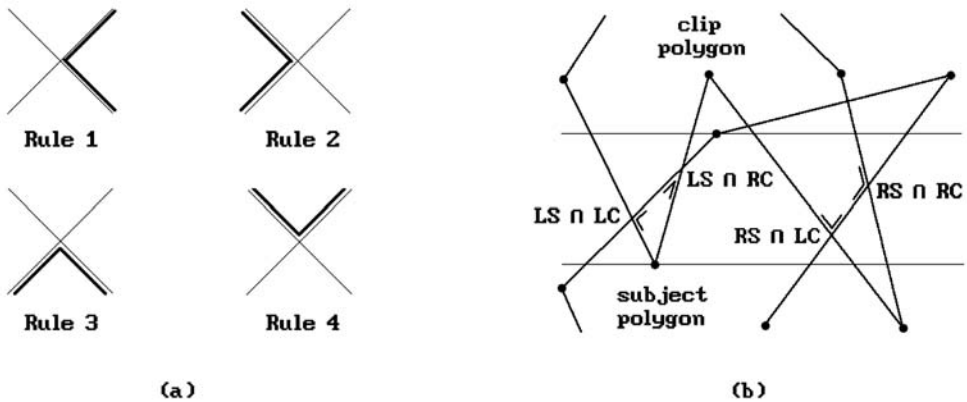


Figure 3.20. Intersection rules.

a left intermediate vertex. Rules 1–4 are shown graphically in Figure 3.20(a). Figure 3.20(b) shows an example of how the rules apply to some real polygon intersections.

As one moves from scan beam to scan beam, one updates the x-values of all the edges (unless they end at the top of the scan beam). Although the AEL is sorted as one enters a new scan beam, if any intersections are found in a scan beam, the AEL will no longer be sorted after the x-values are updated. The list must therefore be resorted, but this can be done in the process of dealing with the intersections. Vatti used a temporary *sorted edge list* (SEL) and an *intersection list* (IL) to identify and store all the intersections in the current scan beam. The SEL is ordered by the x-coordinate of the intersection of the edge with the **top** of the scan beam similarly to the way that the AEL is ordered by the intersection values with the bottom of the scan beams. The IL is a list of nodes specifying the two intersecting edges and also the intersection itself. It is sorted in an increasing order by the y-coordinate of the intersection. The SEL is initialized to empty. One then makes a pass over the AEL comparing the top x-value of the current edge with the top x-values of the edges in the SEL starting at the **right** of the SEL. There will be an intersection each time the AEL edge has a smaller top x-value than the SEL edge. Note that the number of intersections that are found is the same as the number of edge exchanges in the AEL it takes to bring the edge into its correct place at the **top** of the scan beam.

Intersection points of edges are basically treated as vertices. Such "vertices" will be classified in a similar way as the regular vertices. If we get a local maximum, then there are two cases. If two unlike edges intersect, then a contributing edge becomes

a noncontributing edge and vice versa. This is implemented by simply swapping the output polygon pointers. If two like edges intersect, then a left edge becomes a right edge and a right edge becomes a left edge. One needs to swap the intersecting edges in the AEL to maintain the x-sort.

This finishes our discussion of the Vatti algorithm in the case where there are no horizontal edges. Now we address the more complicated general case that allows horizontal edges to exist. (However, we never allow edges to overlap, that is, where they share a common segment.) The only changes we have to make are in procedure ProcessEdgesInAEL. On an abstract level, it is easy to see how horizontal edges should be handled. The classification of vertices described above should proceed as if such edges were absent (had been shrunk to a point). Furthermore, if horizontal edges do not intersect any other edge, then for all practical purposes they could be ignored. The problems arise when intersections exist.

Imagine that the polygons were rotated slightly so that there were no horizontal edges. The edges that used to be horizontal would now be handled without any problem. This suggests how they should be treated when they are horizontal. One should handle horizontal edges the same way that intersections are handled. Note that horizontal edge intersections occur only at the bottom or top of a scan beam. Horizontal edges at local minima should be handled in the AddNewBoundPairs procedure. The others are handled as special cases in that part of the algorithm that tests whether or not an edge ends in the current scan beam. If it does, we also need to look for horizontal edges at the top of the current scan beam and the type classification of a vertex should then distinguish between a local maximum, left intermediate vertex, or right intermediate vertex cases. The corresponding procedures need to continue scanning the AEL for edges that intersect the horizontal edge until one gets past it. One final problem occurs with horizontal edges that are oriented to the left. These would be detected too late, that is, by the time one finds the edge to which they are the successor, we would have already scanned past the AEL edges that intersected them. To avoid this, the simplest solution probably is to make an initial scan of the AEL for all such edges before one checks events at the top of the scan beam and put them into a special left-oriented horizontal edge list (LHL) ordered by the x-values of their left endpoints. Then as one scans the AEL one needs to constantly check the top x-value of an edge for whether it lies inside one of these horizontal edges.

This completes our description of the basic Vatti algorithm. The algorithm can be optimized in the common case of rectangular clip bounds. Another optimization is possible if the clip polygon is fixed (rectangular or not) by computing its bounds only once and initializing the LML to these bounds at the beginning of a call to the clip algorithm.

An attractive feature of Vatti's algorithm is that it can easily be modified to generate trapezoids. This is particularly convenient for scan line–oriented rendering algorithms. Each local minimum starts a trapezoid or breaks an existing one into two depending on whether the local minimum starts with a left-right (contributing case) or right-left (noncontributing case) edge pair. At a contributing local minimum we create a trapezoid. Trapezoids are output at local maxima and left or right intermediate vertices. A noncontributing local minimum should output the trapezoid it is about to split and update the trapezoid pointers of the relevant edges to the two new trapezoids. Vatti compared the performance of the trapezoid version of his algorithm to the Sutherland-Hodgman algorithm and found it to be roughly twice as fast for clipping (the more edges, the more the improvement) and substantially faster if one

does both clipping **and** filling. Because Section 14.4 will describe a special case of the trapezoid form of the Vatti algorithm for use with trimmed surfaces, we postpone any further details on how to deal with trapezoids to there.

Finally, we can also use the Vatti algorithm for other operations than just intersection. All we have to do is replace the classification rules. For example, if we want to output the union of two polygons, use the rules

(1) (LC $\cup$ LS) **or** (LS $\cup$ LC) $\rightarrow$ LI
(2) (RC $\cup$ RS) **or** (RS $\cup$ RC) $\rightarrow$ RI
(3) (LS $\cup$ RC) **or** (LC $\cup$ RS) $\rightarrow$ MN
(4) (RS $\cup$ LC) **or** (RC $\cup$ LS) $\rightarrow$ MX

Local minima of the subject polygon that lie outside the clip polygon and local minima of the clip polygon that lie outside the subject polygon should be treated as contributing local minima.

For the difference of two polygons (subject polygon minus clip polygon) use the rules

(1) (RC – LS) **or** (LS – RC) $\rightarrow$ LI
(2) (RS – LC) **or** (LC – RS) $\rightarrow$ RI
(3) (RS – RC) **or** (LC – LS) $\rightarrow$ MN
(4) (RC – RS) **or** (LS – LC) $\rightarrow$ MX

Local minima of the subject polygon that lie outside the clip polygon should be treated as contributing local minima.

3.3.6 Greiner-Hormann Polygon Clipping

The last polygon-clipping algorithm we consider is the Greiner-Hormann algorithm ([GreH98]). It is very much like Weiler's algorithm but simpler. Like the Weiler and Vatti algorithm it handles any sort of polygons including self-intersecting ones. Furthermore, it, like Vatti's algorithm, can be modified to return the difference and union of polygons, not just their intersections.

Suppose that we want to clip the subject polygon **S** against the clip polygon **C**. What we shall do is find the part of the boundary of **S** in **C**, the part of the boundary of **C** in **S**, and then combine these two parts. See Figure 3.21. Since we allow self-intersecting polygons, one needs to be clear about when a point is considered to be inside a polygon. Greiner-Hormann use the winding number $\omega(\mathbf{p},\gamma)$ of a point **p** with respect to a parameterized curve γ. They define a point **p** to be in a polygon **P** if the winding number of the point with respect to the boundary curve of **P** is odd. (The oddness or evenness of the winding number with respect to a curve is independent of how the curve is parameterized.)

Polygons are represented by doubly-linked lists of vertices. The algorithm proceeds in three phases. One will find it helpful to compare the steps with those of the Weiler algorithm as one reads.

Phase 1. We compare each edge of the subject polygon with each edge of the clip polygon, looking for intersections. If we find one, we insert it in the appropriate place

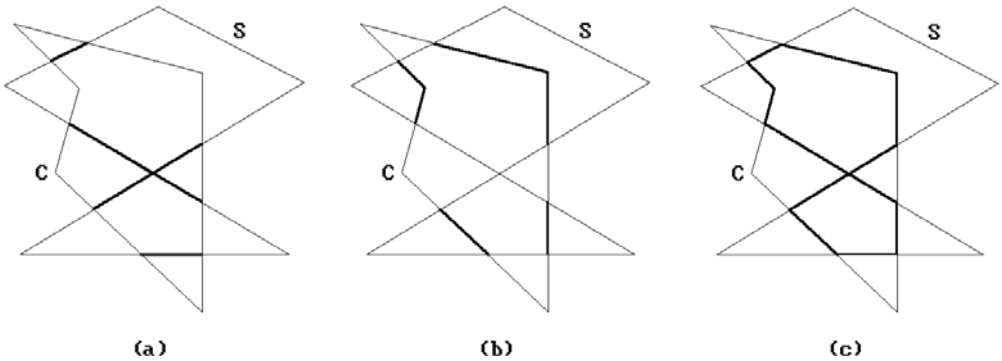


Figure 3.21. Greiner-Hormann polygon clipping.

```
vertex = record
  float           x, y;
  vertex pointer  next, prev;
  boolean         intersect;
  boolean         entry;
  vertex pointer  neighbor;
  float           alpha;
  vertex pointer  nextPoly;
end;
```

Data 3.3.6.1. The Greiner-Hormann vertex structure.

in **both** polygons' vertex lists. If there are no intersections, then either one polygon is contained in the other or they are disjoint. These cases are checked for easily and we then exit the algorithm in this case with our answer.

Phase 2. We traverse each polygon's new vertex lists marking any intersection points as either entry or exit points. This is done by checking whether the first vertex of each polygon lies inside the other polygon or not using the winding number. The rest of the tagging as entry or exit points is then easy.

Phase 3. This stage actually creates the intersection polygons. We start at an intersection point of the subject polygon and then move along its point list either forward or backward depending on its entry-exit flag. If we are at an entry point, then we move forward, otherwise, backward. When we get to another intersection point, we move over to the other polygon's list.

The data structure used for vertices is shown in Data 3.3.6.1. In the case of an intersection point, if the **entry** field is false, then the point is an exit point. At an intersection vertex in one of the polygon's vertex lists the field **neighbor** points to the corresponding vertex in the other polygon's vertex list. The field **alpha** for an intersection point specifies the position of the intersection relative to the two endpoints of the edge

```
vertex pointer current;

while  more unprocessed subject intersection points  do
    begin
        current := pointer to first remaining unprocessed subject intersection point;
        NewPolygon (P);
        NewVertex (current);
        repeat
            if  current→entry
                then
                    repeat
                        current := current→next;
                        NewVertex (current);
                    until  current→intersect
                else
                    repeat
                        current := current→prev;
                        NewVertex (current);
                    until  current→intersect
            current := current→neighbor;
        until  Closed (P);
    end;
```

Algorithm 3.3.6.1. Algorithm for Greiner-Hormann's Phase 3.

containing this intersection point. Because the intersection polygon may consist of several polygons, these polygons are linked with this field. The first vertex of each intersection polygon list has its **nextPoly** field point to the first vertex of the next intersection polygon.

The basic steps for Phase 3 are shown in Algorithm 3.3.6.1. The procedure NewPolygon starts a new polygon P and NewVertex creates a new vertex for this polygon and adds it to the end of its vertex list. Figure 3.22 shows the data structure that is created for a simple example.

The algorithm uses an efficient edge intersection algorithm and handles degenerate cases of intersections by perturbing vertices slightly.

The advantage of the Greiner-Horman algorithm is that it is relatively simple and the authors claim their algorithm can be more than twice as fast as the Vatti algorithm. The reason for this is that Vatti's algorithm also checks for self-intersections which is not done here. Of course, if one knows that a polygon does not have self-intersections, then the extra work could be avoided in Vatti's algorithm also. The disadvantage of the algorithm is that one does not get any trapezoids but simply the boundary curve of the intersection. In conclusion, the Greiner-Horman algorithm is a good one if all one wants is boundaries of polygons because it is simple and yet fast.

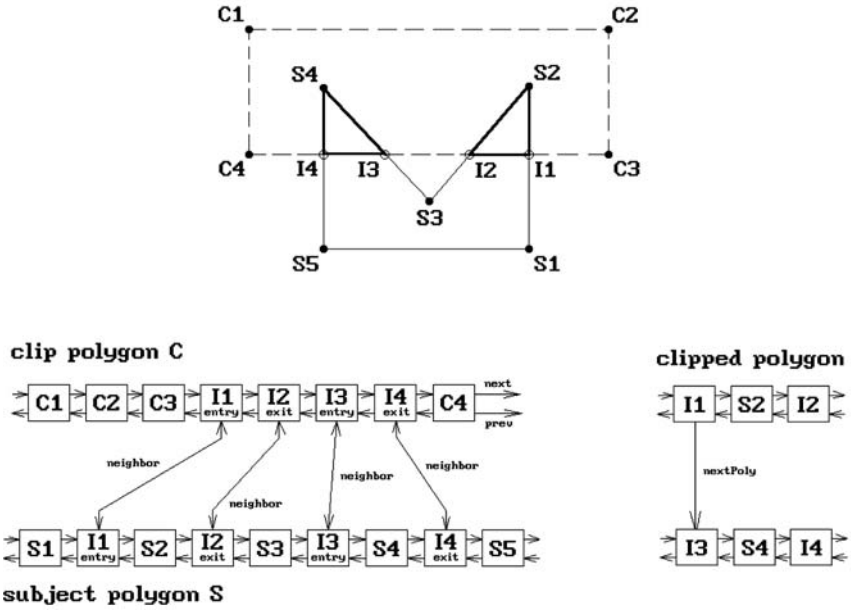


Figure 3.22. The Greiner-Hormann data structures.

3.4 Text Clipping

The topic of text generation and display is a very complex one. We shall barely scratch the surface here.

Characters can be displayed in many different styles and sizes and each such overall design style is called a *typeface* or *font*. Fonts are defined in one of several ways:

Bit-Mapped Fonts. Each character is represented by a rectangular bitmap. All the characters for a particular font are stored in a special part of the graphics memory and then mapped to the frame buffer when needed.

Vector Fonts. Each character is represented by a collection of line segments.

Outline Fonts. Each character's outline is represented by a collection of straight line segments or spline curves. This is more general than vector fonts. An attractive feature of both vector and outline fonts is that they are device independent and are easily scaled, rotated, and transformed in other ways. In either case, one has the option of scan converting them into the frame buffer on the fly or precomputing them and storing the bitmaps in memory. Defining and scan converting outline fonts gets very complicated if one wants the result to look nice and belongs to what is called *digital typography*.

Two overall strategies that are used to clip text are:

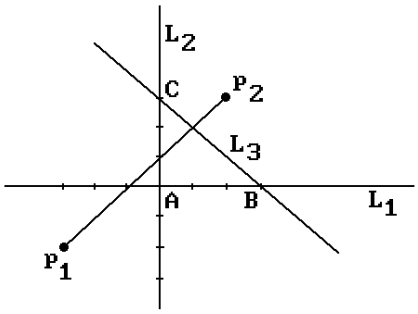


Figure 3.23. A Cyrus-Beck clipping example.

All-or-Nothing String Clipping. Here one computes the size of the rectangle that contains the string and only maps the string to the frame buffer if the rectangle fits entirely into the window in which the string is to be displayed.

All-or-Nothing Character Clipping. Here one clips on a character-by-character basis. One computes the size of the rectangle that contains a given character and only maps the character to the frame buffer if the rectangle fits entirely into the window in which it is to be displayed.

The all-or-nothing approaches are easy to implement because it is easy to check if one rectangle is inside another. The all-or-nothing character clipping approach is often quite satisfactory. A more precise way to clip is to clip on the bit level. What this means in the bit-mapped font case is that one clips the rectangular bitmap of each character against the window rectangle and displays that part which is inside. In the vector or outline font case, one would clip the curve that defines a character against the window using one of the line-clipping algorithms and then scan converts only the part of the character that lies in the window.

3.5 Exercises

Section 3.2.2.

3.2.2.1 Let $\mathbf{p}_1 = (-4,-2)$ and $\mathbf{p}_2 = (2,3)$. Let $\mathbf{A} = (0,0)$, $\mathbf{B} = (3,0)$, and $\mathbf{C} = (0,3)$. Work out the steps of the Cyrus-Beck clipping algorithm and compute the $[a_i,b_i]$s that are generated when clipping the segment $[\mathbf{p}_1,\mathbf{p}_2]$ against triangle **ABC**. See Figure 3.23. Assume that the lines $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$ are defined by equations $y = 0$, $x = 0$, and $x + y = 3$, respectively.

3.6 Programming Projects

1. Clipping (Section 3.3.5 and 3.3.6)

Implement either the Vatti or Greiner-Hormann clipping algorithm in such a way so that it handles all three set operations $\cap$, $\cup$, and $-$.

CHAPTER 4

Transformations and the Graphics Pipeline

Prerequisites: Chapters 2 and 3 in [AgoM05]. Chapter 20 for Section 4.14.

4.1 Introduction

In this chapter we combine properties of motions, homogeneous coordinates, projective transformations, and clipping to describe the mathematics behind the three-dimensional computer graphics transformation pipeline. With this knowledge one will then know all there is to know about how to display three-dimensional points on a screen and subsequent chapters will not have to worry about this issue and can concentrate on geometry and rendering issues. Figure 4.1 shows the main coordinate systems that one needs to deal with in graphics and how they fit into the pipeline. The solid line path includes clipping, the dashed line path does not.

Because the concept of a coordinate system is central to this chapter, it is worth making sure that there is no confusion here. The term "coordinate system" for $\mathbf{R}^n$ means nothing but a "frame" in $\mathbf{R}^n$, that is, a tuple consisting of an orthonormal basis of vectors together with a point that corresponds to the "origin" of the coordinate system. The terms will be used interchangeably. In this context one thinks of $\mathbf{R}^n$ purely as a set of "points" with no reference to coordinates. Given one of these abstract points $\mathbf{p}$, one can talk about the coordinates of $\mathbf{p}$ with respect to one coordinate system or other. If $\mathbf{p} = (x,y,z) \in \mathbf{R}^3$, then x, y, and z are of course just the coordinates of $\mathbf{p}$ with respect to the standard coordinate system or frame $(\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3,\mathbf{0})$.

Describing the coordinate systems and maps shown in Figure 4.1 and dealing with that transformation pipeline in general occupies Sections 4.2–4.7. Much of the discussion is heavily influenced by Blinn's excellent articles [Blin88b,91a–c,92]. They make highly recommended reading. Section 4.8 describes what is involved in creating stereo views. Section 4.9 discusses parallel projections and how one can use the special case of orthographic projections to implement two-dimensional graphics in a

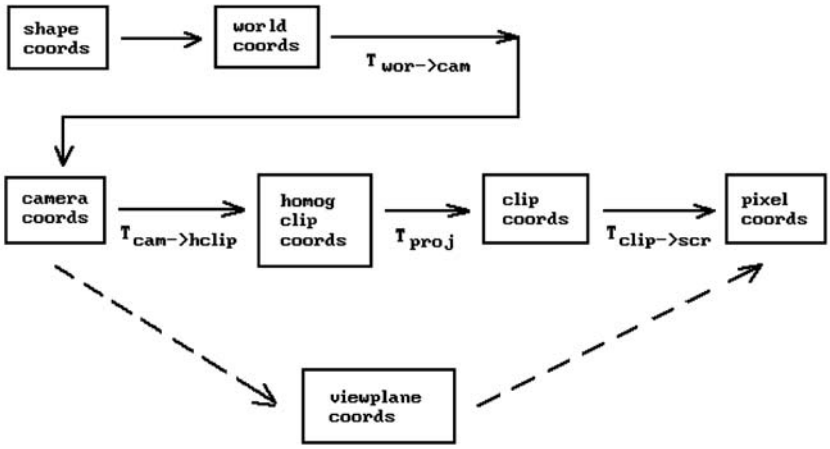


Figure 4.1. The coordinate system pipeline.

three-dimensional environment. Section 4.10 discusses some advantages and disadvantages to using homogeneous coordinates in computer graphics. Section 4.11 explains how OpenGL deals with projections. The reconstruction of objects and camera data is the subject of Section 4.12 and the last graphics pipeline related topic of this chapter. The last two sections of the chapter are basically further examples of transformations and their uses. Section 4.13 takes another look at animation, but from the point of view of robotics. This subject, interesting in its own right, is included here mainly to reinforce the importance of understanding transformations and frames. Next, Section 4.14 explains how quaternions are an efficient way to express transformations and how they are particularly useful in animation. We finish the chapter with some concluding remarks in Section 4.15.

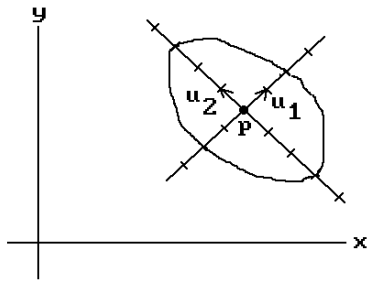
4.2 From Shape to Camera Coordinates

This section describes the first three coordinate systems in the graphics pipeline. In what follows, we shall use the term "shape" as our generic word for a geometric object independent of any coordinate system.

The World Coordinate System. This is the usual coordinate system with respect to which the **user** defines objects.

The Shape Coordinate System. This is the coordinate system used in the actual definition of a shape. It may very well be different from the world coordinate system. For example, the standard conics centered around the origin are very easy to describe. A good coordinate system for the ellipse in Figure 4.2 is defined by the indicated frame $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{p})$. In that coordinate system its equation is simply

Figure 4.2. A shape coordinate system.



$$\frac{x^2}{4}+\frac{y^2}{9}=1.$$

The equation of that ellipse with respect to the standard world coordinate system would be much more complicated.

The Camera Coordinate System. A view of the world obtained from a central projection onto a plane is called a *perspective view*. To specify such view we shall borrow some ideas from the usual concept of a camera (more precisely, a *pinhole camera* where the lens is just a point). When taking a picture, a camera is at a particular position and pointing in some direction. Being a physical object with positive height and width, one can also rotate the camera, or what we shall consider as its "up" direction, to the right or left. This determines whether or not the picture will be "right-side up" or "upside down." Another aspect of a camera is the film where the image is projected. We associate the plane of this film with the view plane. (In a real camera the film is behind the lens, whose position we are treating as the location of the camera, so that an inverted picture is cast onto it. We differ from a real camera here in that for us the film will be in front of the lens.) Therefore, in analogy with such a "real" camera, let us define a *camera* (often referred to as a *synthetic camera*) as something specified by the following data:

a location **p**
a *"view" direction* **v** (the direction in which the camera is looking)
an *"up" direction* **w** (specifies the two-dimensional orientation for the camera)
a real number d (the distance that the view plane is in front of the camera)

Clearly, perspective views are defined by such *camera data* and are easily manipulated by means of it. We can view the world from any point **p**, look in any direction **v**, and specify what should be the top of the picture. We shall see later that the parameter d, in addition to specifying the view plane, will also allow us to zoom in or out of views easily.

A camera and its data define a camera coordinate system specifed by a *camera frame* ($\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3,\mathbf{p}$). See Figure 4.3(a). This is a coordinate system where the camera sits at the origin looking along the positive z-axis and the view plane is a plane parallel to the x-y plane a distance d above it. See Figure 4.3(b). We define this coordinate system from the camera data as follows:

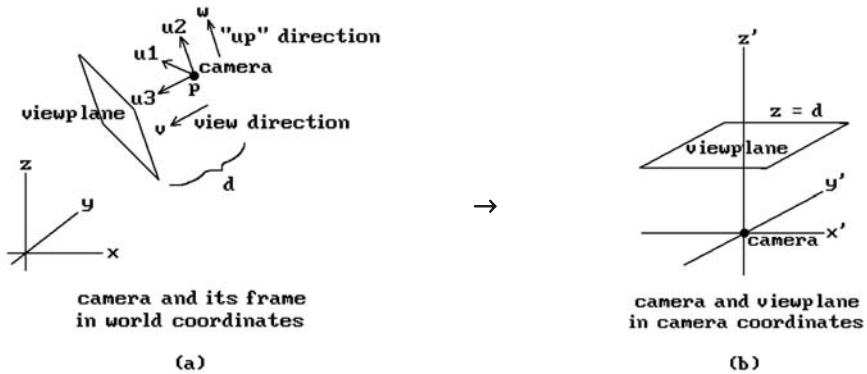


Figure 4.3. The camera coordinate system.

$$\begin{aligned} \mathbf{u}_3 &= \mathbf{v}/|\mathbf{v}| \\ \mathbf{u}_2 &= \mathbf{w}/|\mathbf{w}| \\ \mathbf{u}_1 &= \mathbf{u}_3 \times \mathbf{u}_2. \end{aligned} \tag{4.1}$$

These last two axes will be the same axes that will be used for the viewport. There were only two possibilities for $\mathbf{u}_1$ in equations (4.1). Why did we choose $\mathbf{u}_3 \times \mathbf{u}_2$ rather than $\mathbf{u}_2 \times \mathbf{u}_3$? Normally, one would take the latter because a natural reaction is to choose orientation-**preserving** frames; however, to line this x-axis up with the x-axis of the viewport, which one always wants to be directed to the right, we must take the former. (The easiest way to get an orientation-preserving frame here would be to replace $\mathbf{u}_3$ with $-\mathbf{u}_3$. However, in the current situation, whether or not the frame is orientation-preserving is not important since we will not be using it as a motion but as a change of coordinates transformation.)

Although an up direction is needed to define the camera coordinate system, it is not always convenient to have to define this direction explicitly. Fortunately, there is a natural default value for it. Since a typical view is from some point looking toward the origin, one can take the z-axis as defining this direction. More precisely, one can use the orthogonal projection of the z-axis on the view plane to define the second axis $\mathbf{u}_2$ for the camera coordinate system. In other words, one can define the camera frame by

$$\begin{aligned} \mathbf{u}_3 &= \mathbf{v}/|\mathbf{v}| \\ \mathbf{u}_2 &= \mathbf{w}/|\mathbf{w}|, \qquad \text{where} \quad \mathbf{w} = \mathbf{e}_3 - (\mathbf{e}_3 \bullet \mathbf{u}_3)\mathbf{u}_3 \\ \mathbf{u}_1 &= \mathbf{u}_3 \times \mathbf{u}_2. \end{aligned} \tag{4.2}$$

As it happens, we do not need to take the complete cross product to find $\mathbf{u}_1$, because the z-coordinate of $\mathbf{u}_1$ is zero. The reason for this is that $\mathbf{e}_3$ lies in the plane generated by $\mathbf{u}_2$ and $\mathbf{u}_3$ and so $\mathbf{u}_1$ is orthogonal to $\mathbf{e}_3$. It follows that if $\mathbf{u}_3 = (u_{31}, u_{32}, u_{33})$ and $\mathbf{u}_2 = (u_{21}, u_{22}, u_{23})$, then

$$\mathbf{u}_1 = (u_{32}u_{23} - u_{22}u_{33}, u_{33}u_{21} - u_{23}u_{31}, 0).$$

It is also easy to show that $\mathbf{u}_1$ is a positive scalar multiple of $(u_{32},-u_{31},0)$ (Exercise 4.2.1), so that

$$\mathbf{u}_1 = \frac{1}{\sqrt{u_{31}^2 + u_{32}^2}}(u_{32}, -u_{31}, 0). \tag{4.3}$$

Although this characterization of $\mathbf{u}_1$ is useful and easier to remember than the cross product, it is not as efficient because it involves taking a square root.

Note that there is one case where our construction does not work, namely, when the camera is looking in a direction parallel to the z-axis. In that case the orthogonal projection of the z-axis on the view plane is the zero vector. In this case one can **arbitrarily** use the orthogonal projection of the **y-axis** on the view plane to define $\mathbf{u}_2$. Fortunately, in practice it is rare that one runs into this case. If one does, what will happen is that the picture on the screen will most likely suddenly flip around to some unexpected orientation. Such a thing would not happen with a real camera. One can prevent it by keeping track of the frames as the camera moves. Then when the camera moves onto the z-axis one could define the new frame from the frames at previous nearby positions using continuity. This involves a lot of extra work though which is usually not worth it. Of course, if it **is** important to avoid these albeit rare occurrences then one can do the extra work or require that the user specify the desired up direction explicitly.

Finally, given the frame $F = (\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3,\mathbf{p})$ for the camera, then the world-to-camera coordinate transformation $T_{wor\to cam}$ in Figure 4.1 is the map

$$\mathbf{q} \to (\mathbf{q}-\mathbf{p})\left(\mathbf{u}_1^T\mathbf{u}_2^T\mathbf{u}_3^T\right) = (\mathbf{q}-\mathbf{p})M, \tag{4.4}$$

where M is the 3×3 matrix that has the vectors $\mathbf{u}_i$ as its columns.

We begin with a two-dimensional example.

4.2.1 Example. Assume that the camera is located at $\mathbf{p} = (5,5)$, looking in direction $\mathbf{v} = (-1,-1)$, and that the view plane is a distance $d = 2$ in front of the camera. See Figure 4.4. The problem is to find $T_{wor\to cam}$.

Solution. Let $\mathbf{u}_2 = \mathbf{v}/|\mathbf{v}| = (-1/\sqrt{2}, -1/\sqrt{2})$ ($\mathbf{u}_2$ plays the role of $\mathbf{u}_3$ here). The "up" direction is determined by $\mathbf{e}_2$ in this case, but all we have to do is switch the first and second coordinate of $\mathbf{u}_2$ and change one of the signs, so that $\mathbf{u}_1 = (-1/\sqrt{2}, 1/\sqrt{2})$. We now have the camera frame $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{p})$. It follows that $T = T_{wor\to cam}$ is the map

$$(x, y) \to (x-5, y-5)\begin{pmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

In other words,

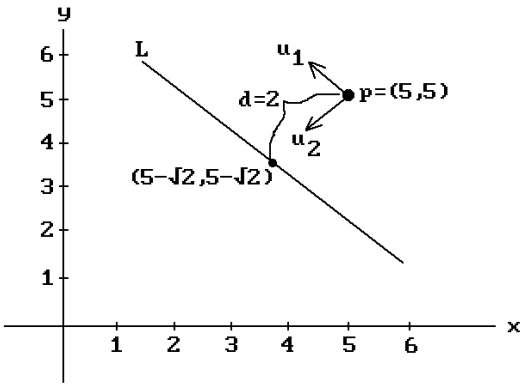


Figure 4.4. Transforming from world to camera coordinates.

$$T_{wor\to cam}: \quad x' = -\frac{1}{\sqrt{2}}(x-5) + \frac{1}{\sqrt{2}}(y-5) = -\frac{1}{\sqrt{2}}x + \frac{1}{\sqrt{2}}y$$
$$y' = -\frac{1}{\sqrt{2}}(x-5) - \frac{1}{\sqrt{2}}(y-5) = -\frac{1}{\sqrt{2}}x - \frac{1}{\sqrt{2}}y + 5\sqrt{2}$$

As a quick check we compute $T(5,5) = (0,0)$ and $T(5-\sqrt{2}, 5-\sqrt{2}) = (0,2)$, which clearly are the correct values.

Next, we work through a three-dimensional example.

4.2.2 Example. Assume that the camera is located at $\mathbf{p} = (5,1,2)$, looking in direction $\mathbf{v} = (-1,-2,-1)$, and that the view plane is a distance $d = 3$ in front of the camera. The problem again is to find $T_{wor\to cam}$.

Solution. Using equations (4.2) we get

$$\mathbf{u}_3 = \frac{1}{\sqrt{6}}(-1,-2,-1)$$
$$\mathbf{u}_2 = \mathbf{w}/|\mathbf{w}| = \frac{1}{\sqrt{30}}(-1,-2,5), \quad \text{where} \quad \mathbf{w} = \frac{1}{6}(-1,-2,5)$$
$$\mathbf{u}_1 = \mathbf{u}_3 \times \mathbf{u}_2 = \frac{1}{\sqrt{5}}(-2,1,0).$$

It follows that

$$T_{wor\to cam}: \quad x' = \frac{-2}{\sqrt{5}}(x-5) + \frac{1}{\sqrt{5}}(y-1)$$
$$y' = \frac{-1}{\sqrt{30}}(x-5) + \frac{-2}{\sqrt{30}}(y-1) + \frac{5}{\sqrt{30}}(z-2)$$
$$z' = \frac{-1}{\sqrt{6}}(x-5) + \frac{-2}{\sqrt{6}}(y-1) + \frac{-1}{\sqrt{6}}(z-2).$$

Note in the two examples how the frame that defines the camera coordinate system also defines the transformation from world coordinates to camera coordinates and conversely. The frame is the whole key to camera coordinates and look how simple it was to define this frame!

The View Plane Coordinate System. The origin of this coordinate system is the point in the view plane a distance d directly in front of the camera and the x- and y-axis are the same as those of the camera coordinate system. More precisely, if $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3,\mathbf{p})$ is the camera coordinate system, then $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{p}+d\mathbf{u}_3)$ is the view plane coordinate system.

4.3 Vanishing Points

If there is no clipping, then after one has the camera coordinates of a point, the next problem is to project to the view plane z = d. The central projection π of $\mathbf{R}^3$ from the origin to this plane is easy to compute. Using similarity of triangles, we get

$$\pi(x,y,z) = (x',y',d) = (d\,x/z,\, d\,y/z,\, d). \tag{4.5}$$

Let us see what happens when lines are projected to the view plane. Consider a line through a point $\mathbf{p}_0 = (x_0,y_0,z_0)$, with direction vector $\mathbf{v} = (a,b,c)$, and parameterization

$$p(t) = (x(t), y(t), z(t)) = \mathbf{p}_0 + t\mathbf{v}. \tag{4.6}$$

This line is projected by π to a curve $p'(t) = (x'(t),y'(t),d)$ in the view plane, where

$$x'(t) = d\,\frac{x_0 + at}{z_0 + ct} \quad \text{and} \quad y'(t) = d\,\frac{y_0 + bt}{z_0 + ct}; \tag{4.7}$$

It is easy to check that the slope of the line segment from $p'(t_1)$ to $p'(t_2)$ is

$$\frac{y'(t_2) - y'(t_1)}{x'(t_2) - x'(t_1)} = \frac{y_0c - bz_0}{x_0c - az_0},$$

which is independent of t_1 and t_2. This shows that the curve $p'(t)$ has constant slope and reconfirms the fact that central projections project lines into lines (but not necessarily onto).

Next, let us see what happens to $p'(t)$ as t goes to infinity. Assume that $c \neq 0$. Then, using equation (4.7), we get that

$$\lim_{t\to\infty} (x'(t), y'(t)) = (d\,a/c,\, d\,b/c) \tag{4.8}$$

This limit point depends only on the direction vector $\mathbf{v}$ of the original line. What this means is that all lines with the same direction vector, that is, all lines parallel to the

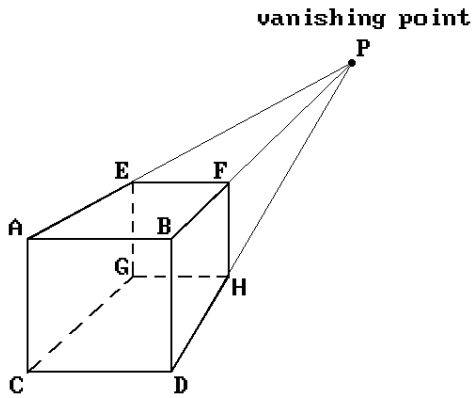


Figure 4.5. Vanishing point.

original line, will project to lines that intersect in a point. If c = 0, then one can check that nothing special happens and parallel lines project into parallel lines.

In the context of the world-to-view plane transformation with respect to a given camera, what we have shown is that lines in the world project into lines in the view plane. Furthermore, the projection of some lines gives rise to certain special points in the view plane. Specifically, let **L** be a line in the world and let equation (4.6) be a parameterization for **L** in camera coordinates. We use the notation in the discussion above.

Definition. If the point in the view plane of the camera that corresponds to the point on the right hand side of equation (4.8) exists, then it is called the *vanishing point* for the line **L** with respect to the given camera or view.

Clearly, if a line has a vanishing point, then this point is well-defined and unique. Any line parallel to such a line will have the same vanishing point. Figure 4.5 shows a projected cube and its vertices. Notice how the lines through the pairs of vertices **A** and **E**, **B** and **F**, **C** and **G**, and **D** and **H** meet in the vanishing point **P**. If we assume that the view direction of the camera is perpendicular to the front face of the cube, then the lines through vertices such as **A**, **B**, and **E**, **F**, or **A**, **C**, and **B**, **D**, are parallel. (This is the c = 0 case.)

Perspective views are divided into three types depending on the number of vanishing points of the standard unit cube (meaning the number of vanishing points of lines parallel to the edges of the cube).

One-point Perspective View. Here we have one vanishing point, which means that the view plane must be parallel to a face of the cube. Figure 4.5 shows such a perspective view.

Two-point Perspective View. Here we have two vanishing points and is the case where the view plane is parallel to an edge of the cube but not to a face. See Figure 4.6 and the vanishing points $\mathbf{P}_1$ and $\mathbf{P}_2$.

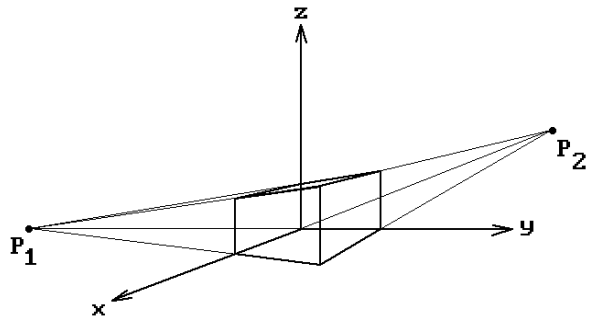


Figure 4.6. Two-point perspective view.

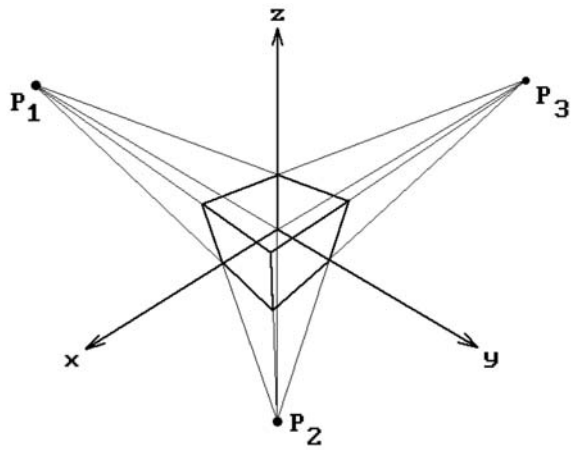


Figure 4.7. Three-point perspective view.

Three-point Perspective View. Here we have three vanishing points and is the case where none of the edges of the cube are parallel to the view plane. See Figure 4.7 and the vanishing points $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$.

Two-point perspective views are the ones most commonly used in mechanical drawings. They show the three dimensionality of an object best. Three-point perspective views do not add much.

4.4 Windows and Viewports Revisited

The simple view of windows and viewports described in Chapter 1 glossed over some important points and so we need to take another look. Assume that [wxmin,wxmax] × [wymin,wymax] and [vxmin,vxmax] × [vymin,vymax] define the window and viewport rectangles, respectively. See Figure 4.8. We shall not change the basic idea that a window specifies **what** we see and that the viewport specifies **where** we see it, but there was a natural implication that it is by **changing** the window that one sees different parts of the world. Is that not how one would scan a plane by moving a rec-

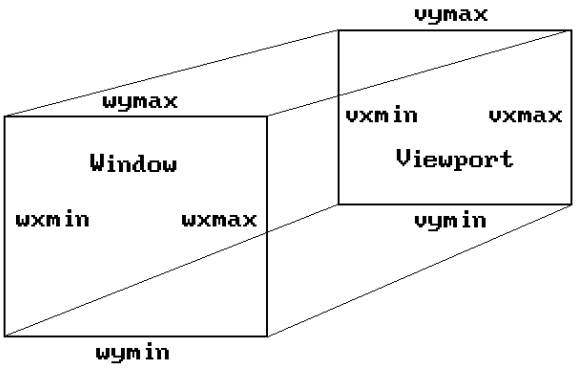


Figure 4.8. The window and viewport rectangles.

tangular window around in it? What is overlooked here is the problem that occurs when the viewport and the window are not the same size rectangle. For example, suppose that the window is the square $[-2,2] \times [-2,2]$ and that the viewport the rectangle $[0,100] \times [0,50]$. What would happen in this situation is that the circle of radius 1 around the origin in the view plane would map to an ellipse centered at (50,25) in the viewport. What we know to be a circle in the world would show up visually as an ellipse on the screen. Would we be happy with that? This is the "aspect ratio" problem. The reader may have noticed this already when implementing some of the programming projects. What can one do to make circles show up as circles?

The best way to deal with the aspect ratio problem would be to let the user change the viewport but not the window. The window would then be chosen to match the viewport appropriately. First of all, users are not interested in such low level concepts anyway and want to manipulate views in more geometric ways by using commands like "pan," "zoom," "move the camera," etc. Secondly, in the case of 3d graphics, from a practical point of view this will in no way affect the program's ability to handle different views. Changing the camera data will have the same effect. In fact, changing the position and direction of the camera gives the program more control of what one sees than simply changing the window. Changing the distance that the view plane is in front of the camera corresponds to zooming. A fixed window would not work in the case of 2d graphics, however. One would have to let the user translate the window and change its size to allow zooming. A translation causes no problem, but the zooming has to be controlled. The size can only be allowed to change by a factor that preserves the height divided by width ratio. There is no reason for a user to know what is going on at this level though. As long as the user is given a command option to zoom in or out, that user will be satisfied and does not need to know any of the underlying technical details.

Returning to the 3d graphics case, given that our default window will be a fixed size, what should this size be? First of all, it will be centered about the origin of the view plane. It should have the same *aspect ratio* (ratio of height to width) as the viewport. Therefore, we shall let the window be the rectangle $[-1,1] \times [-b,b]$, where $b = (vymax - vymin)/(vxmax - vxmin)$. Unfortunately, this is not the end of the story. There is also a *hardware aspect ratio* one needs to worry about. This refers to the fact that

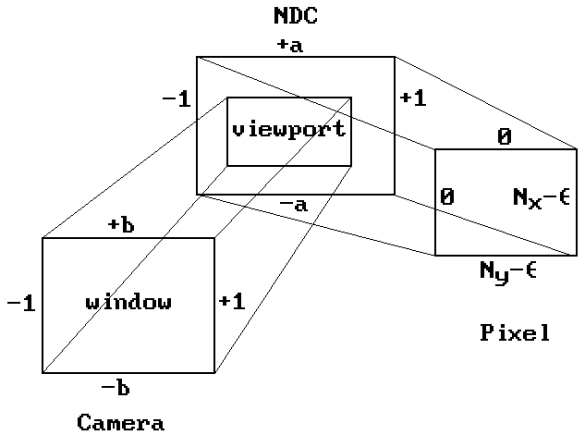


Figure 4.9. Window, normalized viewport, and pixel space.

the dots of the electron beam for the CRT may not be "square." The hardware ratio is usually expressed in the form a = ya/xa with the operating system supplying the values xa and ya. In Microsoft Windows, one gets these values via the calls

xa = GetDeviceCaps (hdc, ASPECTX);

ya = GetDeviceCaps (hdc, ASPECTY);

where hdc is a "device context" and ASPECTX and ASPECTY are system-defined constants.

To take the aspect ratios into account and to allow more generality in the definition of the viewport, Blinn ([Blin92]) suggests using normalized device coordinates (NDC) for the viewport that are separate from pixel coordinates. The normalized viewport in this case will be the rectangle $[-1,1] \times [-a,a]$, where a is the hardware aspect ratio. If N_x and N_y are the number of pixels in the x- and y-direction of our picture in pixel space, then Figure 4.8 becomes Figure 4.9.

We need to explain the "$-\varepsilon$" terms in Figure 4.9. One's first reaction might be that $[0,N_x - 1] \times [0,N_y - 1]$ should be the pixel rectangle. But one needs to remember our discussion of pixel coordinates in Section 2.8. Pixels should be centered at half integers, so that the correct rectangle is $[-0.5,N_x - 0.5] \times [-0.5,N_y - 0.5]$. Next, the map from NDC to pixel space must round the result to the nearest integer. Since rounding is the same thing as adding 0.5 and truncating, we can get the same result by mapping $[-1,1] \times [-a,a]$ to $[0,N_x] \times [0,N_y]$ and truncating. One last problem is that a +1 in the x- or y-coordinate of a point in NDC will now map to a pixel with N_x or N_y in the corresponding coordinate. This is unfortunately outside our pixel rectangle. Rather than looking for this special case in our computation, the quickest (and satisfactory) solution is to shrink NDC slightly by subtracting a small amount ε from the pixel ranges. Smith suggests letting ε be 0.001.

There is still more to the window and viewport story, but first we need to talk about clipping.

4.5 The Clip Coordinate System

Once one has transformed objects into camera coordinates, our next problem is to clip points in the camera coordinate system to the truncated pyramid defined by the near and far clipping planes and the window. One could do this directly, but we prefer to transform into a coordinate system, called the *clip coordinate system* or *clip space*, where the clipping volume is the unit cube $[0,1] \times [0,1] \times [0,1]$. We denote the transformation that does this by $T_{cam \to clip}$. There are two reasons for using this transformation:

(1) It is clearly simpler to clip against the unit cube.
(2) The clipping algorithm becomes independent of boundary dimensions.

Actually, rather than using these coordinates we shall use the associated homogeneous coordinates. The latter define what we shall call the *homogeneous clip coordinate system* or *homogeneous clip space*. Using homogeneous coordinates will enable us to describe maps via matrices and we will also not have to worry about any divisions by zero on our way to the clip stage. The map $T_{cam \to hclip}$ in Figure 4.1 refers to this camera-to-homogeneous-clip coordinates transformation. Let $T_{hcam \to hclip}$ denote the corresponding homogeneous-camera-to-homogeneous-clip coordinates transformation. Figure 4.10 shows the relationships between all these maps. The map T_{proj} is the standard projection from homogeneous to Euclidean coordinates.

Assume that the view plane and near and far clipping planes are a distance d, d_n, and d_f in front of the camera, respectively. To describe $T_{cam \to hclip}$, it will suffice to describe $T_{hcam \to hclip}$.

First of all, translate the camera to (0,0,–d). This translation is represented by the homogeneous matrix

$$M_{tr} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -d & 1 \end{pmatrix}.$$

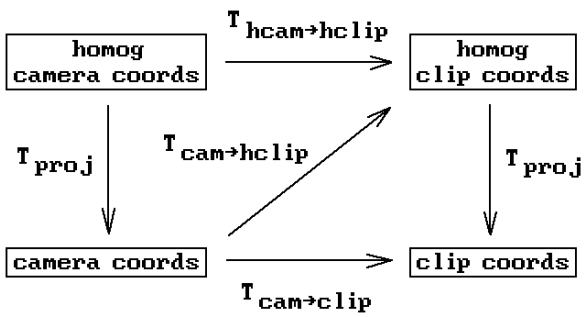


Figure 4.10. The camera-to-clip space transformations.

Next, apply the projective transformation with homogeneous matrix M_{persp}, where

$$M_{persp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.9}$$

To see exactly what the map defined by M_{persp} does geometrically, consider the lines $ax + z = -d$ and $ax - z = d$ in the plane $y = 0$. Note that $(x,y,z,w)\, M_{persp} = (x,y,z,(z/d)+w)$. In particular,

$$\begin{aligned}
(0,0,-d,1)M_{persp} &= (0,0,-d,0) \\
(x,0,-d-ax,1)M_{persp} &= \left(x,0,-d-ax,-\frac{ax}{d}\right) = -\frac{d}{ax}\left(-\frac{d}{a},0,d+\frac{d^2}{ax},1\right) \\
(x,0,-d+ax,1)M_{persp} &= \left(x,0,-d+ax,\frac{ax}{d}\right) = \frac{d}{ax}\left(\frac{d}{a},0,d-\frac{d^2}{ax},1\right).
\end{aligned}$$

This shows that the camera at (0,0,–d) has been mapped to "infinity" and the two lines have been mapped to the lines $x' = -d/a$ and $x' = d/a$, respectively, in the plane $y = 0$. See Figure 4.11. In general, lines through (0,0,–d) are mapped to vertical lines through their intersection with the x-y plane. Furthermore, what was the central projection from the point (0,0,–d) is now an orthogonal projection of $\mathbf{R}^3$ onto the x-y plane. It follows that the composition of M_{tr} and M_{persp} maps the camera off to "infinity," the near clipping plane to $z = d\,(1 - d/d_n)$, and the far clipping plane to $z = d\,(1 - d/d_f)$. The perspective projection problem has been transformed into a simple orthographic projection problem (we simply project (x,y,z) to (x,y,0)) with the clip volume now being

$$[-1,1] \times [-b,b] \times [d(1-d/d_f), d(1-d/d_n)].$$

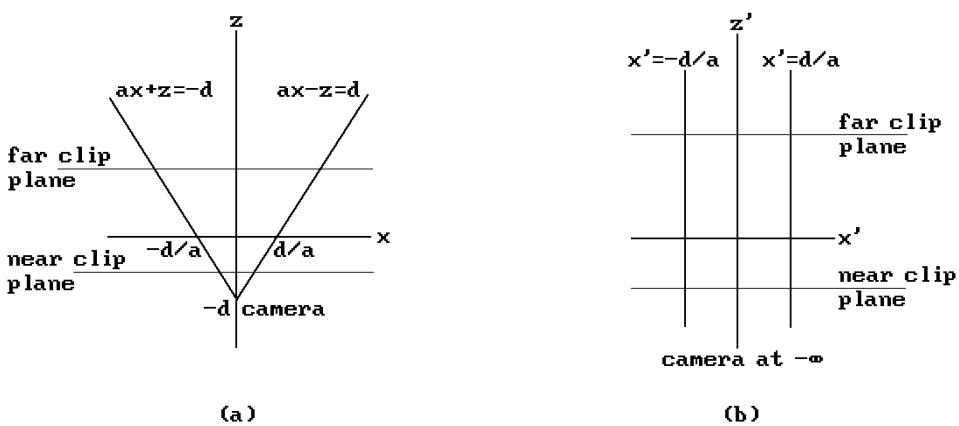


Figure 4.11. Mapping the camera to infinity.

To get this new clip volume into the unit cube, we use the composite of the following maps: first, translate to

$$[0,2]\times[0,2b]\times\left[0,d^2\left(\frac{1}{d_n}-\frac{1}{d_f}\right)\right],$$

and then use the radial transformation which multiplies the x, y, and z-coordinates by

$$\frac{1}{2},\frac{1}{2b},\text{and}\frac{d_n d_f}{d^2(d_f-d_n)},$$

respectively. If M_{scale} is the homogeneous matrix for the composite of these two maps, then

$$M_{scale}=\begin{pmatrix}\frac{1}{2} & 0 & 0 & 0\\ 0 & \frac{1}{2d} & 0 & 0\\ 0 & 0 & \frac{d_n d_f}{d^2(d_f-d_n)} & 0\\ \frac{1}{2} & \frac{1}{2} & -\frac{d_n(d_f-d)}{d(d_f-d_n)} & 1\end{pmatrix},$$

so that

$$M_{hcam\to hclip}=M_{tr}M_{persp}M_{scale}=\begin{pmatrix}\frac{1}{2} & 0 & 0 & 0\\ 0 & \frac{1}{2d} & 0 & 0\\ \frac{1}{2d} & \frac{1}{2d} & \frac{d_f}{d(d_f-d_n)} & \frac{1}{d}\\ 0 & 0 & -\frac{d_n d_f}{d(d_f-d_n)} & 0\end{pmatrix} \tag{4.10}$$

is the matrix for the map $T_{hcam\to hclip}$ that we are after. It defines the transformation from homogeneous camera to homogeneous clip coordinates. By construction the map $T_{cam\to clip}$ sends the truncated view volume in camera coordinates into the unit cube $[0,1]\times[0,1]\times[0,1]$ in clip space.

Note that the camera-to-clip-space transformation does not cost us anything because it is computed only once and right away combined with the world-to-camera-space transformation so that points are only transformed once, not twice.

Finally, our camera-to-clip-space transformation maps three-dimensional points to three-dimensional points. In some cases, such as for wireframe displays, the z-coordinate is not needed and we could eliminate a few computations above. However,

if we want to implement visible surface algorithms, then we need the z. Note that the transformation is not a motion and will deform objects. However, and this is the important fact, it preserves **relative** z-distances from the camera and to determine the visible surfaces we only care about relative and not absolute distances. More precisely, let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two points that lie along a ray in front of the camera and assume that they map to $\mathbf{p}_1'$ and $\mathbf{p}_2'$, respectively, in clip space. If the z-coordinate of $\mathbf{p}_1$ is less than the z-coordinate of $\mathbf{p}_2$, then the z-coordinate of $\mathbf{p}_1'$ will be less than the z-coordinate of $\mathbf{p}_2'$. In other words, the "in front of" relation is preserved. To see this, let $\mathbf{p}_i = (t_i x, t_i y, t_i z)$, $0 < t_1 < t_2$, and $\mathbf{p}_i' = (x_i', y_i', z_i')$. It follows from (4.10) that

$$z_i' = \left(1 - \frac{d_n}{t_i z}\right)\frac{d_f}{(d_f - d_n)}$$

from which it is easy to show that $t_1 < t_2$ if and only if $z_1' < z_2'$.

4.6 Clipping

In the last section we showed how to transform the clipping problem to a problem of clipping against the unit cube in clip space. The actual clipping against the cube will be done in homogeneous clip space using homogeneous coordinates (x,y,z,w). The advantage of homogeneous coordinates was already alluded to: **every** point of camera space is sent to a well-defined point here because values become undefined only when we try to map down to clip space by dividing by w, which may be zero.

Chapter 3 discussed general clipping algorithms for **individual** segments or whole polygons. These have their place, but they are not geared to geometric modeling environments where one often wants to draw **connected** segments. We shall now describe a very efficient clipping algorithm for such a setting that comes from [Blin91a]. It uses the "best" parts of the Cohen-Sutherland, Cyrus-Beck, and Liang-Barsky algorithms.

In homogeneous coordinates halfplanes can be defined as a set of points that have a nonnegative dot product with a fixed vector. For example, the halfplane $ax + by + cz + d \geq 0$, is defined by the vector (a,b,c,d) in homogeneous coordinates. Therefore, by lining up vectors appropriately, any convex region bounded by planes can be defined as the set of points that have nonnegative dot products with a fixed finite set of vectors. In our case, we can use the following vectors for the six bounding planes $x = 0$, $x = 1$, $y = 0$, $y = 1$, $z = 0$, and $z = 1$ for the unit cube $\mathbf{I}^3$:

$$\mathbf{B}_1 = (1,0,0,0), \quad \mathbf{B}_3 = (0,1,0,0), \quad \mathbf{B}_5 = (0,0,1,0),$$
$$\mathbf{B}_2 = (-1,0,0,1), \quad \mathbf{B}_4 = (0,-1,0,1), \quad \mathbf{B}_6 = (0,0,-1,1).$$

If $\mathbf{p} = (x,y,z,w)$, then let $BC_i = BC_i(\mathbf{p}) = \mathbf{p} \bullet \mathbf{B}_i$. We shall call the BC_i the *boundary coordinates* of $\mathbf{p}$. These coordinates are easy to compute:

$$BC_1 = x, \qquad BC_3 = y, \qquad BC_5 = z,$$
$$BC_2 = w - x, \qquad BC_4 = w - y, \qquad BC_6 = w - z.$$

A point will be inside the clip volume if and only if all of its boundary coordinates are nonnegative. If the ith boundary coordinate of a point is nonnegative, then we shall call the point *i-inside*; otherwise, it is *i-out*. Let BC = BC(**p**) = (BC_1(**p**),BC_2(**p**), . . . , BC_6(**p**)) denote the vector of boundary coordinates.

Next, let **p0** and **p1** be two points and set BC0 = BC(**p0**) and BC1 = BC(**p1**). The next table shows the relationship of the segment [**p0**,**p1**] with respect to the ith boundary:

Sign bit $BC0_i$	Sign bit $BC1_i$	Meaning
0	0	Entire segment is i-inside
1	0	Segment straddles boundary, **p0** is i-out
0	1	Segment straddles boundary, **p1** is i-out
1	1	Entire segment is i-out

It will be convenient to record the sign information of a point **p** into a six-bit word called its *outcode* and denote it by CODE(**p**). More precisely, the ith bit of CODE(**p**) will be the sign bit of BC_i(**p**). Let CODE0 = CODE(**p0**) and CODE1 = CODE(**p1**). Simple logical operations on CODE0 and CODE1 now give us a lot of information about the location of the segment. For example, the segment will be inside the clip volume if (CODE0 **or** CODE1) is zero. The segment will be entirely outside the clip volume if (CODE0 **and** CODE1) is nonzero. (Compare this with the Cohen-Sutherland clipping algorithm.)

Whenever the segment crosses the ith clipping plane, we need to find the intersection. This is easy to do if we parameterize the segment, and we have done this sort of thing before. We need to find the t so that

$$(\mathbf{p0} + t(\mathbf{p1} - \mathbf{p0})) \bullet \mathbf{B}_i = 0.$$

With our notation, t = $BC0_i/(BC0_i - BC1_i)$. The segment will intersect the plane only if this t lies in [0,1]. The expression shows that this can only happen if $BC0_i$ and $BC1_i$ have different signs.

Now, the clipping algorithm we are in the process of describing is intended for situations where we want to do a sequence of "DrawTo" and "MoveTo" commands. The flag parameter in the "Clip" procedure is used to distinguish between the two cases and will save us having to write a separate "Clip" procedure for both. The abstract programs are given in Algorithm 4.6.1 with the ViewPt procedure representing the next stage of the graphics pipeline after clipping, namely, the clip-space-to-pixel-space map. A more efficient procedure using goto's, assuming that the trivial rejects are the most common cases, is shown in Algorithm 4.6.2.

Next, we describe the nontrivial stuff that happens when a segment straddles a boundary. We basically use the Liang-Barsky algorithm here. In Algorithm 4.6.3, the variables a0 and a1 keep track of the still-visible part of a segment. MASK is used to select one boundary at a time. Blinn points out that he does the operation CODE0 **or** CODE1 **again** on the theory that it will not get done often and we save storing an

```
homogeneous point  p0, p1;
real            BC0,BC1
6-bit word   CODE0,CODE1;

Procedure Clip ((move,draw) flag)
{We assume that p0, BC0, and CODE0 have been defined. We clip [p0, p1]}
begin
    Calculate BC1, CODE1;

    case  flag  of
        move : DoMoveStuff ();
        draw : DoDrawStuff ();
    end;

    { Update globals }
    [p0,BC0,CODE0] := [p1,BC1,CODE1];
end;

Procedure DoMoveStuff ()
if  CODE1 = 0  then  ViewPt (p1,move);

Procedure DoDrawStuff ()
if  (CODE0 and CODE1) = 0  then
    begin
        if  (CODE0 or CODE1) = 0
            then  ViewPt (p1,draw)
            else  DoNontrivialStuff ()
    end;
```

Algorithm 4.6.1. Abstract programs for clipping using homogeneous coordinates.

unneeded value earlier. He also made all tests as much as possible into integer comparisons to cut down on floating point operations.

There are some limitations to Blinn's clipping algorithm. Although they tend to be more theoretical than practical, one should be aware of them. The problem is that one is clipping to the infinite inverted pyramid in homogeneous coordinate space shown in Figure 4.12(a) when, in fact, one should be clipping to the double pyramid shown in Figure 4.12(b). The points in the negative pyramid will also project to the visible region. On the other hand, the basic graphics pipeline that we have been describing will not introduce any negative w's and so this problem will not arise here. The problem arises only if negative w-coordinates are introduced explicitly or if one wants to represent infinite segments (the complement of a normal segment in a line). If one does want to handle such cases, the quickest way to do it is to draw the world twice, once as described above and then a second time, where the matrix that maps from shape to clip coordinates is multiplied by –1.

```
Procedure Clip ((move,draw) flag)
begin
    label moveit, nontriv, finish;

    Calculate BC1, CODE1;

    if flag = move then goto moveit;
    if (CODE0 and CODE1) ≠ 0 then goto finish;
    if (CODE0 or CODE1) ≠ 0 then goto nontriv;
    ViewPt(p1,draw);

finish:
    [p0,BC0,CODE0] := [p1,BC1,CODE1];
    return;

moveit:
    if CODE1 ≠ 0 then goto finish;
    ViewPt(p1,move);
    goto finish;

nontriv:
    DoNontrivialStuff ();
    goto finish;
end;
```

Algorithm 4.6.2. More efficient clipping using homogeneous coordinates.

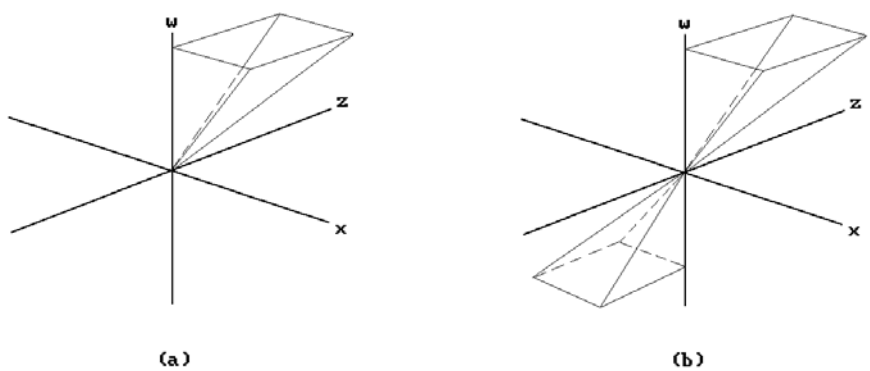


Figure 4.12. Single- and double-clip pyramid.

```
Procedure DoNontrivialStuff ()
begin
    6-bit word BCASE, MASK;
    real       a0, a1, a;
    integer    i;
    homogeneous point p;

    BCASE := CODE0 or CODE1;
    a0 := 0.0;  a1 := 1.0;  MASK := 1;
    for i:=1 to numClipPlanes do
        begin
            if (BCASE and MASK) ≠ 0 then
                begin
                    a := BC0[i] / (BC0[i] − BC1[i]);
                    if (CODE0 and MASK) ≠ 0
                        then a0 := max (a0,a)
                        else a1 := min (a1,a);
                    if a1 < a0 then return;          { reject }
                end;
            Shift MASK left one bit
        end;

    if CODE0 ≠ 0 then
        begin
            p := p0 + a0*(p1 − p0);
            ViewPt (p,move);
        end;

    if CODE1 ≠ 0
        then
            begin
                p := p0 + a1*(p1 − p0);
                ViewPt (p,draw);
            end
        else ViewPt (p1,draw);

end;
```

Algorithm 4.6.3. The nontrivial part of homogeneous coordinate clipping.

4.7 Putting It All Together

We are finally ready to put all the pieces together. See Figure 4.1 again. Starting with some shape we are initially in shape coordinates. We then

(1) transform to world coordinates
(2) transform from world to homogeneous clip coordinates by composing $T_{wor \to cam}$ and $T_{cam \to hclip}$
(3) clip
(4) project (x,y,z,w) down to (x/w,y/w,z/w) in the unit cube of clip space with T_{proj}
(5) map the unit square in the x-y plane of clip space to the viewport
(6) map from the viewport to pixel space

With respect to (4), note that using a front clipping plane does have the advantage that we do not have to worry about a division by zero. Almost, but not quite. There is the very special case of (0,0,0,0) that could occur and hence one needs to check for it (Exercise 4.7.1). It would be complicated to eliminate this case.

Also, because of the clipping step, Blinn suggests a more complete version of the window-to-pixel map than shown in Figure 4.9. See Figure 4.13. The square $[0,1] \times [0,1]$ represents the clipping. This allows one to handle the situation shown in Figure 4.14, where the viewport goes outside the valid NDC range quite easily. One pulls back the clipped viewport

$$[\mathrm{ux\,min}, \mathrm{ux\,max}] \times [\mathrm{uy\,min}, \mathrm{uy\,max}]$$

to the rectangle

$$[\mathrm{wx\,min}, \mathrm{wx\,max}] \times [\mathrm{wy\,min}, \mathrm{wy\,max}]$$

and then uses that rectangle as the window. Only the transformation $T_{cam \to hclip}$ needs to be changed, not the clipping algorithm.

Blinn's approach is nice, but there may not be any need for this generality. A much simpler scheme that works quite well is to forget about the NDC by incorporating the hardware aspect ratio r_h into the window size. Let

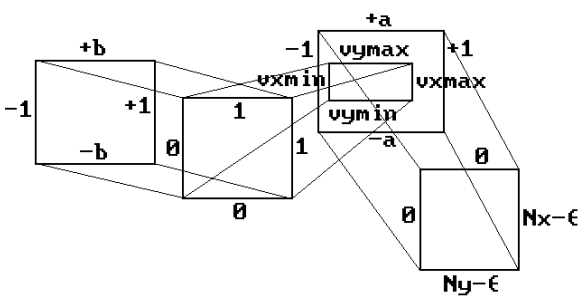


Figure 4.13. From window to pixel coordinates.

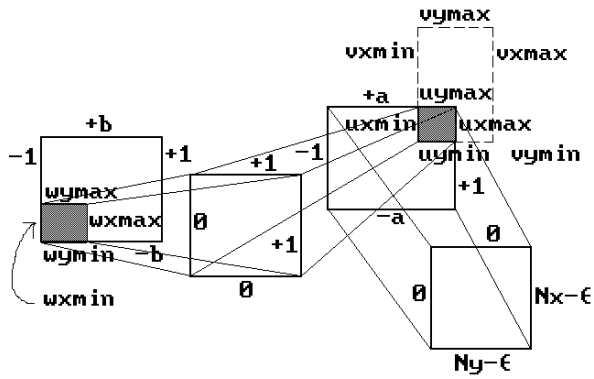


Figure 4.14. General window and viewport example.

$$[\mathrm{vpx\,min}, \mathrm{vpx\,max}] \times [\mathrm{vpy\,min}, \mathrm{vpy\,max}]$$

be the current viewport. Then **fix** the window to be the rectangle $[-1,1] \times [-b,b]$, where

$$b = r_h(\mathrm{y\,max} - \mathrm{y\,min})/(\mathrm{x\,max} - \mathrm{x\,min}).$$

Now map directly from $[0,1] \times [0,1]$ to pixel space. With this window and the view transformations discussed in this chapter, circles will look like circles.

We close with a final remark on clipping. Clipping is expensive and therefore we would rather not do it! In future chapters we shall discuss ways one can often avoid it (by using bounding boxes, the convex hull property of splines, etc.).

4.8 Stereo Views

Occasionally, it is useful to allow the origin of the view plane to be a point other than the one directly in front of the camera. One such case is where one wants to compute stereo views. This involves computing two views, one for each eye.

The Eye Coordinate System. Given a camera, let $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3,\mathbf{p})$ be the camera coordinate system, where the vectors $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$ are defined by equation (4.1) If we think of one eye as being located at $\mathbf{p} + a\mathbf{u}_1 + b\mathbf{u}_2$, then the eye coordinate system with respect to the given camera and some $a, b \in \mathbf{R}$ is defined by the frame $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3,\mathbf{p} + a\mathbf{u}_1 + b\mathbf{u}_2)$. If $a = b = 0$, then this is the same as the camera coordinate system.

It is easy to see that if the coordinates of a point $\mathbf{p}$ in camera coordinates is (x,y,z), then the coordinates of that same point in eye coordinates are $(x - a, y - b, z)$. Furthermore, if $\mathbf{p}$ projects to (x',y',d) in eye coordinates, then it projects to $(x' + a, y' + b, d)$ in camera coordinates. It follows that, using homogeneous coordinates, the only difference in computing the view in camera coordinates to computing it in eye coordinates amounts to replacing the matrix M_{persp} in equation (4.9) by

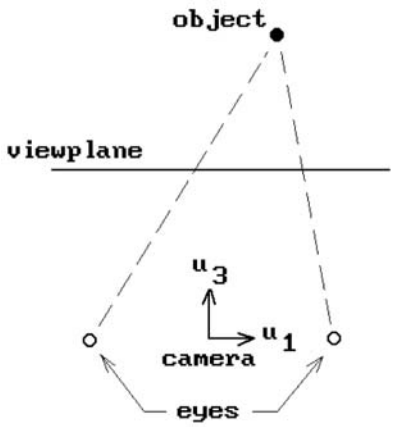


Figure 4.15. Views from two eyes for stereo.

$$M_{eye} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a/d & b/d & 1 & 1/d \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.11)$$

To compute stereo views one would compute two views – one with the eye at $\mathbf{p} + a\mathbf{u}_1$ and one with the eye at $\mathbf{p} - a\mathbf{u}_1$ for some suitable a. See Figure 4.15. The two views are then displayed a suitable distance apart in a viewport. Actually, our discussion here is a simplification of what is involved in stereo rendering and we refer the reader to [Hodg92] for a much more thorough overview.

4.9 Parallel Projections

So far we have dealt solely with perspective views, but there are times when one wants views based on parallel projections. Although this can be thought of as a special case of central projections where the camera is moved to "infinity" along some direction, it is worth considering on its own because one can achieve some simplifications in that case.

Assume that our view plane is the x-y plane and that we are interested in the parallel projection of the world onto that plane using a family of parallel lines. See Figure 4.16.

4.9.1 Proposition. If π is the parallel projection of $\mathbf{R}^3$ onto $\mathbf{R}^2$ with respect to a family of parallel lines with direction vector $\mathbf{v} = (v_1,v_2,v_3)$, then

$$\pi(x,y,z) = (x - z(v_1/v_3), y - z(v_2/v_3), 0).$$

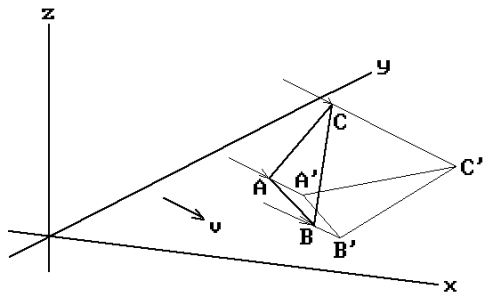


Figure 4.16. A parallel projection onto the x-y plane.

Proof. Exercise 4.9.1.

Passing to homogeneous coordinates, consider the projective transformation T_{par} defined by the matrix

$$M_{par} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{v_1}{v_3} & -\frac{v_2}{v_3} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.12}$$

Our parallel projection onto the x-y plane is then nothing but the Cartesian version of T_{par} followed by the orthogonal projection $(x,y,z) \to (x,y,0)$. It follows that the matrix M_{par} plays the role of the matrix M_{persp} in Section 4.5 (equation (4.9)) in that it reduces a general projection problem into a simple orthogonal projection.

Notice that a parallel projection does not depend on the length of the vector **v**. In fact, any multiple of **v** will define the same projection, as is easily seen from its equations. The parallel projection can also be considered the limiting case of a central projection where one places an eye at a position $\mathbf{v} = (v_1,v_2,v_3) = (a'd,b'd,-d)$ and one lets d go to infinity. This moves the eye off to infinity along a line through the origin with direction vector **v**. The larger d gets, the more parallel are the rays from the eye to the points of an object. The matrix M_{eye} in equation (4.11) (with $a = a'd$ and $b = b'd$) approaches M_{par} because $1/d$ goes to zero.

An even simpler case occurs when the vector **v** is orthogonal to the view plane.

Definition. A parallel projection where the lines we are projecting along are orthogonal to the view plane is called an *orthographic* (or *orthogonal*) *projection*. If the lines have a direction vector that is not orthogonal to the view plane, we call it an *oblique (parallel) projection*. A view of the world obtained via an orthographic or oblique projection is called an *orthographic* or *oblique view*, respectively.

A single projection of an object is obviously not enough to describe its shape.

Definition. An *axonometric projection* consists of a set of parallel projections that shows at least three adjacent faces. A view of the world obtained via an axonometric projection is called an *axonometric view*.

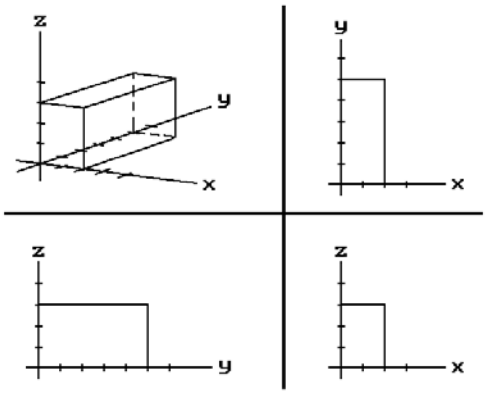


Figure 4.17. Perspective and orthographic views of a $2 \times 5 \times 3$ block.

In engineering drawings one often shows a perspective view along with three orthographic views – a top, front, and side view, corresponding to looking along the z-, y-, and x-axis, respectively. See Figure 4.17. For a more detailed taxonomy of projections see [RogA90].

Finally, in a three-dimensional graphics program one might want to do some 2d graphics. For example, one might want to let a user define curves in the plane. Rather than maintaining a separate 2d structure for these planar objects it would be more convenient to think of them as 3d objects. Using the orthographic projection, one can simulate a 2d world for the user.

4.10 Homogeneous Coordinates: Pro and Con

The computer graphics pipeline as we have described it made use of homogeneous coordinates when it came to clipping. The given reason for this was that it avoids a division by zero problem. How about using homogeneous coordinates and matrices everywhere? This section looks at some issues related to this question. We shall see that both mathematical and practical considerations come into play.

Disadvantages of the Homogeneous Coordinate Representation. The main disadvantage has to do with efficiency. First, it takes more space to store 4-tuples and 4×4 matrices than 3-tuples and 3×4 matrices (frames). Second, 4×4 matrices need more multiplications and additions to act on a point than 3×4 matrices. Another disadvantage is that homogenous coordinates are less easy to understand than Cartesian coordinates.

Advantages of the Homogeneous Coordinate Representation. In a word, the advantage is uniformity. The composite of transformations can be dealt with in a more uniform way (we simply do matrix multiplication) and certain shape manipulations become easier using a homogeneous matrix for the shape-to-world coordinate system

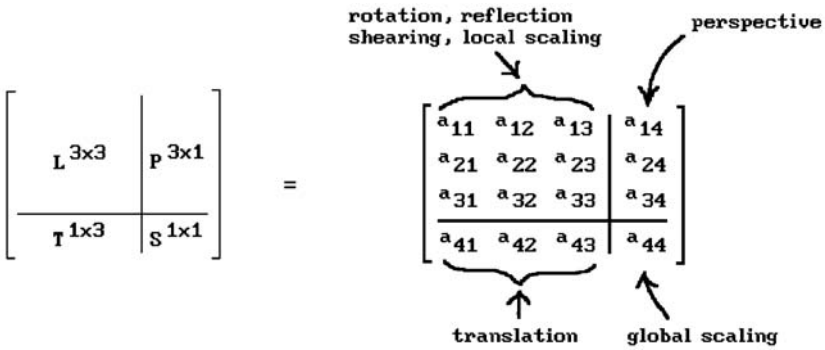


Figure 4.18. Parts of a homogeneous matrix.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{(a)} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{(b)} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 3 & 5 & 1 \end{pmatrix} \quad \text{(c)}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix} \quad \text{(d)} \qquad \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{(e)}$$

Figure 4.19. Transformation examples.

transformation. Furthermore, computer hardware can be optimized to deal with 4×4 matrices to more than compensate for the inefficiency of computation issue mentioned above.

Let us look at the advantage of homogeneous coordinates in more detail. To see the geometric power contained in a 4×4 homogeneous matrix consider Figure 4.18. The matrix can be divided into the four parts L, T, P, and S as shown, each of which by itself has a simple geometric interpretation. The matrix corresponds to an affine map if and only if P is zero and in that case we have a linear transformation defined by L followed by a translation defined by T. If P is nonzero, then some plane will be mapped to infinity. We illustrate this with the examples shown in Figure 4.19.

First, consider L. That matrix corresponds to a linear transformation of $\mathbf{R}^3$. If L is a pure diagonal matrix, then we have a map that expands and/or contracts along

the x, y, or, z axis. For example, the map in Figure 4.19(a) sends the point (x,y,z) to the point (x,7y,z), which expands everything by a factor of 7 in the y direction.

A lower triangular matrix causes what is called a *shear*. What this means is that the map corresponds to sliding the world along a line while expanding or contracting in a possibly not constant manner along a family of lines not parallel to the first line. The same thing holds for upper triangular matrices. For example, consider the matrix M in Figure 4.19(b). The point (x,y,z) gets sent to (x + 3y,y,z). Points get moved horizontally. The bigger the y-coordinate is, the more the point is moved. Note that this map is really an extension of a map of the plane.

Next, consider the map in Figure 4.19(c). This map sends the point (x,y,z) to (x – 1,y + 3,z + 5) and is just a simple translation by the vector (–1,3,5). The map in Figure 4.19(d) sends the homogenous point (x,y,z,1) to (x,7y,z,5), in other words, (x,y,z) is sent to (x/5,7y/5,z/5), which is just a global scaling. Finally, the map in Figure 4.19(e) sends (x,y,z) to (x/(2x + 3y + 1), y/(2x + 3y + 1), z/(2x + 3y + 1)). The plane 2x + 3y + 1 = 0 gets sent to infinity. The map is a two-point perspective map with vanishing points for lines parallel to the x- or y-axes.

We finish this section by describing a way to visualize homogeneous coordinates and why some caution should be exercised when using them.

The standard embedding of $\mathbf{R}^3$ in $\mathbf{P}^3$ maps (x,y,z) to [x,y,z,1]. This means that we can use the space of 4-tuples, that is, $\mathbf{R}^4$, to help us visualize $\mathbf{P}^3$. More precisely, since the lines through the origin correspond in a one-to-one fashion with the points of $\mathbf{P}^3$, we can use the plane w = 1 in $\mathbf{R}^4$ to represent the **real** points of $\mathbf{P}^3$. Furthermore, if someone talks about a point $\mathbf{p}_1$ with homogeneous coordinates (x,y,z,w), then we can pretty much deal with $\mathbf{p}_1$ as if it actually were that 4-tuple in $\mathbf{R}^4$. We need to remember, however, that if $\mathbf{p}_1$ lies on a line through the origin and a point **A** on the plane w = 1, then $\mathbf{p}_1$ and **A** will represent the same point of $\mathbf{P}^3$. See Figure 4.20. Now, once one decides to use homogeneous coordinates for a graphics problem, although one usually starts out with a representative like **A**, after one has applied several transformations (represented by 4 × 4 matrices), one may not assume that the 4-tuple one ends up with will again lie on the plane w = 1. Although one could continually project back down to the w = 1 plane, that would be awkward. It is simpler to let our new

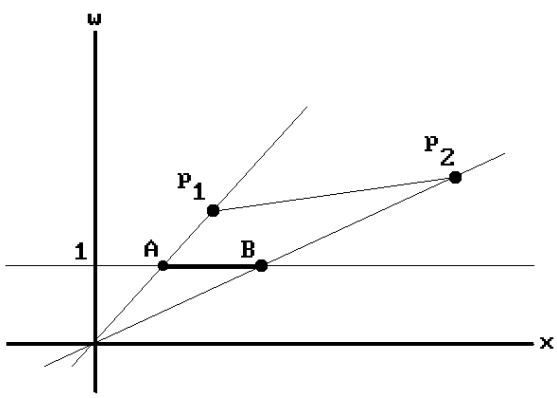


Figure 4.20. The w = 1 plane in $\mathbf{R}^4$.

points float around in $\mathbf{R}^4$ and only worry about projecting back to the "real" world at the end. There will be no problems as long as we deal with individual points. Problems can arise though as soon as we deal with nondiscrete sets.

In affine geometry, segments, for example, are completely determined by their endpoints and one can maintain complete information about a segment simply by keeping track of its endpoints. More generally, in affine geometry, the boundary of objects usually determines a well-defined "inside," and once we know what has happened to the boundary we know what happened to its "inside." A circle in the plane divides the plane into two parts, the "inside," which is the bounded part, and the "outside," which is the unbounded part. This is not the case in projective geometry, where it is not always clear what is "inside" or "outside" of a set. Analogies with the circle and sphere make this point clearer. Two points on a circle divide the circle into two curvilinear segments. Which is the "inside" of the two points? A circle divides a sphere into two curvilinear disks. Which is the "interior" of the circle?

Here is how one can get into trouble when one uses homogeneous coordinates with segments. Again, consider Figure 4.20 and the segment corresponding to the "real" points **A** and **B**. The figure shows that at least with some choices of representatives, namely, $\mathbf{p}_1$ and $\mathbf{p}_2$, nothing strange happens. The segment $[\mathbf{p}_1,\mathbf{p}_2]$ in $\mathbf{R}^4$ projects onto the segment $[\mathbf{A},\mathbf{B}]$ and so the points

$$s\mathbf{p}_1 + t\mathbf{p}_2, 0 \le s, t \le 1, s + t = 1,$$

represent the same points of $\mathbf{P}^3$ as the points of $[\mathbf{A},\mathbf{B}]$. It would appear as if one can deal with segments in projective space by simply using the ordinary 4-tuple segments in $\mathbf{R}^4$. But what if we used $\mathbf{p}_1' = a\mathbf{p}_1$ instead, where $a < 0$? See Figure 4.21. In that case, the segment $[\mathbf{p}_1',\mathbf{p}_2]$ projects to the exterior segment on **A** and **B** and so determines different points in $\mathbf{P}^3$ from $[\mathbf{A},\mathbf{B}]$. The only way to avoid this problem would be to ensure that the w-coordinate of all the points of our objects always stayed positive as they got mapped around. Unfortunately, this is not always feasible.

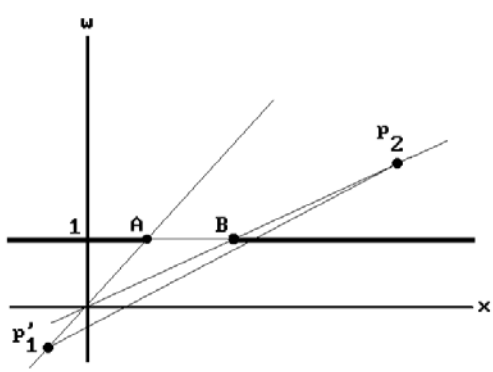


Figure 4.21. Problems with homogeneous representatives for points.

4.11 The Projections in OpenGL

In OpenGL one needs to specify the viewing volume. This is done in the way indicated in Figure 4.22. Note that there is no view plane as such and that the z-axis does not point in the direction one is looking but in the opposite direction. The view volume is specified by the far clipping plane $z = -f$ and a rectangle $[\ell,r] \times [b,t]$ in the near clipping plane $z = -n$.

The steps in OpenGL are

(1) Convert to camera coordinates.

(2) Map the homogeneous camera coordinates to homogeneous clip space, but this time one maps into the cube $[-1,1] \times [-1,1] \times [-1,1]$. The homogeneous matrix M that does this is defined by the equation

$$nM = \begin{pmatrix} \frac{2n}{r-\ell} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ \frac{r+\ell}{r-\ell} & \frac{t+b}{t-b} & -\frac{f+n}{f-n} & -1 \\ 0 & 0 & -\frac{2fn}{f-n} & 0 \end{pmatrix}.$$

The matrix M is obtained in the same manner as was $M_{hcam \to hclip}$ in Section 4.5. The call to the function glFrustum (ℓ,r,b,t,n,f) in OpenGL generates the matrix nM.

(3) Project to normalized device coordinates in Euclidean space (division by w).

(4) Transform to the viewport.

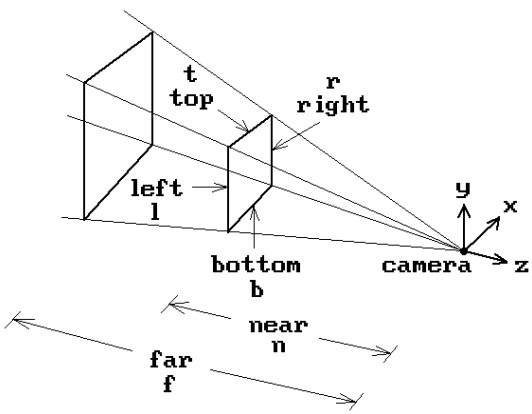


Figure 4.22. The OpenGL viewing volume.

4.12 Reconstruction

Ignoring clipping, which we shall in this section, by using homogeneous coordinates the mathematics in our discussion of the graphics pipeline basically reduced to an equation of the form

$$\mathbf{a}\mathrm{M} = \mathbf{b}, \tag{4.13}$$

where M was a 4×3 matrix, $\mathbf{a} \in \mathbf{R}^4$, and $\mathbf{b} \in \mathbf{R}^3$. The given quantities were the matrix M, computed from the given camera, and a point in the world that determined **a**. We then used equation (4.13) to compute **b** and the corresponding point in the view plane. Our goal here is to give a brief look at basic aspects of two types of inverse problems. For additional details see [PenP86]. For a much more thorough and mathematical discussion of this section's topic see [FauL01].

The Object Reconstruction Problem. Can one determine the point in the world knowing one or more points in the view plane to which it projected with respect to a given camera or cameras?

The Camera Calibration Problem. Can one determine the world-to-view-plane transformation if we know some world points and where they get mapped in the view plane?

Engineers have long used two-dimensional drawings of orthogonal projections of three-dimensional objects to describe these objects. The human brain is quite adept at doing this but the mathematics behind this or the more general problem of reconstructing objects from two-dimensional projections using arbitrary projective transformations is not at all easy. Lots of work has been done to come up with efficient solutions, even in what might seem like the simpler case of orthographic views. See, for example, [ShiS98]. Given three orthographic views of a point (x,y,z), say a front, side, and top view, one would get six constraints on the three values x, y, and z. Such overconstrained systems, where the values themselves might not be totally accurate in practice, are typical in reconstruction problems and the best that one can hope for is a best approximation to the answer.

Before describing solutions to our two reconstruction problems, we need to address a complication related to homogeneous coordinates. If we consider projective space as equivalence classes [**x**] of real tuples **x**, then mathematically we are really dealing with a map

$$\begin{aligned} \mathrm{T}\colon \mathbf{P}^3 &\to \mathbf{P}^2 \\ \mathbf{p} &\to \mathbf{q} = \mathrm{T}(\mathbf{p}) \end{aligned} \tag{4.14}$$

Equation (4.13) had simply replaced equation (4.14) with an equation of representatives **a**, M, and **b** for **p**, T, and **q**, respectively. The representatives are only unique up to scalar multiple. If we are given **p** and T and want to determine **q**, then we are free to choose any representatives for **p** and T. The problems in this section, however, involve solving for **p** given T and **b** or solving for T given **p** and **q**. In these cases, we

cannot assume that the representatives in equation (4.13) all have a certain form. We must allow for the scalar multiple in the choice of representatives at least to some degree. Fortunately, however, the equations can be rewritten in a more convenient form that eliminates any explicit reference to such scalar multiples. It turns out that we can always concentrate the scalar multiple in the "**b**" vector of equation (4.13). Therefore, rather than choosing the usual representative of the form $\mathbf{b} = (b_1,b_2,1)$ for $[\mathbf{b}]$, we can allow for scalar multiples by expressing the representative in the form

$$\mathbf{b} = (c \cdot b_1, c \cdot b_2, c). \tag{4.15}$$

Let $\mathbf{a} = (a_1,a_2,a_3,a_4)$ and $M = (m_{ij})$. Let $\mathbf{m}_j = (m_{1j},m_{2j},m_{3j},m_{4j})$, $j = 1,2,3$, be the column vectors of M. Equation (4.13) now becomes

$$(\mathbf{a} \bullet \mathbf{m}_1, \mathbf{a} \bullet \mathbf{m}_2, \mathbf{a} \bullet \mathbf{m}_3) = (c \cdot b_1, c \cdot b_2, c).$$

It follows that $c = \mathbf{a} \bullet \mathbf{m}_3$. Substituting for c and moving everything to the left, equation (4.13) can be replaced by the equations

$$\mathbf{a} \bullet (\mathbf{m}_1 - b_1\mathbf{m}_3) = 0 \tag{4.16a}$$

$$\mathbf{a} \bullet (\mathbf{m}_2 - b_2\mathbf{m}_3) = 0. \tag{4.16b}$$

It is this form of equation (4.13) that will be used in computations below. They have a scalar multiple for **b** built into them.

After these preliminaries, we proceed to a solution for the first of our two reconstruction problems. The object reconstruction problem is basically a question of whether equation (4.13) determines **a** if M and **b** are known. Obviously, a single point **b** is not enough because that would only determine a ray from the camera and provide no depth information. If we assume that we know the projection of a point with respect to two cameras, then we shall get two equations

$$\mathbf{a}M' = \mathbf{b}' \tag{4.17a}$$

$$\mathbf{a}M'' = \mathbf{b}''. \tag{4.17b}$$

At this point we run into the scalar multiple problem for homogeneous coordinates discussed above. In the present case we may assume that $M' = (m_{ij}')$ and $M'' = (m_{ij}'')$ are two fixed predetermined representatives for our projections and that we are looking for a normalized tuple $\mathbf{a} = (a_1,a_2,a_3,1)$ as long as we allow a scalar multiple ambiguity in $\mathbf{b}' = (c'b_1',c'b_2',c')$ and $\mathbf{b}'' = (c''b_1'',c''b_2'',c'')$. Expressing equations (4.17) in the form (4.16) leads, after some rewriting, to the matrix equation

$$(a_1\ a_2\ a_3)A = \mathbf{d}, \tag{4.18}$$

where

$$A = \begin{pmatrix} m_{11}' - b_1'm_{31}' & m_{21}' - b_2'm_{31}' & m_{11}'' - b_1''m_{31}'' & m_{21}'' - b_1''m_{31}'' \\ m_{12}' - b_1'm_{32}' & m_{22}' - b_2'm_{32}' & m_{12}'' - b_1''m_{32}'' & m_{22}'' - b_1''m_{32}'' \\ m_{13}' - b_1'm_{33}' & m_{23}' - b_2'm_{33}' & m_{13}'' - b_1''m_{33}'' & m_{23}'' - b_1''m_{33}'' \end{pmatrix}$$

and

$$\mathbf{d} = (b_1'm_{34}' - m_{14}' \quad b_2'm_{34}' - m_{24}' \quad b_1''m_{34}'' - m_{14}'' \quad b_2''m_{34}'' - m_{24}'').$$

This gives four equations in three unknowns. Such an overdetermined system does not have a solution in general; however, if we can ensure that the matrix A has rank three, then there is a least squares approximation solution

$$(a_1\ a_2\ a_3) = \mathbf{d}A^+ = \mathbf{d}A^T\left(AA^T\right)^{-1}. \tag{4.19}$$

using the generalized matrix inverse A^+ (see Theorem 1.11.6 in [AgoM05]).

Next, consider the camera calibration problem. Mathematically, the problem is to compute M if equation (4.13) holds for known points $\mathbf{a}_i$ and $\mathbf{b}_i$, i = 1, 2, . . . , k. This time around, we cannot normalize the $\mathbf{a}_i$ and shall assume that $\mathbf{a}_i = (a_{i1}, a_{i2}, a_{i3}, a_{i4})$ and $\mathbf{b}_i = (c_i b_{i1}, c_i b_{i2}, c_i)$. It is convenient to rewrite equations (4.16) in the form

$$\mathbf{m}_1 \bullet \mathbf{a}_i - \mathbf{m}_3 \bullet b_{i1}\mathbf{a}_i = 0 \tag{4.20a}$$

$$\mathbf{m}_2 \bullet \mathbf{a}_i - \mathbf{m}_3 \bullet b_{i2}\mathbf{a}_i = 0. \tag{4.20b}$$

We leave it as an exercise to show that equations (4.20) can be written in matrix form as

$$\mathbf{n}A = \mathbf{0},$$

where

$$A = \begin{pmatrix} \mathbf{a}_1^T & \mathbf{a}_2^T & \cdots & \mathbf{a}_k^T & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{a}_1^T & \mathbf{a}_2^T & \cdots & \mathbf{a}_k^T \\ -b_{11}\mathbf{a}_1^T & -b_{21}\mathbf{a}_1^T & \cdots & -b_{n1}\mathbf{a}_1^T & -b_{12}\mathbf{a}_1^T & -b_{22}\mathbf{a}_1^T & \cdots & -b_{n2}\mathbf{a}_n^T \end{pmatrix}$$

and

$$\mathbf{n} = (m_{11}\ m_{21}\ m_{31}\ m_{41}\ m_{12}\ m_{22}\ m_{32}\ m_{42}\ m_{13}\ m_{23}\ m_{33}\ m_{43}).$$

This overdetermined homogeneous system in twelve unknowns m_{ij} will again have a least squares approximation solution that can be found with the aid of a generalized inverse provided that $\mathbf{n}$ is not zero.

4.13 Robotics and Animation

This section is mainly intended as an example of frames and transformations and how the former can greatly facilitate the study of the latter, but it also enables us to give a brief introduction to the subject of the kinematics of robot arms. Even though we can only cover some very simple aspects of robotics here, we cover enough so that the reader will learn something about what is involved in animating figures.

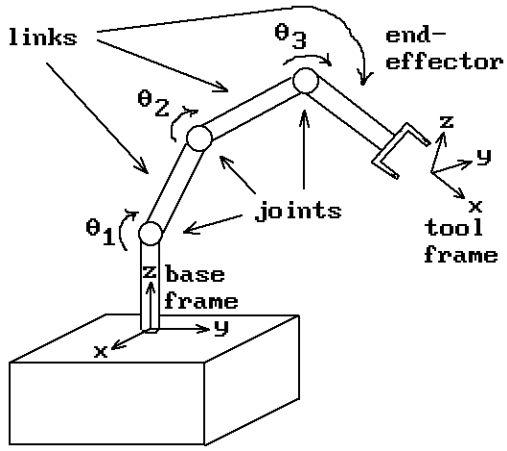


Figure 4.23. Robot arm terminology.

Mechanical manipulators are the most important examples of industrial robots and much work has been done to understand the mechanics and control of such manipulators. We begin with some terminology (see [Crai89]).

Kinematics: The science of motion where motion is studied without worrying about what caused it.

Manipulator: This is assumed to be an object that consists of nearly rigid *links* connected by *joints* that allow neighboring links to move. One end is usually fixed to some nonmoving part and the other end is free. See Figure 4.23. The joints may be either *revolute joints*, which allow rotational motion measured by *joint angles*, or *prismatic joints*, which allow sliding motion that is measured by *joint offsets*.

Degrees of freedom of a manipulator: This is the number of variables that it takes to completely describe the state or position of the manipulator. Typically this is the number of joints since joints can usually be described by one variable.

End-effector: This is the tool at the free end of the manipulator such as a gripper.

Tool frame: The frame associated to the end-effector.

Base frame: The frame associated to the fixed end of the manipulator.

Forward kinematics: This is the problem where one wants to compute the tool frame (intuitively, the position and orientation of the tool) relative to the base frame given the set of joint angles.

Inverse kinematics: This is the problem where one wants to compute all possible sets of joint angles that can give rise to given tool and base frames. This problem is usually more difficult than the forward kinematics problem. There may not even be a solution to a particular problem or there may be more than one solution.

Workspace for a given manipulator: The possible tool frames (position and orientation of the tool) that are achievable by the manipulator.

Trajectory generation: The determination of the trajectories of each joint of a manipulator that lead from some initial configuration to a final configuration. Since manipulators are usually moved by actuators that apply a force or torque to each joint, these forces and torques would also have to be computed in order for a solution to be effective.

For all practical purposes, links will be treated as rigid bodies connecting joints. Links will be numbered starting with link number 0 at the base. Joints will also be numbered with joint number i being the joint between link number i – 1 and i. The ith joint can be defined by an axis $\mathbf{L}_i$ about which the joint rotates or along which it slides. Define the *link length* a_i to be the distance between the axes $\mathbf{L}_i$ and $\mathbf{L}_{i+1}$ and let $\mathbf{p}_i$ be the point on $\mathbf{L}_i$ and $\mathbf{q}_{i+1}$ the point on $\mathbf{L}_{i+1}$ so that $a_i = |\mathbf{p}_i\mathbf{q}_{i+1}|$. The points $\mathbf{p}_i$ and $\mathbf{q}_i$ are always unique except in the case where axes are parallel. Since the choice of these points is only an issue in defining the **initial** configuration of a manipulator, we can either disallow this case or assume that they have been picked appropriately in some other manner. We assume that all the a_i are nonzero. Next associate a *link frame* $F_i = (\mathbf{x}_i,\mathbf{y}_i,\mathbf{z}_i,\mathbf{p}_i)$ to the ith link as follows:

(1) $\mathbf{z}_i$ is a direction vector for $\mathbf{L}_i$, that is, it is parallel to the ith joint axis,
(2) $\mathbf{x}_i$ is the unit vector in the direction $\mathbf{p}_i\mathbf{q}_{i+1}$, that is, $\mathbf{x}_i = (1/a_i)\ \mathbf{p}_i\mathbf{q}_{i+1}$, and
(3) $\mathbf{y}_i = \mathbf{z}_i \times \mathbf{x}_i$, so that the ordered basis $(\mathbf{x}_i,\mathbf{y}_i,\mathbf{z}_i)$ determines the standard orientation of $\mathbf{R}^3$.

Define the *link twist* α_i to be the angle between $\mathbf{z}_i$ and $\mathbf{z}_{i+1}$ and choose the sign of the angle to be positive or negative depending on whether or not the ordered bases $(\mathbf{z}_i,\mathbf{z}_{i+1})$ and $(\mathbf{y}_i,\mathbf{z}_i)$ determine the same orientation in the $\mathbf{y}_i$-$\mathbf{z}_i$ plane of F_i (this corresponds to using the right hand rule for angles about the $\mathbf{x}_i$-axis). Define the *joint offset* d_{i+1} by $\mathbf{q}_{i+1}\mathbf{p}_{i+1} = d_{i+1}\mathbf{z}_{i+1}$. Note that $|d_{i+1}|$ is the distance between $\mathbf{q}_{i+1}$ and $\mathbf{p}_{i+1}$. d_{i+1} is a variable if the joint is prismatic. Finally, define the *joint angle* θ_{i+1} to be the signed angle between $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ about $\mathbf{z}_{i+1}$. This is a variable if the joint is revolute. The quantities a_i, α_i, d_i, and θ_i are called the *link parameters*. All, except a_i, are signed numbers. See Figure 4.24.

There are some special cases at the ends of the manipulator. Assume that there are n + 1 links. It is convenient to choose F_0 to be the same as F_1 when θ_1 is zero. At the other end there are two cases. If the last joint is revolute, then $\mathbf{x}_n$ is set equal to $\mathbf{x}_{n-1}$ when θ_n is zero. If the joint is prismatic, then $\mathbf{x}_n$ is set equal to $\mathbf{x}_{n-1}$. The point $\mathbf{p}_n$ is always set equal to $\mathbf{q}_n$.

The robot or manipulator is completely described in one of two ways. We can either specify the link frames F_i or we can specify the list of link parameters a_i, α_i, d_i,

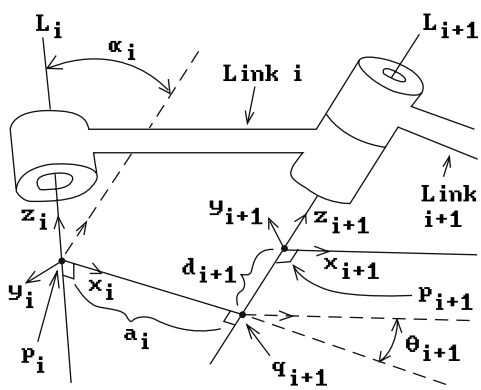


Figure 4.24. The geometry of links.

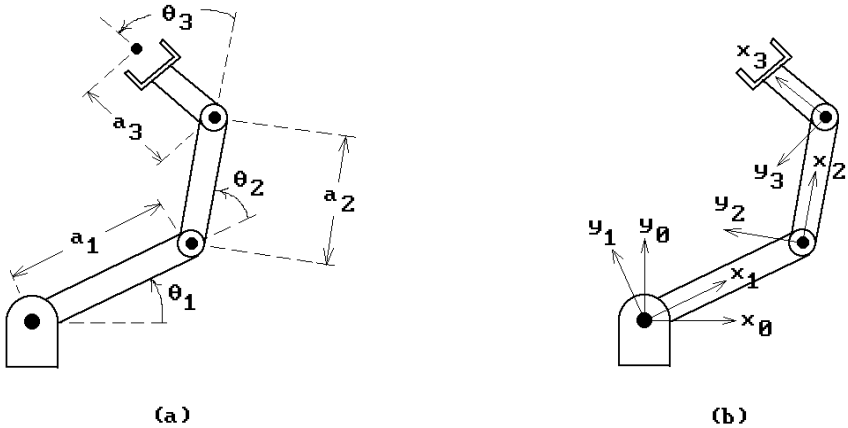


Figure 4.25. Two-dimensional robot arm geometry.

and θ_i. Our discussion above basically showed how the latter are derived from the former. One can also show that the link parameters completely define the frames. In practice it is easier for a user to manipulate the link parameters and so the usual problem is to find the frames given their values. As another example, consider a two-dimensional robot with three links and revolute joints. We can think of this as a special case of the general one where all the z-axes of the frames point in the same direction and all the a_i and d_i are zero. Figure 4.25(a) shows the link parameters and Figure 4.25(b), the associated frames.

As one can see, frames play a role in defining the state of a robot, but how are they used to solve problems? Well, the forward kinematic problem is to find the tool frame ("where the tool is") given the link parameters. This problem will be solved if we can determine the transformation T_n, which, given the coordinates of a point $\mathbf{p}$ in F_n coordinates, finds the coordinates of $\mathbf{p}$ with respect to F_0. Let dT_i, $0 < i \leq n$, denote the transformation that maps coordinates relative to F_i to coordinates relative to F_{i-1}. It follows that

$$T_n = dT_1 \circ dT_2 \circ \cdots \circ dT_n. \tag{4.21}$$

The dT_i are relatively easy to compute from the link parameters because they are the composition of four simple maps.

The Computation of dT_i and Its Homogeneous Matrix dM_i. Let $T_i(\mathbf{z}_i, d_i)$ denote the translation with translation vector $d_i\mathbf{z}_i$. Its homogeneous matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & d_i & 1 \end{pmatrix}$$

Let $R_i(\mathbf{z}_i,\theta_i)$ denote the rotation about the $\mathbf{z}_i$ axis of the frame F_i through an angle θ_i. Its homogeneous matrix is

$$\begin{pmatrix} \cos\theta_i & \sin\theta_i & 0 & 0 \\ -\sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $T_{i-1}(\mathbf{x}_{i-1},a_{i-1})$ denote the translation with translation vector $a_{i-1}\mathbf{x}_{i-1}$. Its homogeneous matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_{i-1} & 0 & 0 & 1 \end{pmatrix}$$

Finally, let $R_{i-1}(\mathbf{x}_{i-1},\alpha_{i-1})$ denote the rotation about the $\mathbf{x}_{i-1}$ axis of the frame F_{i-1} through an angle α_{i-1}. Its homogeneous matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_{i-1} & \sin\alpha_{i-1} & 0 \\ 0 & -\sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then

$$dT_i = R_{i-1}(\mathbf{x}_{i-1}, \alpha_{i-1}) \circ T_{i-1}(\mathbf{x}_{i-1}, a_{i-1}) \circ R_i(\mathbf{z}_i, \theta_i) \circ T_i(\mathbf{z}_i, d_i), \tag{4.22}$$

and multiplying the matrices for the corresponding maps together (but in reverse order since matrices act on the right of a point), we get that the matrix dM_i associated to the transformation dT_i is defined by

$$dM_i = \begin{pmatrix} \cos\theta_i & \sin\theta_i \cos\alpha_{i-1} & \sin\theta_i \sin\alpha_{i-1} & 0 \\ -\sin\theta_i & \cos\theta_i \cos\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & 0 \\ 0 & -\sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ a_{i-1} & -d_i \sin\alpha_{i-1} & d_i \cos\alpha_{i-1} & 1 \end{pmatrix} \tag{4.23}$$

Equations (4.21–4.23) provide the solution to the forward kinematic problem. In the two-dimensional robot case where α_i and d_i are zero, the matrices dM_i specialize to matrices dN_i, where

$$dN_i = \begin{pmatrix} \cos\theta_i & \sin\theta_i & 0 & 0 \\ -\sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_{i-1} & 0 & 0 & 1 \end{pmatrix} \tag{4.24}$$

This concludes what we have to say about robotics. For a more in-depth study of the subject see [Crai89], [Feat87], or [Paul82]. The description of mechanical manipulators in terms of the four link parameters described in this section is usually called the *Denavit-Hartenberg notation* (see [DenH55]).

As is often the case when one is learning something new, a real understanding of the issues does not come until one has worked out some concrete examples. The animation programming projects 4.13.1 and 4.13.2 should help in that regard. Here are a few comments about how one animates objects. Recall the discussion in Section 2.11 for the 2d case. To show motion one again discretizes time and shows a sequence of still pictures of the world as it would look at increasing times $t_1, t_2, \ldots, t_n$. One changes the apparent speed of the motion by changing the size of the time intervals between t_i and t_{i+1}, the larger the intervals, the larger the apparent speed. Therefore, to move an object **X**, one begins by showing the world with the object at its initial position and then loops through reshowing the world, each time placing the object in its next position.

4.14 Quaternions and In-betweening

This short section describes another aspect of how transformations get used in animation. In particular, we discuss a nice application of quaternions. Unit quaternions are a more efficient way to represent a rotation of $\mathbf{R}^3$ than the standard matrix representation of **SO**(3). Chapter 20 provides the mathematical foundation for quaternions. Other references for the topic of this section are [WatW92], [Hogg92], and [Shoe93].

We recall some basic facts about the correspondence between rotations about the origin in $\mathbf{R}^3$ and unit quaternions. First of all, the quaternion algebra **H** is just $\mathbf{R}^4$ endowed with the quaternion product. The metric on **H** is the same as that on $\mathbf{R}^4$. The standard basis vectors $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, $\mathbf{e}_4$ are now denoted by **1**, **i**, **j**, **k**, respectively. The subspace generated by **i**, **j**, and **k** is identified with $\mathbf{R}^3$ by mapping the quaternion a**i**+b**j**+c**k** to (a,b,c) and vice versa. The rotation R of $\mathbf{R}^3$ through angle θ about the directed line through the origin with unit direction vector **n** is mapped to the quaternion **q** defined by

$$\mathbf{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\mathbf{n} \in \mathbf{H}. \tag{4.25}$$

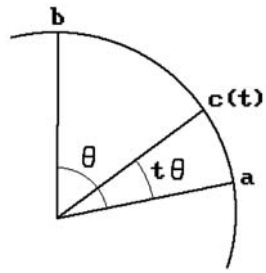
Conversely, let $\mathbf{q} = r + a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$ be a unit quaternion ($\mathbf{q} \in \mathbf{S}^3$) and express **q** in the form

$$\mathbf{q} = \cos\theta + \sin\theta\mathbf{n}, \tag{4.26}$$

where **n** is a unit vector of $\mathbf{R}^3$. If $M_{\mathbf{q}}$ is the matrix defined by

$$M_{\mathbf{q}} = \begin{pmatrix} 1-2b^2-2c^2 & 2rc+2ab & 2ac-2rb \\ 2ab-2rc & 1-2c^2-2a^2 & 2ra+2bc \\ 2rb+2ac & 2bc-2ra & 1-2a^2-2b^2 \end{pmatrix}, \tag{4.27}$$

Figure 4.26. Interpolating between two quaternions.



then $M_{\mathbf{q}} \in \mathbf{SO}(3)$ and the map of $\mathbf{R}^3$ that sends $\mathbf{p}$ to $\mathbf{p}M_{\mathbf{q}}$ is a rotation $R_{\mathbf{q}}$ about the line through the origin with direction vector $\mathbf{n}$ through the angle 2θ. This mapping

$$\begin{array}{ccc} \text{unit quaternions} & \to & \text{rotations of } \mathbf{R}^3 \text{ about the origin} \\ \mathbf{q} & \to & R_{\mathbf{q}} \end{array}$$

has the property that $R_{\mathbf{q}} = R_{-\mathbf{q}}$.

Now, suppose an object is moved by a one-parameter family of matrices $M(s) \in \mathbf{SO}(3)$. Assume that we have only specified this family at a fixed set of values s_i. How can we interpolate between these values? In animation such an interpolation is called *in-betweening*. A simple interpolation of the form

$$tM(s_i) + (1-t)M(s_{i+1})$$

would not work because the interpolants would not again be elements of $\mathbf{SO}(3)$. One could try to use Euler angles, but there are problems with this also. See [Shoe85]. A better way is to translate our maps into quaternions and to look for a one-parameter family of unit quaternions $q(s)$ that interpolates between two quaternions $\mathbf{a}$ and $\mathbf{b}$. However, a simple linear interpolation followed by a normalization to get unit quaternions does not work well either for the same reason that one does not get a uniform subdivision of an arc of a circle by centrally projecting a uniform subdivision of the chord connecting its endpoints. What would happen in the animation is that the object would move faster in the middle of the interpolation. A better solution is to subdivide the arc of the great circle in $\mathbf{S}^3$ connecting $\mathbf{a}$ and $\mathbf{b}$. See Figure 4.26.

4.14.1 Lemma. Let $\mathbf{a}$ and $\mathbf{b}$ be two unit quaternions that make an angle of $\theta \neq 0$ with each other, that is, $\mathbf{a} \cdot \mathbf{b} = \cos\theta$ and $0 < \theta < \pi$. Then the unit quaternion $\mathbf{c}(t)$ that lies in the plane spanned by $\mathbf{a}$ and $\mathbf{b}$ and which makes an angle $t\theta$ with $\mathbf{a}$, $0 \le t \le 1$ is defined by the equation

$$\mathbf{c}(t) = \frac{\sin(1-t)\theta}{\sin\theta}\mathbf{a} + \frac{\sin t\theta}{\sin\theta}\mathbf{b}. \tag{4.28}$$

Proof. By hypothesis,

$$\mathbf{c}(t) = r\mathbf{a} + s\mathbf{b}$$

for some real numbers r and s. Taking the dot product of both sides of this equation with **a** and **b** leads to equations

$$\cos t\theta = \mathbf{c}(t) \bullet \mathbf{a} = r + s\mathbf{b} \bullet \mathbf{a} \tag{4.29}$$

and

$$\cos(1-t)\theta = \mathbf{c}(t) \bullet \mathbf{b} = r\mathbf{a} \bullet \mathbf{b} + s. \tag{4.30}$$

Solving equations (4.29) and (4.30) for r and s and using some standard trigonometric identities leads to the stated result.

4.14.2 Example. Let $\mathbf{L}_0$ and $\mathbf{L}_1$ be the positively directed x- and y-axis, respectively. Let R_0 and R_1 be the rotations about the directed lines $\mathbf{L}_0$ and $\mathbf{L}_1$, respectively, through angle $\pi/3$ and let M_0 and M_1 be the matrices that represent them. If M_t, $t \in [0,1]$, is the 1-parameter family of in-betweening matrices in $\mathbf{SO}(3)$ between M_0 and M_1, then what is $M_{1/2}$?

Solution. The unit direction vectors for $\mathbf{L}_0$ and $\mathbf{L}_1$ are $\mathbf{n}_0 = (1,0,0)$ and $\mathbf{n}_1 = (0,1,0)$, respectively. Therefore, by equation (4.25)

$$\mathbf{q}_0 = \cos\frac{\pi}{6} + \sin\frac{\pi}{6}\mathbf{n}_0 = \frac{\sqrt{3}}{2} + \frac{1}{2}(1,0,0)$$

and

$$\mathbf{q}_1 = \cos\frac{\pi}{6} + \sin\frac{\pi}{6}\mathbf{n}_1 = \frac{\sqrt{3}}{2} + \frac{1}{2}(1,0,0)$$

are the unit quaternions corresponding to rotation R_0 and R_1. The angle θ between $\mathbf{q}_0$ and $\mathbf{q}_1$ is defined by

$$\cos\theta = \mathbf{q}_0 \bullet \mathbf{q}_1 = \frac{3}{4}.$$

It follows that

$$\sin\theta = \sqrt{1-\cos^2\theta} = \frac{\sqrt{7}}{4} \quad \text{and} \quad \sin\frac{\theta}{2} = \sqrt{\frac{1-\cos\theta}{2}} = \frac{1}{2\sqrt{2}}.$$

Using equation (4.28), let

$$\mathbf{q}_{1/2} = \frac{2}{\sqrt{14}}(\mathbf{q}_0 + \mathbf{q}_1) = 2\sqrt{\frac{3}{14}} + \left(\frac{1}{\sqrt{14}}, \frac{1}{\sqrt{14}}, 0\right) = 2\sqrt{\frac{3}{14}} + \frac{1}{\sqrt{7}}\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right).$$

Finally, equation (4.27) implies that

$$M_{1/2} = \begin{pmatrix} \frac{6}{7} & \frac{1}{7} & -\frac{2\sqrt{3}}{7} \\ \frac{1}{7} & \frac{6}{7} & \frac{2\sqrt{3}}{7} \\ \frac{2\sqrt{3}}{7} & -\frac{2\sqrt{3}}{7} & \frac{5}{7} \end{pmatrix}.$$

We can also see from the expression for $\mathbf{q}_{1/2}$ and equation (4.26) that $M_{1/2}$ defines a rotation about the directed line through the origin with direction vector (1,1,0) and angle 2α, where

$$\cos\alpha = 2\sqrt{\frac{3}{14}}.$$

It is clear that the quaternions $\mathbf{c}(t)$ defined by Lemma 4.14.1 do indeed lie on a great circles in the unit sphere of the quaternions. We have solved the uniform spacing problem, but unfortunately this is not the end of the story as far as animation is concerned. Two points on a circle divide the circle into two arcs. If the points are not antipodal points, then one typically is interested in the smaller arc. In our situation we cannot simply always take the smaller arc without further consideration because we are representing rotations by quaternions, and if $\mathbf{q}$ is a unit quaternion, both $\mathbf{q}$ and $-\mathbf{q}$ correspond to the same rotation in $\mathbf{SO}(3)$. The solution suggested in [WatW92] is, given representations $\mathbf{a}$ and $\mathbf{b}$ for two rotations, to choose between $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{a}$, $-\mathbf{b}$. One compares the distance between $\mathbf{a}$ and $\mathbf{b}$, $|\mathbf{a} - \mathbf{b}|$, to the distance between $\mathbf{a}$ and $-\mathbf{b}$, $|\mathbf{a} + \mathbf{b}|$. If the former is smaller use $\mathbf{a}$, $\mathbf{b}$ otherwise use $\mathbf{a}$, $-\mathbf{b}$.

After getting our uniformly spaced quaternions $\mathbf{c}(t_i)$ along the arc, if we were to do a linear interpolation between them, then the motion may look jerky. It is better to smooth things out by using Bezier curves or, more generally, splines, but this is somewhat more complicated in quaternion space than it was in $\mathbf{R}^n$. See [BCGH92], [WatW92], [Hogg92], or [Shoe93] for what needs to be done.

4.15 Conclusions

Transformations were at the center of all the discussions in this chapter. We would like to emphasize one last time that when it comes to representing and defining affine transformations one should do that via frames if at all possible. Frames are orthonormal basis and these are easy to define. Without them, geometric modeling for n-dimensional objects would become very complicated when n is larger than 2. Once one has a frame it can be interpreted as a transformation, a coordinate system, or as defining a change of coordinates. See Chapter 2 in [AgoM05] for more details.

The main role of homogeneous coordinates and projective space is as the natural setting for projective transformations. The mathematics becomes much easier. A practical application is that one can replace all transformations, including translations, with matrices in uniform way. We described some of the main perspective and parallel projections.

The discussion in this chapter emphasized a mathematical view of transformations. Let us add a few comments about efficiency because there are times when efficiency is more important than clarity. Transformations get used a lot in geometric modeling and generating them in a systematic way, which usually involves representing them as a composite of primitive ones, involves more arithmetic operations than necessary. The papers [Gold90] and [Mill99] describe a more efficient approach. For example, suppose we express an arbitrary affine transformation T of $\mathbf{R}^3$ in the form

$$T(\mathbf{p}) = M\mathbf{p}^T + \mathbf{v},$$

where M is a 3×3 matrix and $\mathbf{v}$ is a fixed translation vector. If T is a rotation through an angle θ about a line $\mathbf{L}$ through a point $\mathbf{q}$ with direction vector $\mathbf{w}$, then it is shown that

$$M = \cos\theta I + (1 - \cos\theta)\mathbf{w} \otimes \mathbf{w} + \sin\theta A_w \quad \text{and} \tag{4.31a}$$

$$\mathbf{v} = \mathbf{q} - \mathbf{M}\mathbf{q}^T, \tag{4.31b}$$

where I is the 3×3 identity matrix,

$$\mathbf{a} \otimes \mathbf{b} = \begin{pmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} (b_1\ b_2\ b_3),$$

and

$$A_w = \begin{pmatrix} 0 & w_3 & -w_2 \\ -w_3 & 0 & w_1 \\ w_2 & -w_1 & 0 \end{pmatrix}.$$

With this representation, an optimized computation takes 15 multiplications and 10 additions to compute M and 9 multiplications and 9 additions to compute $\mathbf{v}$. The number of operations to compute the matrix M and vector $\mathbf{v}$ with the "composite-of-transformations" approach would be much greater. See [Gold90] for efficient formulas for other transformations.

Finally, one topic not discussed much in this book (other than in the documentation for the GM and SPACE program) is the user interface of a modeling program, even though this takes up a large percentage of the time needed to develop such a program in general. Certain aspects of such a discussion, if we had the space or time, would probably be most appropriate in the context of the topic of this chapter. There are lots of interesting issues here as to how one can make it easier for the user to define the desired view of the world. How to best look at the world is often a major concern to a user. How does a user specify a view? How does one control panning and zooming? How does one specify a user's or the camera's arbitrary movement through the world? In a three-dimensional world this is not always easy if all one has is a keyboard and a mouse.

4.16 EXERCISES

Section 4.2

4.2.1 Prove equation (4.3).

4.2.2 Find the world-to-camera coordinates transformation T in two dimensions given the following data: the camera is at (1,4) looking in direction (–1,–3) with the view "plane" (a line) a distance $\sqrt{10}$ in front of the camera.

4.2.3 Find the world-to-camera coordinates transformation T in three dimensions given that the camera is at (2,3,5) looking in direction (–1,2,1) with the view plane a distance 7 in front of the camera.

Section 4.3

4.3.1 A camera is situated at the origin looking in direction **v**. Find the vanishing points of the view defined by lines parallel to the standard unit cube when

(a) $\mathbf{v} = (2,0,3)$
(b) $\mathbf{v} = (0,3,0)$
(c) $\mathbf{v} = (3,1,2)$

Section 4.5

4.5.1 With regard to Figure 4.11 show that the regions below are mapped as indicated:

$$\begin{array}{lcl} -\infty < z/w < -1 & \rightarrow & 1 < z/w < +\infty \\ -1 < z/w < 0 & \rightarrow & -\infty < z/w < 0 \\ 0 < z/w < +\infty & \rightarrow & 0 < z/w < +1 \end{array}$$

Note that z/w denotes the "real" z coordinate of a projective point (x,y,z,w).

4.5.2 Assume that the near and far planes for a camera are z = 2 and z = 51, respectively, in camera coordinates. If the view plane is z = 5, find the matrix $M_{hcam \rightarrow hclip}$.

Section 4.7

4.7.1 Explain how the case (0,0,0,0) can occur.

Section 4.9

4.9.1 Prove Proposition 4.9.1.

4.9.2 Compute the parallel projection of $\mathbf{R}^3$ onto the x-y plane in the direction $\mathbf{v} = (2,1,5)$.

4.9.3 Compute the parallel projection of $\mathbf{R}^3$ onto the plane $x - 2y + 3z = 1$ in the direction $\mathbf{v} = (2,1,-3)$.

Section 4.14

4.14.1 Let $\mathbf{L}_0$ and $\mathbf{L}_1$ be the directed lines through the origin in $\mathbf{R}^3$ with direction vectors (1,–1,0) and (1,1,0), respectively. Let R_0 and R_1 be the rotations about the directed lines $\mathbf{L}_0$ and $\mathbf{L}_1$, respectively, through angle $\pi/3$ and let M_0 and M_1 be the matrices that represent them. If M_t, $t \in [0,1]$, is the 1-parameter family of in-betweening matrices in $\mathbf{SO}(3)$ between M_0 and M_1, find $M_{1/2}$?

Section 4.15

4.15.1 Show that equations (4.31) compute the rotation T correctly.

4.17 Programming Projects

Section 4.5

4.5.1 A simpler graphics pipeline

The approach to clipping we described in Section 4.5 is more general than may be needed in some simple situations. This project describes a quick way to deal with clipping if one already has a two-dimensional clipper implemented. The basic idea is to do the z-clip in three dimensions but then to project to the view plane and do the rest of the clipping with the old 2-dimensional clipper. Since the far clipping plane is usually not very important, we shall ignore it and only worry about the near clipping plane. Then a simple graphics pipeline for an object that is either a point or a segment would go as follows:

(1) Transform any vertex (x,y,z) of the object in shape coordinates to camera coordinates (x′,y′,z′).

(2) Clip the resulting point or segment against the near clip plane.

(3) Project any vertex (x′,y′,z′) of the remaining object to the point (x″,y″) in the view plane using the equations

$$x'' = d\frac{x'}{z'} \quad \text{and} \quad y'' = d\frac{y'}{z'},$$

where d is the distance that the view plane is in front of the camera.

Clipping against the near clip plane is essential for two reasons. One is that we do not want points from behind the camera to project to the front. The second is that we want to prevent any division by zero. The clipping itself is very easy here. Assume that the near clipping plane is defined in the camera coordinate system by the equation $z = d_n$. Mathematically, clipping a single point simply amounts to checking if $z' < d_n$ or not. In the case of a segment, we have to find the intersection of that segment with the plane.

Consider a world of rectangular blocks for this program. Each block is defined by three real parameters a, b, and c and a frame f. The basic block is situated at the origin with faces parallel to the coordinate planes and vertices (0,0,0), (a,0,0), (a,0,c), (0,0,c), (0,b,0), (a,b,0), (a,b,c), and (0,b,c) and all other blocks are copies of this block moved by the frame f.

Section 4.7

4.7.1 The full-fledged clipping pipeline

Implement the graphics pipeline described in steps (1)–(6) in Section 4.7. Use the same world of blocks as in project 4.5.1.

Section 4.9

4.9.1 Implement stereo views of some simple objects like a cube.

Section 4.12

4.12.1 Implement the object reconstruction algorithm as described in Section 4.12 in a working graphics program like GM and check its accuracy. Let a user input a three-dimensional point $\mathbf{p}$, compute its projection $\mathbf{q}$ with respect to a known camera, and compare the value of $\mathbf{p}$ with the one computed by the reconstruction algorithm from $\mathbf{q}$ and the camera data.

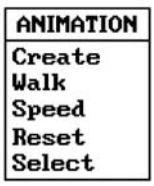
4.12.2 Implement the camera calibration algorithm as described in Section 4.12 in a working graphics program like GM and check its accuracy. Let a user input three-dimensional points $\mathbf{p}_i$, compute their projections $\mathbf{q}_i$ with respect to a known camera, and compare the given camera data with the one computed by the camera calibration algorithm from the $\mathbf{p}_i$ and $\mathbf{q}_i$ data.

Section 4.13

4.13.1 A walking robot

This project applies some of the mathematical ideas behind the forward kinematics solution for robots to a three-dimensional animation problem. The idea is to animate a simple robot figure built out of blocks walking on the xy-plane. See Figure 4.27. The parts of the robot are a head, torso, two arms, and two legs. The robotics comes into the picture when moving the arms and legs. Each arm and leg consists of three links that are constrained to planar motion. Figure 4.28 shows a two-dimensional projection of a leg. The base frame of the figure is that of the torso. The head is rigidly attached to the torso.

The following menu should be associated to this project



and the items should perform the following tasks when activated:

Create: Ask the user for a size and then create a new robot of that given size. At any given time there is a *currently active* robot (the one to which the "Walk," "Speed," and "Reset" commands apply).

Walk: Ask the user for a number of steps, then walk the robot that many steps in the direction it is facing with its joints moving in a way so that the walk is reasonably close to a human's walk.

Speed: Ask the user for a new speed that is a function of the distance between positions at which the robot is displayed.

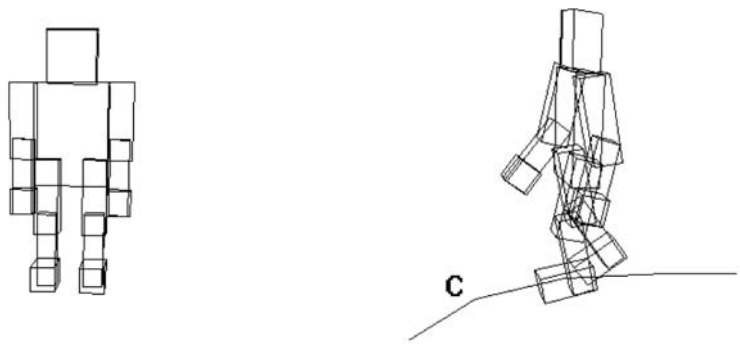
Reset: Resets the robot to its initial configuration.

Select: Let the user specify another robot to be the currently active one.

4.13.2 Robot walking along path

Extend project 4.13.1 by associating a path to each robot along which it walks. For example, the robot in Figure 4.27(b) is walking along path **C**. The default path for a robot (if the user did not explicitly define one) is a straight line in the direction the robot is pointing.

The menu should now look like



(a) (b)

Figure 4.27. Robots in motion.

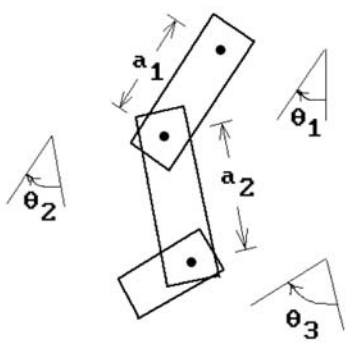


Figure 4.28. Two-dimensional link parameters.

ANIMATION
Create
Walk
Speed
Path
Reset
Select

Activating the Path item should allow the user to define a new path in the xy-plane by selecting points that correspond to the vertices of a polygonal curve. After the new path has been defined the robot should be placed at the beginning of the path looking in the direction defined by the first edge.

Walking along paths entails some slight changes to the "Walk" and "Reset" operations.

Walk: Now when the robot moves along points on its path it should always be looking in the direction that is tangent to the curve. In those few instances where the robot lands on a vertex, assume that the tangent at the vertex is the average of the tangent vectors for the segments meeting in that vertex.

Reset: If the robot has a path associated to it, place it at the beginning of the path looking in the direction defined by the first edge. If there is no path, reset the robot to some initial configuration as before.

CHAPTER 5

Approaches to Geometric Modeling

Prerequisites: Section 4.2 (topology of $\mathbf{R}^n$) and Chapter 7 (cell complexes, Euler characteristic) in [AgoM05]

5.1 Introduction

The last four chapters covered the basic mathematics and computer graphics algorithms needed to display two- or three-dimensional points and segments. As limited as this may sound, this is actually enough to develop a quite decent modeling system that would handle complex linear three-dimensional objects as long as we represent them only in terms of their edges ("wireframe" mode). Such a system might be adequate in many situations. On the other hand, one would certainly not get any eye-catching displays in this way. To generate such displays, we need to represent objects more completely. Their surfaces, not just their edges, must be represented. After that, there is the problem of determining which parts of a surface are visible and finally the problem of how to shade those visible parts.

Recall the general geometric modeling pipeline shown in Figure 5.1. Of interest are the last three boxes and maps between them. This chapter presents a survey of the various approaches that have been used to deal with that part of the pipeline. First, one has to understand the "pure" mathematical objects and maps. The next task is to represent these in a finite way so that a computer can handle them. Finally, the finite (discrete) representations have to be implemented with specific data structures. By in large, users of current CAD systems did not require the systems to have much understanding of the "geometry" of the objects. That is not to say that no fancy mathematics is involved. The popular spline curves and surfaces involve very intricate mathematics, but the emphasis is on "local" properties. So far, there has not been any real need for understanding global and intrinsic properties, the kind studied in topology for example.

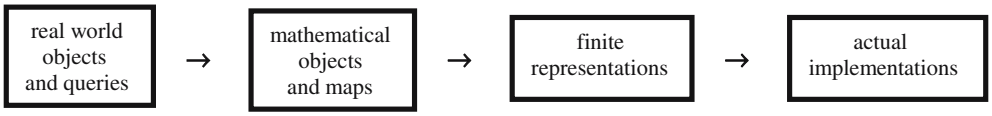


Figure 5.1. The real world to implementation pipeline.

Some texts and papers use the term "solid modeling" in the context of representing objects in 3-space. Since this term connotes the study of homogeneous spaces (n-manifolds), we prefer to use the term "geometric modeling" and use it to refer to modeling **any** geometric object. Three-dimensional objects may be the ones of most interest usually, but we do not always want to restrict ourselves to those.

The first steps to develop a theoretical foundation for the field of geometric modeling were taken in the Production Automation Project at the University of Rochester in the early 1970s. The notion of an r-set and that of a representation scheme were introduced there. These concepts, along with the creation of the constructive solid geometry (CSG) modeler PADL-1 and the emphasis on the **validity** of representations, had a great influence on the subsequent developments in geometric modeling. R-sets were thought of as the natural mathematical equivalent of what one would refer to as a "solid" in everyday conversation. Using r-sets one could define the domain of coverage of a representation more carefully than before. The relevance of topology to geometric modeling was demonstrated. The terms "r-set" and "representation scheme" are now part of the standard terminology used in discussions about geometric modeling. Most of this chapter is spent on describing various approaches to and issues in geometric modeling within the context of that framework.

Section 5.2 defines r-sets and related set operations. Section 5.3 defines and discusses what is called a representation scheme. The definitions in these two sections are at the core of the theoretical foundation developed at the University of Rochester. After some observations about early representation schemes in Section 5.3.1, Sections 5.3.2–5.3.9 describe the major representation schemes for solids in more or less historical order, with emphasis on the more popular ones. The two most well-known representation schemes, the boundary and CSG representations, are discussed first. After that we describe the Euler operations representation, generative modeling and the sweep representations, representations of solids via parameterizations, representations based on decomposition into primitives, volume modeling, and the medial axis representation. Next, in Section 5.4, we touch briefly on the large subject of representations for natural phenomena. Section 5.5 is on the increasingly active subject of physically based modeling, which deals with incorporating forces acting on objects into a modeling system. Feature-based modeling, an attempt to make modeling easier for designers, is described in Section 5.6. Having surveyed the various ways to represent objects, we discuss, in Section 5.7, how functions and algorithms fit into the theory. Section 5.8 looks at the problem of choosing appropriate data structures for the objects in geometric modeling programs. Section 5.9 looks at the important problem of converting from one scheme to another. Section 5.10 looks at the ever-present danger of round-off errors and their effect on the robustness of programs. Section 5.11 takes a stab at trying to unify some of the different approaches to geometric modeling. We describe what is meant by algorithmic modeling and discuss

what computability might mean in the continuous rather than discrete setting. Finally, Section 5.12 finishes the chapter with some comments on the status and inadequacies in the current state of geometric modeling.

5.2 R-Sets and Regularized Set Operators

One of the terms that is used a lot in geometric modeling is the term "solid." What does it mean? It should be very general and include all the obvious objects. In particular, one would want it to include at the very least all linear polyhedral "solids." One also wants the set of solids to be closed under the natural set operations such as union, intersection, and difference.

Intuitively, a solid is something that is truly three-dimensional and also **homogeneous** in the sense that, if we take a solid like the unit cube and stick a (one-dimensional) segment onto it forming a set such as

$$\mathbf{X} = [0,1] \times [0,1] \times [0,1] \cup [(1,1,1),(2,2,2)], \tag{5.1}$$

which is shown in Figure 5.2, then we do **not** want to call $\mathbf{X}$ a solid. A definition of a solid needs to exclude the existence of such lower-dimensional parts.

Definition. Let $\mathbf{X} \subseteq \mathbf{R}^n$. Define the *regularization operator* r and the *regularization* of $\mathbf{X}$, r$\mathbf{X}$, by

$$\mathrm{r}\mathbf{X} = \mathrm{cl}\,(\mathrm{int}(\mathbf{X})).$$

The set $\mathbf{X}$ is called a *regular set* or an *r-set* (in $\mathbf{R}^n$) if $\mathbf{X}$ = r$\mathbf{X}$, that is, the set is the closure of its interior.

Note that the definitions depend on the dimension n of the Euclidean space under consideration because the interior of a set does. For example, the unit square is an r-set in $\mathbf{R}^2$ but not in $\mathbf{R}^3$ (Exercise 5.2.1). Note also that the set $\mathbf{X}$ in equation (5.1) is not an r-set because

$$\mathrm{cl}\,(\mathrm{int}(\mathbf{X})) = [0,1] \times [0,1] \times [0,1] \neq \mathbf{X}.$$

One can also show that

$$\mathrm{r}(\mathrm{r}\mathbf{X}) = \mathrm{r}\mathbf{X} \tag{5.2}$$

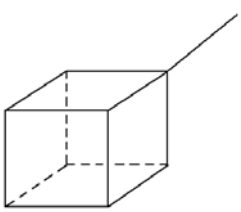


Figure 5.2. A nonsolid.

(Exercise 5.2.2). In other words, r**X** is an r-set for any subset **X** of $\mathbf{R}^n$. R-sets seem to capture the notion of being a solid. Anything called a **solid** should be an r-set, but we shall refrain from giving a formal definition of the word "solid." In many situations, one would probably want that to mean a compact (closed and bounded) n-manifold. R-sets are more general than manifolds, however. The union of two tetrahedra which meet in a vertex is an r-set but not a 3-manifold because the vertex where they meet does not have a Euclidean neighborhood.

Because halfplanes are r-sets we get all our linear polyhedral "solids" from those via the Boolean set operators such as union, intersection, and difference. We can think of halfspaces as primitive building blocks for r-sets if we allow "curved halfspaces" by extending the notion as follows:

Definition. A *halfspace* in $\mathbf{R}^n$ is any set of the form

$$H_+(f) = \{\mathbf{p} \mid f(\mathbf{p}) \geq 0\} \quad \text{or} \quad H_-(f) = \{\mathbf{p} \mid f(\mathbf{p}) \leq 0\},$$

where $f : \mathbf{R}^n \to \mathbf{R}$. If **H** is a halfspace, then we shall call r**H** a *generic halfspace*. A finite combination of generic halfspaces using the standard operations of union, intersections, difference, and complement is called a *semialgebraic* or *semianalytic set* if the functions f are all polynomials or analytic functions, respectively.

For example, the infinite (solid) cylinder of radius R about the z-axis, that is,

$$\{(x, y, z) \mid x^2 + y^2 - R^2 \leq 0\},$$

is a generic halfspace, in fact, a semialgebraic set. See Figure 5.3. Semialgebraic sets are an adequate set of building blocks for most geometric modeling and are also "computable" (see Section 5.11).

Next, we need to address a problem with the standard Boolean set operators, namely, they are not closed when restricted to r-sets. For example, the intersection of the two r-sets $\mathbf{X} = [0,1] \times [0,1]$ and $\mathbf{Y} = [1,2] \times [0,1]$ is not an r-set. See Figure 5.4. From the point of view of solids, we would like to consider **X** and **Y** as being disjoint. One sometimes calls **X** and **Y** *quasi-disjoint*, which means that their intersection is a lower-dimensional set. If we want closure under set operations, we need to revise their definitions.

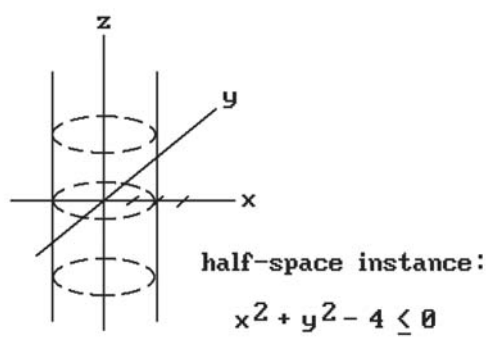


Figure 5.3. A generic halfspace.

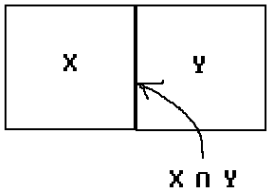


Figure 5.4. Quasi-disjoint sets.

Definition. Define the *regularized set operators* $\cup^*$, $\cap^*$, $-^*$, c*, and Δ^* by

$$\begin{aligned} \mathbf{X} \cup^* \mathbf{Y} &= \mathrm{r}(\mathbf{X} \cup \mathbf{Y}), \\ \mathbf{X} \cap^* \mathbf{Y} &= \mathrm{r}(\mathbf{X} \cap \mathbf{Y}), \\ \mathbf{X} -^* \mathbf{Y} &= \mathrm{r}(\mathbf{X} - \mathbf{Y}), \\ \mathrm{c}^* \mathbf{Y} &= \mathrm{r}(\mathrm{c}\mathbf{Y}), \quad \text{and} \\ \mathbf{X} \Delta^* \mathbf{Y} &= (\mathbf{X} -^* \mathbf{Y}) \cup^* (\mathbf{Y} -^* \mathbf{X}), \end{aligned}$$

where c and Δ are the complement and symmetric difference operators, respectively.

5.2.1 Theorem

(1) The regularized set operators take r-sets into r-sets. Furthermore, there are algorithms that perform these operations.
(2) The class of regular semialgebraic or semianalytic sets is closed under regularized set operations.

Proof. For (1) see [Tilo80] or [Mort85]. For (2) see [Hiro74].

Even though r-sets are quite general, they have their limitations.

(1) Although they have attractive features from a mathematical point of view, they are complicated to deal with computationally.
(2) One may want to deal with nonsolids like in Figure 5.2. This is not possible with r-sets.

Nevertheless, at least one has something mathematically precise on which to base proofs.

5.3 Representation Schemes

Geometric modeling systems have taken many different approaches to representing geometric objects. The following definitions ([ReqV82]) can be thought of as a start towards being able to evaluate and judge these approaches in a rigorous way.

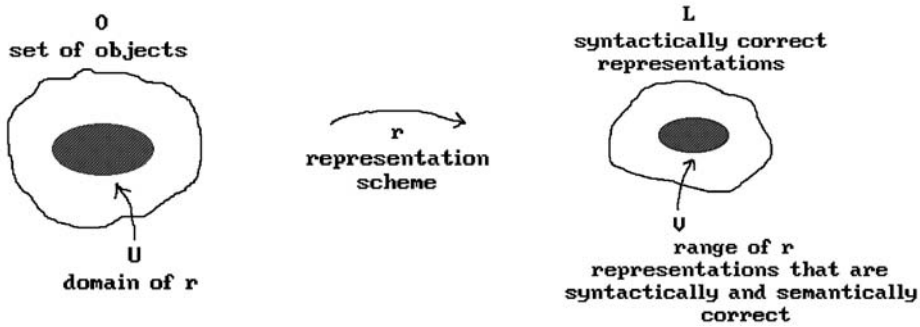


Figure 5.5. Representation scheme.

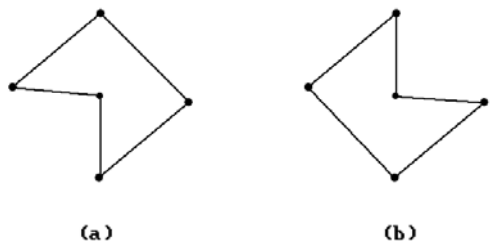


Figure 5.6. Example of ambiguous representation scheme.

Definition. A *representation scheme*, or simply *representation*, of a set of "objects" **O** using a set **L** is a relation r between **O** and **L**. If (x,y) ∈ r, then we shall say that y *represents* x. A representation scheme r is *unambiguous* (or *complete*) if r is one-to-one. A representation scheme r is *unique* if r is a function (that is, single-valued). The elements of **L** are called *representations* or *syntactically correct representations* and those in the range of r are called the *semantically correct* or *valid representations*.

See Figure 5.5. The term "syntactically/semantically correct" is used, because if r is a representation scheme, we can think of r(x) as a set of encodings for x in a "language" **L**. The semantically correct elements of **L** are those "sentences" which have a "meaning" in that there is an object that corresponds to them. The terms unambiguous and unique separate out those relations that are **not** many-to-one or one-to-many, respectively. To be unambiguous means that if one has the encoding, then one knows the object to which it corresponds. To be unique means that there is only one way to encode an object.

5.3.1 Example. Let **O** be the set of polygons in the plane that have positive area but no holes. Let **L** be the set of finite sequences of points. For example, the sequence (2,1), (–1,3), (4,5) belongs to **L**. Define a representation scheme for **O** using **L** by associating to each object in **O** the set of its vertices listed in some order. This representation scheme is neither unambiguous nor unique. It is ambiguous because the objects in Figures 5.6(a) and (b) both have the same vertices. It is not unique because the vertices of an object can be listed in many ways. Furthermore, not all sequences of points

are semantically correct. A sequence of collinear points does not correspond to a polygon in **O**.

We could modify Example 5.3.1. For example, we could require the polygons to be convex or we could require that the vertices be listed in counter-clockwise order. In both instances we would then have an unambiguous representation scheme.

There are reasons for why unambiguousness and uniqueness are important properties of a representation scheme. It is difficult to compute properties from ambiguous schemes. For example, it would be impossible to compute the area of a polygon with the ambiguous scheme in Example 5.3.1. An example of why uniqueness is important is when one wants to determine if two objects are the same. The ability to test for equality is important because one needs it for

(1) detecting duplication in data base of objects
(2) detecting loops in algorithms, and
(3) verifying results such as in case of numerically controlled (NC) machines where it is important that the desired object is created

With uniqueness one merely needs to compare items syntactically. Note that the problem of determining whether two sets are the same can be reduced to a problem of whether a certain other set is empty, because two sets **X** and **Y** are the same if and only if the regularized symmetric difference $\mathbf{X}\Delta^*\mathbf{Y}$ is empty.

Although unambiguousness and uniqueness are highly desirable, such representations are hardly ever found. Two common types of nonuniqueness are

(1) permutational (as in the example where sequences of points represent a polygon) and
(2) positional (where different representations exist due to primitives that differ only by a rigid motion).

Eliminating these types of nonuniqueness would involve a high computational expense.

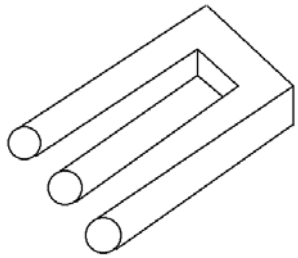
The domain of a representation scheme specifies the objects that the scheme is capable of representing. One clearly wants this to be as large as possible. In particular, one would want it to include at the very least all linear polyhedral "solids." One also wants the domain to be closed under some natural set operations such as union, intersection, and difference. This raises some technical issues.

One issue that has become very important in the context of representation schemes is **validity**.

> **The Basic Validity Problem for a Representation Scheme:** When does a representation correspond to a "real" object, that is, when is a syntactically correct representation semantically correct or valid?

Ideally, every syntactically correct representation should be semantically correct because syntactical correctness is usually much easier to check than semantic correctness. Certainly, a geometric database should not contain representations of nonsense objects. The object in Figure 5.7 could easily be described in terms of surface

Figure 5.7. A nonsense object.



patches and so its definition would seem correct from a **local** point of view, but taken in its entirety it clearly does not correspond to a real object. In early geometric modeling systems, validity of a representation was the responsibility of the user, but this has changed. It is no longer acceptable to assume human intervention to correct errors. For one thing, a modeling system might have to feed its geometric data directly to another system such as a robot and bad data might crash that system.

Here are some other informal properties of representation schemes:

(1) Robustness and numeric precision (see Section 5.10 for a discussion of this topic)
(2) Compactness (for storing): "Verbose" representations may contain redundancies that would make verifying validity harder. On the other hand, in the usual trade-off, this may improve performance.
(3) Computational ease and applicability: No representation is best for everything. To support a variety of applications, we could have multiple representations for each object, but then one must maintain consistency.
(4) Ability to convert between different representation schemes: One may want to pass data between different modelers, but even a single modeler may contain more than one representation scheme.

Along with a formalization of the objects that constitute the domain of a modeler, one should also specify and formalize the allowable operations and functions. This formalization has only been carried out in a minimal way so far. We postpone this largely ad hoc discussion to Section 5.7. Insofar as the usual definition of the term "representation scheme" does not address operations and functions, it is an incomplete concept. The term "object representation scheme" would have been more appropriate because it is more accurate.

Representation schemes coupled with the user interface of a modeler have a great influence on the way that a user will think about objects or shapes. One needs to distinguish between a *machine representation* and a *user representation*. The discussion above has concentrated on the former, which may or may not be visible to the user, but the latter is also very important and deals with the user interface. A driving force behind generative modeling, which will be described in Section 5.3.5, had to do with giving a modeler a desirable user representation. The issues involved with user representations are similar to but not the same as those for machine representations. Some important informal questions that a **user** representation must address are

(1) To what class of shapes is a user restricted?
(2) How does a user describe and edit the possible shapes and how easy is this?

    (a) How shapes are described can easily limit the user's ability to use good designs and even to think up a good design in the first place.
    (b) How much input is required for each shape?
    (c) Can a user easily predict the output from the input?
    (d) How accurate are the representations with respect to what the user wants?
    (e) Are the operations that a user can perform on shapes closed in the sense that the output to an operation can be the input to another?

(3) How fast and how realistically can the shapes be generated on a display?
(4) What operations can a user perform on shapes and how fast can they be carried out?

Of course, the type of user representation that one wants depends on the user. Here we have in mind a more technical type of user. Later in Section 5.6 we consider a user in the context of a manufacturing environment.

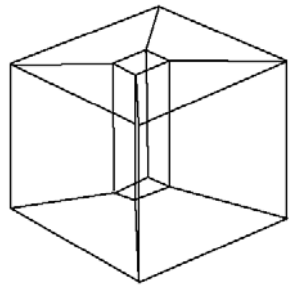
5.3.1 Early Representation Schemes

Approaches to geometric modeling have changed over the years. These changes began before computers existed and all one had was pencil and paper. Since the advent of computers, these changes were largely influenced by their power, the essential mathematics behind the changes being basically not new. As computers become more and more powerful, it gradually becomes possible to implement mathematical representations that mathematicians have used in their studies. The history of the development of geometric modeling shows this trend. Of course, the new ways of interactively visualizing data that was not possible before will undoubtedly cause its own share of advances in knowledge. We shall comment more on this at the end of this chapter.

Engineering Drawings. Engineering drawings were the earliest attempts to model objects. Computers were not involved and they were intended as a means of communication among **humans**. They often had errors but humans were able to use common sense to end up with correct result. There was no formal definition of such drawings as a representation scheme. The basic idea was to represent objects by a collection of planar projections. As such it is a highly ambiguous representation scheme because if one were to try to implement it on a computer, it is very difficult to determine how many two-dimensional projections would be needed to completely represent a three-dimensional object. Constructing an object from some two-dimensional projections of it is a highly interesting and difficult problem. We touched on two small aspects of this problem in Section 4.12. For more, see [RogA90], [BoeP94], [PenP86], or [Egga98].

Wireframe Representations. Wireframe representations were the first representation schemes for three-dimensional linear polyhedra. It is a natural approach, the idea

Figure 5.8. An ambiguous wireframe representation.



being to represent them using only their edges. After all, edges are some of the most important features of an object that one "sees." Unfortunately, this representation scheme is ambiguous. For example, Figure 5.8 shows a block with a beveled hole through its center. It is not possible to tell along which axis the hole lies from the edge information alone. Two problems caused by the ambiguity are that one cannot remove hidden lines reliably and one cannot produce sections automatically.

Many early commercially available modeling systems used wireframe representations. Even now many systems support a wireframe display mode because it is fast and adequate for some jobs. A *wireframe display* is one where only edges and no faces are shown. Note that how objects are displayed is quite independent of how they are represented internally.

Faceted Representations. A simple solution that eliminates the major wireframe representation problems for three-dimensional objects is to add faces. This representation is unambiguous. We shall look at this approach in more detail later in the section on the boundary representation. Again, there is a difference between a modeling system using a faceted **representation** and one using a *faceted display*. The latter means that objects (of **all** dimensions) are displayed as if they were linear polyhedra even though the system may maintain an exact analytic representation of objects internally. For example, a sphere centered at the origin is completely described by one number, its radius, but it might be displayed in a faceted manner.

Attempts have been made to develop algorithms that generate faces from a wireframe representation automatically, but it is known that only using topological information leads to an NP-complete problem, so that the algorithms will not be very efficient in general. See, for example, [BagW95].

Primitive Instancing Schemes. In this scheme we simply have a finite number of generic parameterized primitives that can be represented via tuples of the form

(type code, parameter 1 , . . . , parameter k)

where the parameters are either reals or integers. See Figure 5.9. We do not need all dimensions as parameters, only those that are variable. The representation is unambiguous and may be unique. It is certainly very compact. With regard to algorithms for computing properties of objects represented by such scheme, one basically needs a special case for each primitive.

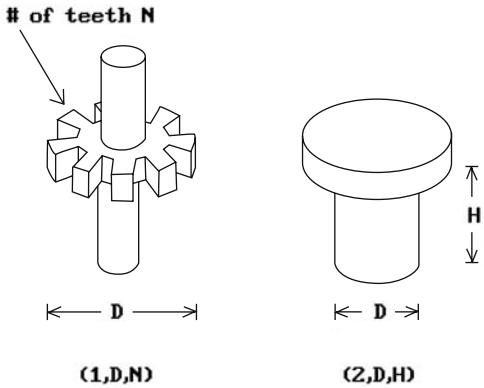


Figure 5.9. Primitive instancing scheme.

A primitive instancing scheme is only really useful when the number of primitives is small. The scheme is similar to a language in which words cannot be combined to make a sentence. Note that a good structured modeling system can achieve all the advantages of a primitive instancing scheme by providing a macro capability.

5.3.2 Boundary Representations

Since one only sees the surface of solids, it is natural to represent them via their boundary. Mathematically this is justified because in the special case of closed and bounded solids in $\mathbf{R}^n$ the boundary of a solid uniquely defines that solid. (In general, though, the boundary of a manifold does **not** define the manifold.)

Definition. The *boundary representation* or *b-rep* of r-sets in $\mathbf{R}^n$ is the representation that associates to each r-set $\mathbf{X}$ its boundary b$\mathbf{X}$. More generally, a b-rep for r-sets is a representation that associates to an r-set $\mathbf{X}$ a representation of b$\mathbf{X}$.

B-reps are probably the most common representations used in computer graphics. If $\mathbf{X}$ is a solid in $\mathbf{R}^3$, then

$$b\mathbf{X} = \bigcup_{\text{"face" } \mathbf{F} \text{ of } \mathbf{X}} \mathbf{F}.$$

The boundary of each face is itself a union of edges and the boundary of an edge is a set of two points. One can therefore think of a boundary representation scheme for solids in terms of a relation between objects and certain graphs. See Figure 5.10 for the most basic version of such a graph. For linear polyhedra it is actually a function. For curved objects it is not a function because it depends on how the boundary of the object is divided into cells (curved patches).

There are some difficult questions having to do with the validity of boundary representations. In particular,

When does a graph of the type shown in Figure 5.10 come from a real polyhedron?

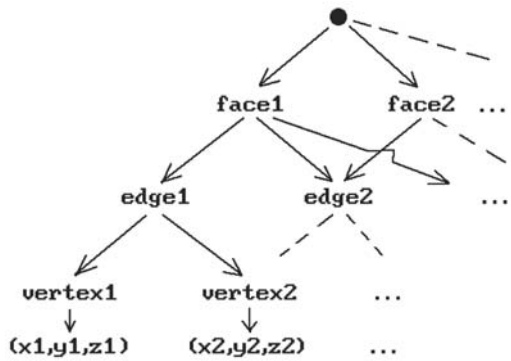


Figure 5.10. A boundary representation graph.

There is no easy answer to this question. Here are some conditions in case the b-rep for a solid is supposed to be induced from a simplicial decomposition, that is, the solid is the underlying space of a simplicial complex:

(1) Each face must have three edges.
(2) Each edge must have two vertices.
(3) Each edge must belong to an even number of faces.
(4) Each vertex of a face must belong to precisely two edges of the face.
(5) All points (x,y,z) must be distinct.
(6) Edges must be disjoint or intersect in a vertex.
(7) Faces must be disjoint or intersect in edges.

Conditions (1)–(4) deal with the combinatorial topology of simplicial complexes and are easy to check. Conditions (5)–(7) are point set topology questions that are expensive to test.

Some common data structures that are used to implement the boundaries of linear polyhedra are described in Section 5.8.1.

5.3.3 The CSG Representation

In constructive solid geometry (CSG) one represents objects in terms of a sequence of Boolean set operations and rigid motion operators starting with a given collection of primitive objects. One can express this representation pictorially as a binary tree. For example, in Figure 5.11 the binary tree on the left is used to represent the union of three blocks, one of which has been translated by a vector **v**. Although the idea is simple enough, we must get a little technical in order to give a precise definition.

Let P be a set of r-sets in $\mathbf{R}^n$. The elements of P will be called primitive objects. Let O be a set of regularized binary set operators such as $\cup^*$, $\cap^*$, $-^*$, ... Let M be a set of pairs (m,x) where m is a label for a rigid motion of $\mathbf{R}^n$ such as a translation, rotation, . . . , and x is data that defines a specific such motion. For example, if n = 2, then (rotation,($\mathbf{p}$,π/2)) is a possible pair in M and represents the rotation of $\mathbf{R}^2$ about $\mathbf{p}$ through an angle $\pi/2$. If (m,x) $\in$ M, then let m(x) denote the rigid motion defined by the pair (m,x).

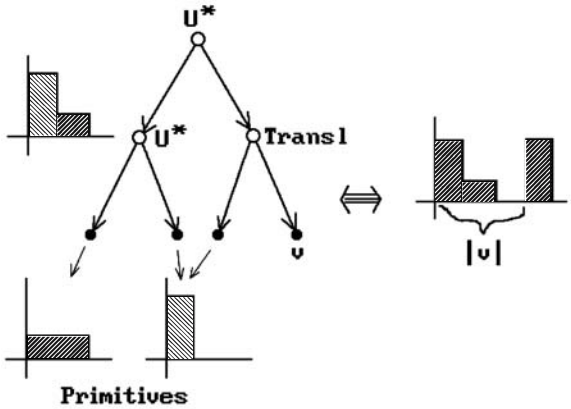


Figure 5.11. A CSG example.

Definition. A *CSG-tree* for the tuple (P,O,M) is an expression T defined recursively in BNF (Backus-Naur Form) notation by

$$\mathbf{T} ::= p \mid ToT \mid Tmx,$$

where $p \in P$, $o \in O$, $(m,x) \in M$.

An informal definition of a CSG-tree is

$$\langle \text{CSG-tree} \rangle ::= \langle \text{primitive leaf} \rangle \mid \langle \text{CSG-tree} \rangle \langle \text{set operation node} \rangle \langle \text{CSG-tree} \rangle$$
$$\mid \langle \text{CSG-tree} \rangle \langle \text{motion m} \rangle \langle \text{motion args} \rangle$$

It is clear that a CSG-tree can be identified with a binary tree like the one shown in Figure 5.11 and that is what is usually done in practice. Some typical values that have been used for P, O, and M are

$$P = \{\text{block, cylinder, disk (“solid” sphere)}\}$$
$$O = \{\cup^*, \cap^*, -^*\}$$
$$M = \{(m, x) \mid m(x) \text{ is translation or rotation}\}$$

Definition. The *realization* of a CSG-tree T is the space |T| defined recursively as follows:

If $T = p$, $p \in P$, then $|T| = p$.

If $T = T_1 o T_2$, where T_1 and T_2 are CSG-trees and $o \in O$, then $|T| = |T_1| \, o \, |T_2|$.

If $T = T_1 mx$, where T_1 is a CSG-tree and $(m, x) \in M$, then $|T| = m(x)(|T_1|)$.

Definition. The *CSG representation* or *csg-rep* for a tuple (P,O,M) is the representation of r-sets using CSG-trees which consists of pairs (**X**,T) where **X** is an r-set, T is a CSG-tree for (P,O,M), and **X** = |T|.

Although one is free to choose any set of primitives or transformations for a csg-rep, generic halfspaces of one sort or another are usually used as primitives. Two common csg-reps used

(1) "arbitrary" (possibly unbounded) generic halfspaces as primitives, or
(2) bounded generic halfspace combinations as primitives.

Primitives are often parameterized. For example, a primitive block is usually considered to be situated at the origin and to be defined by the three parameters of length, width, and height. One then talks about *instancing* a primitive, where that term means

(1) assigning values to the configuration parameters, and then
(2) positioning the result of (1) via a rigid motion (which could also be viewed as assigning values to positional parameters).

Csg-reps can handle nonmanifold objects. Their exact domain of coverage depends on

(1) the primitives (actually the halfspaces which define them),
(2) the motion operators that are available, and
(3) the set operators that are available.

It is interesting to note the results of an extensive survey of mechanical parts and what it takes to describe them which can be found in [SaRE76]. Fully 63% of all the parts could be handled with a CSG system based on only orthogonal block and cylinder primitives. A larger class of primitives provided a natural description of over 90% of the parts. This indicated that CSG is therefore a good fit for a CAD system in that sort of environment because most mechanical parts seemed to be relatively simple.

If one uses general operations and bounded primitives, then one gets a representation that is

(1) unambiguous,
(2) not unique,
(3) very concise, and
(4) easy to create (at least for its domain of coverage).

One of the biggest advantages of a csg-rep over other representation schemes is that validity is pretty much built into the representation if one is a little careful about choosing primitives. For example, if one uses r-sets as primitives and arbitrary regularized set operations, then the algebraic properties of r-sets ensure that a representation is always valid. This is not the case if operations are not general, for example, if the union operation is only allowed for quasi-disjoint objects. Also, in a CSG system based on general generic halfspaces, some trees may represent unbounded sets and hence not be valid. It is true however that, by in large, all syntactically correct CSG representations (trees) are also semantically correct.

Because of the tree structure of a CSG representation, one can often use a divide-and-conquer approach to algorithms: one first solves a problem for the primitive

```
answer function Solve (BTree        T)
begin
    answer ans1, ans2;
    operation op;

    if  IsPrimitive (T)
        then  return (PrimitiveAnswerOf (T));
        else
            begin
                ans1: = Solve (LeftSubtree (T));
                ans2: = Solve (RightSubtree (T));
                op   : = ValueAtRoot (T);
                return (CombineAnswers (ans1,op,ans2));
            end
end;
```

Algorithm 5.3.3.1. A divide-and-conquer approach in CSG.

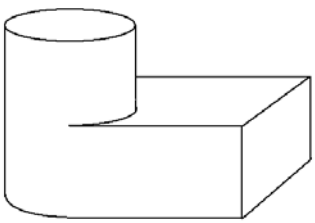


Figure 5.12. What are the faces of this solid?

objects and then uses recursion. See Algorithm 5.3.3.1. The point membership classification function, which is discussed in Section 5.9, is an example of this.

One disadvantage with a CSG representation is that it is not easy to generate a display using it because one needs the boundary of an object for that. Getting a boundary representation for an object defined by CSG (a process referred to as *boundary evaluation*) is relatively hard. We look at this in more detail in Section 5.9. One problem in this context (especially for mechanical design) is how to define a "face" of an object. This certainly is no problem in the linear case, but it is for curved objects. See Figure 5.12. What should the faces be in that figure? Some minimal characteristics of a face are:

(1) A face should be contained in the boundary of the solid.
(2) Topologically, a face should be a surface with boundary.
(3) If the solid was defined via regularized Boolean set operations from a collection of halfspaces $\{\mathbf{H}_i\}$, then each face should be contained in $b\mathbf{H}_i$ for some i.
(4) Faces should be quasi-disjoint, that is, pairwise intersections of faces should either be empty or lie in their boundary.

Another issue when it comes to faces is how to represent them? We shall see in Section 5.9 that to represent a face **F** we can

(1) represent the halfspace in whose boundary the face **F** lies (for example, in the case of a cylinder, use its equation),
(2) represent the boundary edges of **F** (the boundary of a face is a list of edges), and
(3) maintain some neighborhood information for these bounding edges and orient the edges (for example, we can arrange it so that the inside of the face is to the right of the edge or we can store appropriate normal vectors for the edges).

This scheme works pretty well for simple surfaces but for more complicated surfaces one needs more.

5.3.4 Euler Operations

Representation schemes based on using Euler operations to build objects are an attempt to have a boundary representation meet at least part of the validity issue head on. The idea is to permit only boundary representations that have a valid Euler characteristic. If we only allow operations that preserve the Euler characteristic or that change it in a well-defined way (such operations are called *Euler operations*), then we achieve this. Of course this is only a part of what is involved for an object not to be a nonsense object. Nevertheless we have at least preserved the combinatorial validity since the Euler characteristic is a basic invariant of combinatorial topology. As for metric validity, one still must do a careful analysis of face/face intersections. In any case, to say that a modeler is built on Euler operations means that it represents objects as a sequence of Euler operations.

Topologically, Euler operations are based on elementary collapses and expansions and/or cutting and pasting (see Sections 7.2.4 and 6.4 in [AgoM05], respectively). Figure 5.13 shows two elementary collapse and expansion examples. One says that the space **Y** consisting of the two segments on the right of Figure 5.13(a) is obtained from the solid triangle **X** on the left via an elementary collapse of the cell **c** from the edge **e**. Conversely, the space **X** is said to be obtained from **Y** via an elementary expansion. Figure 5.13(b) shows another elementary collapse and expansion, this time involving a three-dimensional cell **ABCD** and a face **ABC**. Figure 5.14 shows a cutting

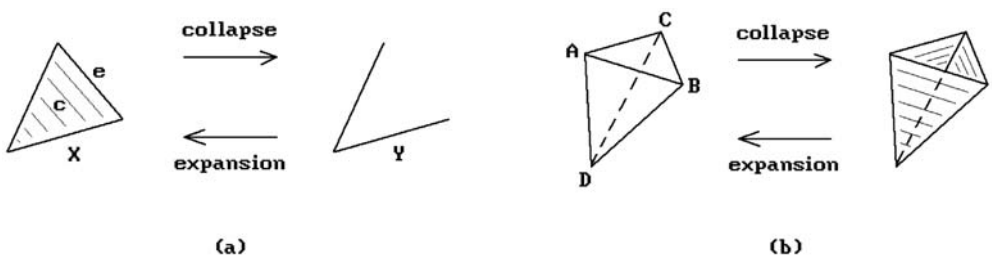


Figure 5.13. Elementary collapses/expansions.

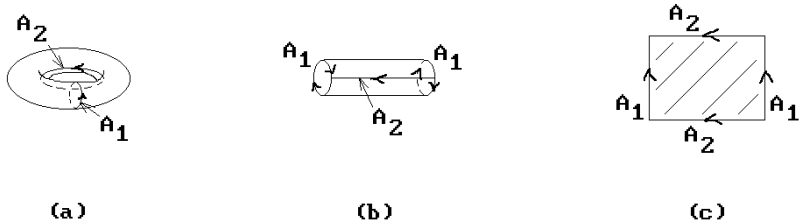


Figure 5.14. Cutting and pasting.

and pasting example. Specifically, we show how to cut the torus to get a rectangle and how, looking at it backward, we can get the torus from the rectangle by pasting appropriate edges together.

Elementary collapses or expansions do not change the Euler characteristic of a space. On the other hand, cutting and pasting operations usually do change the Euler characteristic. It turns out that these four operations do an excellent job to completely describe and define surfaces. (In higher dimensions things get more complicated.) Every surface, and hence solid in 3-space, can be obtained from a point by a sequence of elementary expansions, cuts, and pastes. Modelers based on Euler operations use a boundary representation for solids and simply define procedures that mimic the collapses, expansions, cutting, and pasting operations just described by modifying the cell structure of this boundary representation in a well-defined way.

Definition. The *Euler operation representation* of polyhedra is defined by the collection of pairs $(\mathbf{X},(\sigma_1,\sigma_2, \ldots , \sigma_k))$, where $\mathbf{X}$ is a polyhedra and $\sigma_1,\sigma_2, \ldots , \sigma_k$ is a sequence of Euler operations that produces $\partial\mathbf{X}$ starting with the empty set.

Euler operations were first introduced by Baumgart in his thesis and then used in his computer vision program GEOMED ([Baum75]). Braid, Hillyard, and Stroud ([BrHS80]) showed that only five operators are needed to describe the boundary surfaces of three-dimensional solids. Such a surface satisfies the Euler equation

$$V - E + F = 2(S - H),$$

where

V = the number of vertices,
E = the number of edges,
F = the number of faces,
S = the number of solid components, and
H = the number of holes in the solid.

They used a set of these Euler operations in their BUILD modeling system. Although one can make other choices for the five primitive operators, it seems that the boundary representation part of modelers built on Euler operations tend to use either

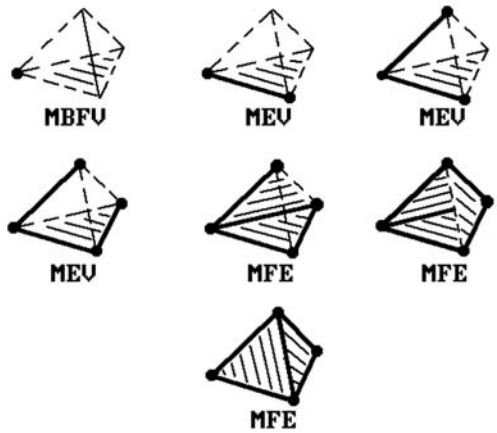


Figure 5.15. Building a tetrahedron with Euler operations.

Baumgart's winged edge representation (see Section 5.8.1) or some variant of it, so that this is what these operators modify.

Historically, Euler operators were given cryptic mnemonic names consisting of letters. A few of these are shown below along with their meanings:

M – make	K – kill	L - loop	
V – vertex	E – edge	F – face	B/S – body/solid

Using that notation, three typical operators were:

MEV – make ede and vertex
MFE – make face and edge
MBFV – make body, face, and vertex

Figure 5.15 shows how one could create a solid tetrahedron using these operators. The operators create the appropriate new data structure consisting of vertices, edges, faces, and solids and merge it into the current data structure. Along with each Euler operator that creates vertices, edges, or faces, there are operators that delete or kill them. This enables one to easily undo operations, a very desirable feature for a modeler.

There are good references for implementing modelers based on Euler operations. One is the book by Mäntylä ([Mant88]), which describes a modeling program GWB (the Geometric WorkBench). Another is the book by Chiyokura ([Chiy88]), which describes the modeling program DESIGNBASE. Euler operations were originally defined only for polyhedra but were extended to curved surfaces by Chiyokura.

To summarize, modelers based on Euler operations are really "ordinary" b-rep modelers except that the objects and boundary representations that can be built are constrained by the particular Euler operators that were chosen, so that they at least have combinatorial validity. The Euler operators are flexible enough though so that

the modelers share all the advantages (and some of the disadvantages) that one gets with a boundary representation.

We need to leave the reader, at least those who might be interested in modeling objects in higher dimensions than three, with one word of caution however. The result about five operators sufficing to construct objects raises some subtle issues. It applies only to the two-dimensional boundaries of solids and not to cell structures of solids. The fact is that not all n-cells, $n > 2$, are shellable (another term for collapsible)! For a proof see [BurM71]. To put it another way, the higher-dimensional analogs of the Euler operators are not adequate for creating all cell decompositions of higher-dimensional objects.

5.3.5 Sweep Representations and Generative Modeling

Sweep representations correspond naturally to the way many mechanical parts are manufactured. The basic idea of this scheme is to "sweep" one set **A** along another **B**. See Figure 5.16. There are several different types of sweeps.

Translational sweeps: These are common in sheet metal systems. See Figure 5.17(a).

Rotational sweeps: These are used in the context of turned or lathed parts. See Figure 5.17(b).

Solid sweeps: These are used with milling machines. See Figure 5.17(c).

General sweeps: Not much is known in this case because sweeping even nice sets along simple paths can produce nonsolids. See Figure 5.18. It is difficult to guarantee that swept objects will be solids.

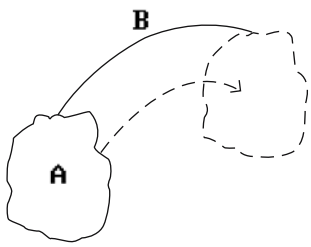


Figure 5.16. Sweeping an object along a curve.

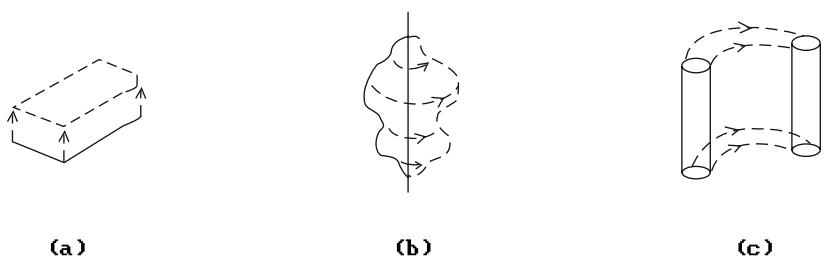


Figure 5.17. Sweep operations.

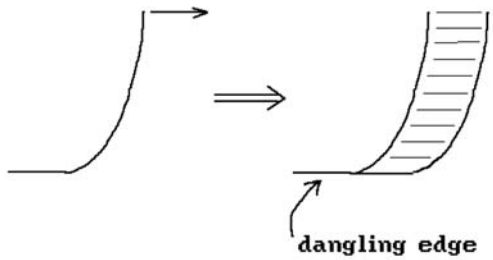


Figure 5.18. Problem with sweeps.

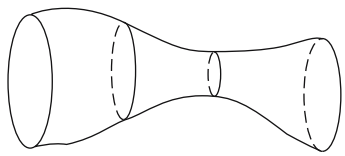


Figure 5.19. Generalized cylinder.

Sweeps sometimes become inputs to other representations. For example, in CADD (a program developed by McDonnell Douglas) one can translate certain sweep representations, such as translational and rotational ones, into boundary representations.

Related to sweeps is the multiple sweeping operation using quaternions described in [HarA02]. There are also the generalized cylinders of Binford ([Binf71]). See Figure 5.19. Here the "sweeping" is parameterized. We shall now discuss a representation scheme developed by J. Snyder at Caltech that is more general yet. It was the basis for the GENMOD modeling system, which Snyder's book [Snyd92] describes in great detail.

Definition. A *generative model* is a shape generated by a continuous transformation of a shape called the *generator*.

Arbitrary transformations of the generator are allowed. There is no restriction as to the dimension of the model. The general form of a parameterization S(u,v) for a generative model which is a surface is

$$S(u,v) = f(\gamma(u), v), \tag{5.3}$$

where $\gamma : [a,b] \to \mathbf{R}^3$ is a curve in $\mathbf{R}^3$ and $f : \mathbf{R}^3 \times \mathbf{R} \to \mathbf{R}^3$ is an arbitrary function. One of the simplest examples of this is where one sweeps a circle along a straight line to get a cylinder. Specifically, let

$$\begin{aligned} \gamma : [0,1] &\to \mathbf{R}^3 \\ u &\to (\cos 2\pi u, \sin 2\pi u, 0) \end{aligned}$$

be the standard parameterization of the unit circle. Define f by

$$f(\mathbf{p}, v) = \mathbf{p} + (0, 0, v).$$

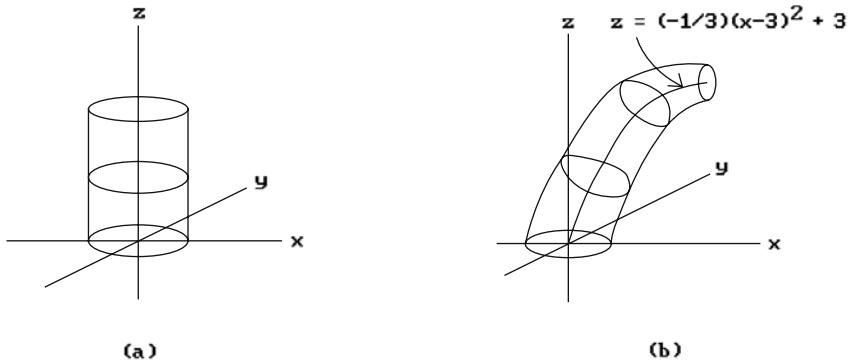


Figure 5.20. Generative models.

See Figure 5.20(a). A more interesting example is shown in Figure 5.20(b) where we use

$$f(\mathbf{p}, v) = \left(1 - \frac{v}{6}\right) R_v(\mathbf{p}) + \left(v, 0, -\frac{1}{3}(x-3)^2 + 3\right)$$

and R_v is the rotation about the y-axis in $\mathbf{R}^3$ through an angle of $\pi v/6$. This corresponds to sweeping the unit circle along the parabola

$$z = -\frac{1}{3}(x-3)^2 + 3$$

in the xz-plane. The circle gets rotated and scaled by a factor of $1 - v/6$ as we move it.

The parameterization S(u,v) in equation (5.3) can be thought of as defining a one-parameter family of curves γ_v defined by $\gamma_v(u) = f(\gamma(u), v)$. As the examples in Figure 5.20 suggest, this family of curves can correspond to a fixed curve being operated on by quite general transformations as it is swept along arbitrary curves. This is, in fact, one reason for creating the generative model representation, namely, that it allows powerful operators for modifying objects.

More generally, generative models of arbitrary dimension have parameterization S(u,v) of the form

$$\begin{aligned} S : \mathbf{R}^k \times \mathbf{R}^s = R^{k+s} &\to \mathbf{R}^n \\ (u, v) \qquad\qquad &\to T(F(u), v), \end{aligned} \tag{5.4}$$

where $F : \mathbf{R}^k \to \mathbf{R}^m$ and $T : \mathbf{R}^m \times \mathbf{R}^s \to \mathbf{R}^n$. This is thought of as a (k + s)-dimensional model obtained by sweeping a k-dimensional object along an s-dimensional path. For example, this allows us to define solids as generative models. A common representation is to represent a solid as sweeping an area along a curve.

Definition. The *generative modeling representation* consists of pairs (**X**,F), where F is a parameterization of the generative model **X** of the form shown in equation (5.4).

The driving force behind GENMOD was correcting some perceived deficiencies in the geometric modeling systems of that time and some key defining points listed by [Snyd92] for the generative modeling approach as implemented in GENMOD are:

(1) The representation is a generalization of the sweep representation.
(2) Shapes are specified procedurally.
(3) Specifying a shape involves combining lower-dimensional shapes into higher-dimensional ones.
(4) An interactive shape description language allows low- and high-level operators on parametric functions.
(5) It is closed, that is, the outputs to operations can be inputs to operations (like CSG).
(6) It allows parameterized shapes whose parameters a user can change.
(7) It supports powerful high-level operators and functions, such as

    reparameterizing a curve by arc length,
    computing the volume of a shape enclosed by surface patches, and
    computing distances between shapes.

    These operations are closed and free of approximation error.
(8) It supports deformation operators, CSG, and implicitly defined shapes.
(9) One has the ability to control the error in the representation.

A large variety of symbolic operators on the parameterizations and their coordinates help the user define generative models, such as vector and matrix operations, differentiation (partial derivatives), integration, concatenation, and constraint operators. Since parameterizations can be thought of as vector fields, another useful operator is one that solves ordinary differential equations. GENMOD had a language in which a user could define models using the various operators.

Now, models will have to be displayed. By converting to polygonal meshes and ad hoc error control, the interactive rendering of generative models becomes feasible. One can specify the subdivisions in two ways: uniform in domain or adaptive sampling. More realistic images can be obtained at the expense of speed.

For accuracy, GENMOD used interval analysis. Interval analysis (see Chapter 18) is an attempt to make numeric computations on a computer more robust and has its advantages and disadvantages. Snyder argued for its use in geometric modeling and described various applications to computing nonintersecting boundaries of offset curves and surfaces, approximating implicitly defined curves and surfaces, and trimmed surfaces and CSG operations on them.

In summary, three more advantages used by Snyder to justify the generative modeling approach are:

(1) The representation handles all dimensions, is high-level, and extensible.
(2) Using a high-level interpreted language, the mathematically knowledgeable user can easily build a library of useful shapes.
(3) An adequate number of robust tools for rendering and manipulating generative models exist.

5.3.6 Parametric Representations

Many of the representations of solids rest on a representation of their boundaries. That was true even in the case of the csg-rep. Although the primitives were solids, in practice one only had equations or parameterizations for their surfaces, and the interior of the solid was not referenced explicitly. As far as parameterizations are concerned, there is no reason why we have to limit ourselves to parameterizations of two-dimensional objects. If we want access to interior points, we can define three-dimensional parameterizations just as easily. For example,

$$p(r, \theta, z) = (r\cos\theta, r\sin\theta, z), \quad r \in [0,1], \quad \theta \in [0, 2\pi], \quad z \in [0, 2],$$

is a parameterization of a solid cylinder of radius 1 and height 2 with axis the z-axis. If we allowed such parameterizations, then we could also generate interior points of the object at will. Chapter 12 describes a number of basic surfaces and their parameterizations. Similarly, one could describe a corresponding basic collection of solids and their parameterizations. In other words, three-dimensional parameterizations are a representation scheme for solids. See [Mort85] for a discussion of what he calls a *tricubic parametric solid*. This is a space parameterized by a function p(u,v,w) of the form

$$p(u,v,w) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} \mathbf{a}_{ijk} u^i v^j w^k, \quad u,v,w \in [0,1] \text{ and } \mathbf{a}_{ijk} \in \mathbf{R}^3.$$

This is the most general cubic parameterization, but one can look at special cases such as Bezier or spline forms, just like in the surface case. See [HosL93].

5.3.7 Decomposition Schemes

Decomposition representation schemes represent objects as a union of quasi-disjoint pieces. These representations come in two flavors: *object-based* or *space-based*. The object-based versions present a subdivision of the object itself. The space-based versions, on the other hand, subdivide the whole space and then mark those pieces that belong to the object. The hatched cells in Figure 5.21(b) define a space-based decomposition representation of the object in Figure 5.21(a). Figure 5.21(c) shows an object-based decomposition of the same object.

Another distinction between decomposition schemes is whether they use a *uniform* or *adaptive subdivision*. The choice is driven by the geometry of the object. For example, at places where an object is very curved it would be advantageous to subdivide it more to get a more accurate representation. Object-based decomposition schemes tend to be adaptive.

Cell Decompositions. This is a very general object-based decomposition representation. Here the primitive pieces that an object is broken into can be arbitrary (curved) cells, typically triangles in the two-dimensional case or tetrahedra in the three-dimensional one. The idea is to find triangular or tetrahedral pieces each of which

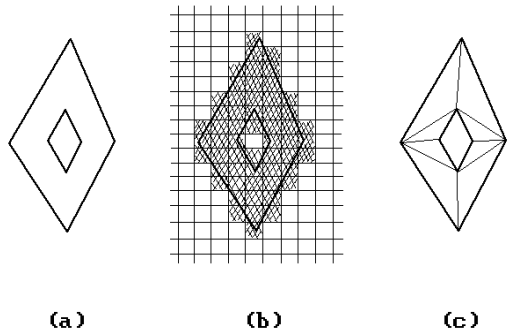


Figure 5.21. Decomposition representations.

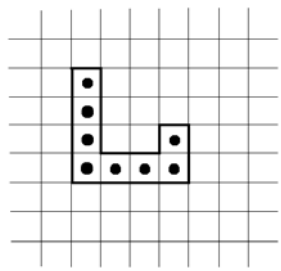


Figure 5.22. Spatial occupancy representation.

has a relatively simple definition, something that presumably the whole object did not have. The representation is unambiguous but certainly not unique. Cell decompositions are an essential ingredient of finite element modeling (see Chapter 19).

Certain important topological properties can be computed relatively easily from a cell decomposition, such as answers to the questions

(1) Is the object connected?
(2) How many holes does it have?

The representation is also good for nonhomogeneous objects. See Section 7.2.4 in [AgoM04] for a general definition of a cell complex. Handle decompositions of manifolds (see Section 8.6 in [AgoM05]) are a special case of this type of representation. Chapter 16 will address the usefulness of "intrinsic" cell decompositions of spaces.

Spatial Occupancy Enumeration. This space-based scheme represents objects by a finite collection of uniformly sized cells. Areas are divided into squares (pixels). Volumes are divided into cubical cells called *voxels*, an abbreviation for "volume elements." There are two choices here in that one can either represent the object boundary or its interior. In the latter case, one can, for example, list the coordinates of the center of grid cells in the object. See Figure 5.22.

Spatial occupancy enumeration is an ambiguous representation. Furthermore, a big problem with this scheme is the amount of data that has to be stored. For that

reason it was not used much for mechanical CAD or CAM (computer-aided manufacture) initially except for gross models to help with certain calculations such as collision checking and getting a rough estimate of volume. This has changed now that computers with gigabytes of memory have become a reality and voxel-based representation schemes for volumes have become very popular in certain parts of computer graphics. A more detailed discussion of this subject follows in the next section. Section 5.8.2 will describe the standard approach to cutting down on the amount of data one has to store.

5.3.8 Volume Modeling

Here are four terms and their definitions that usually appear in the same context:

Volumetric data: The aggregate of voxels tessellating a volume.

Volume modeling: The synthesis, analysis, and manipulation of sampled, computed, and synthetic objects contained within a volumetric data set.

Volume visualization: A visualization method concerned with the representation, manipulation, and rendering of volumetric data.

Volume graphics: The subfield of computer graphics that employs a volume buffer for scene representation and is concerned with synthesizing, manipulating, and rendering such scenes. Volume graphics is the three-dimensional counterpart of raster graphics.

The definitions are taken from [KaCY93] and are an adequate representation of how these terms are usually used. The subject matter that is addressed by these terms is what this section is about. It really only dates back to the early 1980s and started in the context of tomography.

Although our main interest in this book is on modeling geometric objects, volume modeling covers a much broader subject in that the "volumes" may have arisen in other ways. Volume modeling in its most general sense deals with scalar-valued functions defined on three-dimensional space. In that sense, it is not really a modeling scheme per se but has close connections with modeling. In the special case where the function takes on only two values, 0 and 1, we can, in fact, interpret the function as defining a space-based decomposition scheme generalizing the voxel-based spatial occupancy enumeration scheme. The voxel case is the uniform case, but the data set may have different geometries such as being composed of rectangular or curved cells. Cells might be different distances apart. On the other hand, the function could come from some arbitrary mathematical model. For example, one might want to display the temperature of a heated solid visually, perhaps by displaying the surfaces of constant temperature. We can think of volume modeling as modeling data that is acquired from appropriate instruments and then sampled to get the voxelization. The data could also be an "object" that is defined in terms of point samples. *Volume rendering* refers to the process of displaying such models. We shall have more to say about volume rendering in Section 10.4.

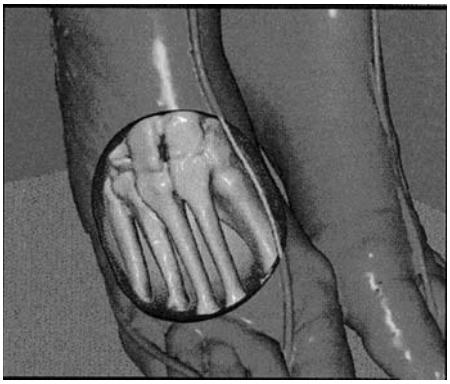


Figure 5.23. Foot with bones exposed ([ScML98]). (Reprinted from Schroeder et al: The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, third edition, 2003, 1-930934-07-6, by permission of the publisher Kitware Inc.).

Volume modeling is beginning to make an impact on the more conventional CAD and CAGD. Here are some of its advantages:

(1) One can "cut away" parts of an object and look at its interior. See Figure 5.23.
(2) CSG can be implemented quite easily because at the voxel level the set operations are easy, especially if one has support for voxBlt (*voxel block transfer*) operations that are the analog of the bitBlt operations.
(3) Rendering is viewpoint independent.
(4) It is independent of scene and object complexity.

The author has felt for many years that it was advantageous to model the whole world and not just the objects within it. It gives one much more information. For example, to trace a ray, one simply marches through the volume and sees what one hits along the way, rather than having to check each object in the world for a possible intersection. Volume modeling is now making this possible.

Some disadvantages of volume modeling are:

(1) A large amount of data has to be maintained.
(2) The discretization causes loss of information.
(3) The voxelization causes aliasing effects.

Volume modeling plays an important role in the visualization of scientific data. This is a big field in computer graphics. Although not the focus of this book, it would not be right to omit mentioning some examples of it:

Medical Imaging. This was one of the first applications of volume modeling. See [StFF91] for an overview of early work. Physicians used MRI (magnetic resonance imaging) and CT (computed tomography) scanners to get three-dimensional data of a person's internal organs. In tomography one gets two-dimensional slices of the object using X-rays. One projects X-rays through the body and measures their intensity with detectors on the other side of the body. The X-ray projector is rotated about the body and measurements are taken at hundreds of locations around the patient. A

picture of the slice is then obtained from a reconstruction process applied to all this data. Radiologists were apparently good at seeing three-dimensional models from these two-dimensional slices, but surgeons and doctors were not. Fortunately, there exist algorithms that, when applied to a stack of such slices, produce a representation of the whole organ and volume rendering makes it possible to display it. One is able to remove uninteresting tissues to see those parts that one wants to see. At this point in time, three-dimensional medical graphics is not yet widely used, mainly because of the cost. Also, the slices are more accurate and have more information than the three-dimensional reconstruction, so that radiologists tend to refer to them more.

In another recent development, surgeons can now also use haptic systems to practice surgeries beforehand. "Haptic" means that one gets physical touch feedback from the system.

Modeling Natural Phenomena. Understanding the flow of air over an airplane wing is important for its design. A similar understanding is needed for designing intake or exhaust manifolds in engines. This is where fluid dynamics enters. Fluid dynamics deals with fluid flow, which is governed by a set of differential equations called the Navier-Stokes equations. These equations define the velocity and vorticity of the fluid. The *vorticity* describes the rotational part of the flow and is defined by a vector at each point of the fluid. Understanding vector-valued functions is not easy, but volume-rendering techniques have enabled scientists to get a better visual understanding of what happens inside a flow. Volume modeling has been helpful in modeling other phenomena such as ocean turbulence and hurricanes. Oil exploration has been greatly aided by the ability to use volume modeling to analyze geological data.

Education. Volume modeling has been used to avoid having to use actual bodies in dissection experiments. As a result of the visible human project sponsored by the National Library of Medicine, there now exist models of a human male and female. If one tried to model a human in the more traditional way by means of facets, it would take millions of triangles to do so.

Nondestructive Testing. Volume modeling has been used to enable mechanical and materials engineers to find structural flaws in objects without having to take them apart.

This ends our brief overview of volume modeling. We return to the very interesting topic of volume rendering in Section 10.4. There is a large body of literature on volume modeling and the related subject of scientific visualization. A good place to begin more reading is [LiCN98], [ScML98], and various ACM SIGGRAPH course notes such as [Kauf98].

5.3.9 The Medial Axis Representation

In mathematics, when one tries to characterize or classify geometric objects, one first looks for coarse invariants (topology) and then successively refines the classification by adding metric criteria, differentiability criteria, etc. For example, at a very top level,

a doughnut and a circle are similar because one can collapse the doughnut down to a circle. A double doughnut (two doughnuts attached to each other along a disk) is like a figure-eight curve. Therefore, since the circle is clearly a quite different shape from a figure-eight, one can see that the more complicated solids to which they are associated must also be fundamentally different shapes. This section is about a similar idea, namely, to facilitate dealing with objects by representing them by simpler (lower-dimensional) objects that nevertheless still capture the essence of the shape of the original object. The idea of using a "skeleton" of an object as a shape descriptor goes back to [Blum67] and [Blum73]. The fact that one gets a representation that has many attractive features has led to quite a bit of research activity on this subject. It should be noted, however, that the skeletal representation of an object is not a stand-alone representation for objects in practice. Mostly, it is intended to be used in conjunction with others, typically a boundary representation for continuous objects and a spatial occupancy enumeration representation based on pixels or voxels for discrete objects.

Skeletons come in two flavors, namely, continuous and discrete. We shall begin with definitions for the continuous case.

Definition. Let $\mathbf{X} \subseteq \mathbf{R}^n$. A *maximal disk* in $\mathbf{X}$ is a closed disk $\mathbf{D}^n(\mathbf{p},r)$ contained in $\mathbf{X}$ with the property that it is not properly contained in any other closed disk in $\mathbf{X}$.

Definition. Let $\mathbf{X} \subseteq \mathbf{R}^n$. The *medial axis* (*MA*) or *skeleton* or *symmetric axis* of $\mathbf{X}$ is the closure of the set of centers of maximal disks in $\mathbf{X}$. The medial axis of a solid in $\mathbf{R}^3$ is sometimes called a *medial surface*. The real-valued function that assigns to each center of a maximal disk in $\mathbf{X}$ the radius of that disk extends to a continuous function on the medial axis called the *radius function* of that medial axis.

Note. Unfortunately, there is not complete agreement with regard to the terms medial axis, skeleton, and symmetric axis in the literature. For example, the medial axis in the continuous case is often also defined as the set of points equidistantly closest to at least two points in the boundary. The advantage of the definition given here with its closure condition is that if $\mathbf{X}$ is bounded then the medial axis will be a compact set.

Figure 5.24 shows the medial axis (indicated by solid lines) of a planar L-shaped bracket and a three-dimensional block. For a convex planar polygon it always con-

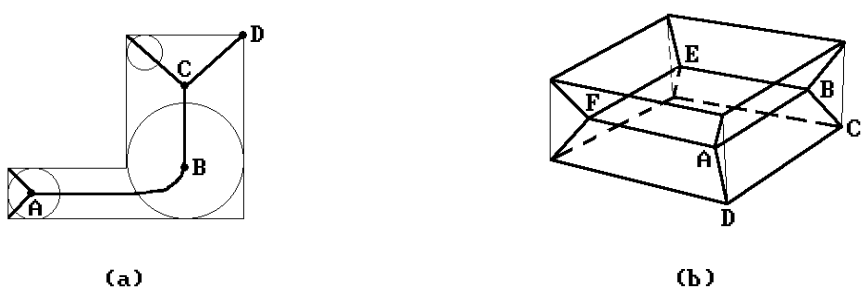


Figure 5.24. Medial axes.

sists of straight line segments but if the polygon is nonconvex there may be curved arcs as Figure 5.24(a) shows. There is a close relation between the medial axis and the Voronoi diagram of an object ([ShAR96]).

The medial axis for a polyhedron has a natural partition into cells. Determining the medial axis basically reduces to determining its cell decomposition. In two dimensions the cells are called *arcs* and *junctions*. For example, in Figure 5.24(a) **BC** and **CD** are arcs and the points **A**, **B**, and **C** are called junctions. In the nondegenerate case, junctions are the points where the maximal disk has three or more contact points with the boundary. The maximal disks at endpoints of arcs that lie in the boundary, like point **D**, have one contact point with the boundary. In three dimensions the cells are called *sheets*, *seams*, and *junctions*. The sheets are surface patches. These are further subdivided into *wing sheets* and *body sheets*. Wing sheets are those with points in their boundary where the maximal disk makes contact with the boundary at only one point, such as **ABCD** in Figure 5.24(b). Body sheets are the remaining sheets, such as **ABEF**. The seams are curves that typically are the intersection of two or more sheets where the maximal disk has three or more contact points with the boundary. Junctions are points that are the intersections of three or more sheets. See [BBGS99].

Next, consider discrete objects. We could give the same definitions because all that we need is a metric which we have. However, there are several natural metrics to choose from in this case and so it is possible to play around with the definition a bit and choose a variant which may be more suitable for a particular discrete problem. We follow [RosK76].

Definition. Let $\mathbf{X} \subseteq \mathbf{U} \subseteq \mathbf{Z}^n$. The *medial axis* (*MA*) or *skeleton* or *symmetric axis* of $\mathbf{X}$ with respect to $\mathbf{U}$ is the set of points whose distances from the complement $\mathbf{U} - \mathbf{X}$ are a local maximum, that is, no neighboring point has a greater distance to the complement. The *distance function* for the medial axis is the real-valued function that assigns to each point of the medial axis its distance to $\mathbf{U} - \mathbf{X}$.

In practice, the "universe" $\mathbf{U}$ is a rectangle when n = 2 (the pixel case) and a rectangular box when n = 3 (the voxel case). Figure 5.25 shows the medial axes for a 7 × 8 discrete rectangle in $\mathbf{Z}^2$. In Figure 5.25(a) we used the taxicab metric and in Figure 5.25(b), the max metric. The medial axes are the numbered points with the numbers giving the distance of the point to the complement.

```
1 · · · · · · 1
· 2 · · · · 2 ·
· · 3 · · 3 · ·
· · · 4 4 · · ·
· · 3 · · 3 · ·
· 2 · · · · 2 ·
1 · · · · · · 1
```

(a)

```
· · · · · · · ·
· · · · · · · ·
· · · · · · · ·
· · · 4 4 · · ·
· · · · · · · ·
· · · · · · · ·
· · · · · · · ·
```

(b)

Figure 5.25. Discrete medial axes.

Definition. The *medial axis representation* or *medial axis transform* (*MAT*) of an object consists of its medial axis together with the associated radius function in the continuous case and the distance function in the discrete case.

One can show that an object is completely specified by its medial axis representation. See [Verm94] and [RosK76]. Furthermore, in the continuous case the envelope of the maximal disks is just the boundary of the object. One nice thing about the medial axis representation is that it depends on the geometry of the object and not on the choice of coordinate axes like the quadtree or octree representation for discrete objects defined by pixels or voxels.

Algorithms that compute medial axes divide into two types based on whether they apply to discrete or continuous objects. The basic thinning algorithm for computing the discrete medial axis is often referred to as the "**grassfire**" **algorithm**. If a fire started at the boundary of the object were to burn into the object at a constant rate, then it would meet in the medial axis. One starts on the boundary of the object and strips away one layer of pixels or voxels after another until one reaches points that the fire reaches from two directions. See [RosK76] and [WatP98] for thinning of two-dimensional discrete sets. Similar arguments work in three dimensions.

In describing algorithms for finding the medial axis of continuous objects we shall concentrate on three-dimensional objects. Such algorithms can be classified by the specific approach that is used: volume thinning, tracing of seams and sheets, Voronoi diagrams, or Delaunay triangulation. See [BBGS99] for advantages and disadvantages for various schemes. [CuKM99] also describes previous work.

Volume Thinning. One voxelizes the object and then computes the discrete medial axis that is then polygonized. An additional extra pass is required at the end to determine the radius function. Of course, this will only determine an approximation to the medial axis and one must be careful that it is accurate.

Tracing Approaches. One tracing approach is described in [ShPB95]. One starts at a known junction like a vertex of the polyhedron and then traces along an adjacent seam, defined as the zero set of some functions, until one gets to another junction. At that point one repeats this process for each seam that ends at that new junction. Polygonal approximations to the seams are computed. The main difficulty is determining the next junction. A similar approach is used in [CuKM99] but is claimed to be more accurate because it uses exact arithmetic.

Voronoi Diagrams. A number of algorithms use Voronoi diagrams because of their close connection to the medial axis problem since they also deal with equidistant sets of points. See Section 17.7 for a definition of Voronoi diagrams and some of their properties. The idea is to use a suitable sample of points in the boundary and compute their Voronoi diagram. See [Bran92], [ShPB95], or [ShPB96]. [CuKM99] describes an algorithm for polyhedra via Voronoi diagram and exact arithmetic.

Delaunay Triangulations. See Section 17.8 for a definition of a Delaunay triangulation of a set of points. A Delaunay triangulation is the geometric dual to the Voronoi diagram. [ShAR95] and [ShAR96] generate a domain Delaunay triangulation consisting of a set of tetrahedra based on an adaptive collection of boundary points. The medial axis is obtained from this triangulation.

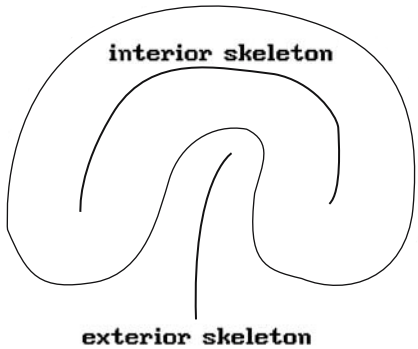


Figure 5.26. Interior and exterior skeletons.

Most algorithms are basically discrete algorithms. One exception is the algorithm described in [Hoff94] for CSG objects. In this regard, see also [LBDW92]. The authors describe how one can obtain an approximation to a variant of the Voronoi diagram for CSG objects. Sometimes bisectors of surfaces are rational. See [ElbK99]. In case of polyhedra, the medial surface consists of bisectors that are planes or quadric surfaces. An algorithm for planar regions with curved boundaries can be found in [RamG03].

To use the medial axis representation effectively one needs to know not only how to compute it but also how one can reconstruct the original object from it. The latter task is often referred to as *refleshing*. Algorithms for refleshing divide into direct and implicit approaches.

The **direct approach** to refleshing tries to reconstruct the boundary surface of the original object directly using the given radius or distance function. This amounts to constructing the surface from a variable offset type surface. See [GelD95]. Self-intersections are a problem with offsets and the exterior skeleton has been used here to help prevent these. See [STGLS97]. The *exterior skeleton* or *exoskeleton* of an object is the skeleton of the complement of the object. The ordinary skeleton is sometimes called the *interior skeleton* or *endoskeleton*. The exterior skeleton comes in handy at those places where the boundary of the object is concave. See Figure 5.26.

The **implicit approaches** to refleshing try to define the boundary surface implicitly as the zero set of a suitable function. They can be further subdivided into those that use a distance function and those that use convolution methods. See [BBGS99] for a more detailed discussion of this along with references. The paper also describes a new distance function approach. This involves triangulating the medial axis and defining a local distance function for each triangular facet. The global distance function is then the minimum of all the local ones. To give the reader a flavor of how a local distance function is constructed, we sketch the construction in the case where the radius function is constant over a facet. The local distance function is a composite of functions defined over regions that are related to the Voronoi cells associated to the facet, its edges, and its vertices. See Figure 5.27, which shows a triangular facet with vertices $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ and its associated Voronoi cells whose boundaries are obtained by sweeping a vector orthogonal to the edges of the facet orthogonally to the plane of the facet. Let f_{si} be the local distance function for point $\mathbf{p}_i$ determined by the

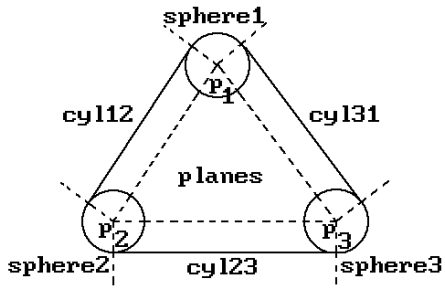


Figure 5.27. The regions used to define a local distance function for a facet.

sphere labeled sphere**i**. Let f_{cylij} be the local distance function associated to the cylinder labeled cyl**ij** that is centered on the edge from $\mathbf{p}_i$ to $\mathbf{p}_j$ and meets the spheres labeled sphere**i** and sphere**j** tangentially. Let f_{planes} be the local distance function associated to the planes that meet the spheres labeled sphere**i** tangentially. Then the local distance function f(**p**) associated to the facet is defined by

$$\begin{aligned} f(\mathbf{p}) &= f_{si}(\mathbf{p}), && \text{if } \mathbf{p} \in \text{sphere labeled sphere}\mathbf{i} \\ &= f_{cylij}(\mathbf{p}), && \text{if } \mathbf{p} \in \text{cylinder labeled cyl}\mathbf{ij} \\ &= f_{planes}(\mathbf{p}), && \text{if } \mathbf{p} \in \text{region labeled planes.} \end{aligned}$$

If the radius is not constant over a facet but varies linearly over it, then a similar construction works using cones rather than cylinders. In the end, the refleshed object is defined as the halfspace of a (distance) function. The implicitly defined boundary (the zero set of the function) can then be polygonized by some standard method if this is desired.

One goal of the medial axis representation is to make modeling easier for the user. For one thing, we have reduced the dimension by one. An example of this is the representation of an object by orthogonal projections. See [Bloo97], [STGLS97], and [BBGS99] for how a user might edit an object using its medial axis. In [BBGS99] the basic approach to editing a solid was

(1) Compute the medial axis and radius function for the solid.
(2) Allow the user to interactively edit the skeleton and radii.
(3) Reflesh to obtain the edited solid.
(4) Polygonize the boundary of the solid so that the user can use the b-rep for other purposes.

The allowed editing operations were

(1) Stretching: The user picks skeletal vertices and a translation vector.
(2) Bending: The user picks a joint and specifies a rotation by clicking with the mouse on one side of a separating plane through the rotation axis.
(3) Rounding: At sharp convex edges the wing sheets meet the boundary of the object and the disk radii go to zero. The user can either remove the wing sheets or change the disk radii.

(4) Editing disk radii: This allows a user to round, thicken, or thin parts of the object in uniform or nonuniform ways.

The bending operation in particular shows why the medial axis representation has an advantage over a b-rep. With a b-rep such an operation can produce tears if one is not careful. Although bending the medial axis may produce tears or intersections, the refleshing operation removes all that.

Medial axis computations have many applications. Just to list a few topics and references, they are used in finite element mesh generation ([STGLS97]), shape optimization and robot path planning ([GelD95]), and pattern analysis and shape recognition ([FarR98]). See [Nack82] for relationships between the curvature of a surface and curvature functions associated to its medial axis representation.

Finally, related to the medial axis are the **level sets** of [LazV99] and the **Reeb graph** of [ShKK91] and [ShiK91]. With level sets the goal was to describe both the topology and geometry of the object, whereas with the Reeb graph the goal was to encode the topology. Both of these approaches are based on the handle decomposition of manifolds that is central to the classification of manifolds. See Chapter 8 in [AgoM04]. Reeb graphs have also been useful for volume data mining ([FTAT00]).

5.4 Modeling Natural Phenomena

Except for the pixel- and voxel-based types, the representation schemes we have discussed so far are not very useful for modeling natural phenomena. Objects such as trees, mountains, grass, or various terrain cannot easily be modeled by linear polyhedra or smooth surface patches. Using very small pieces in the representation would overwhelm one with massive amounts of data. Even if this were not a problem, it would not be a satisfactory solution. The picture might look all right at the start, but what if one were to zoom in? One would have to adjust the fineness of the subdivision dynamically to prevent things from eventually looking flat. Modeling and rendering natural phenomena is a digression from the main thrust of this book. For that reason, we shall only take a brief look at this subject. The four topics we consider are fractals, iterated function systems, grammar based models, and particle systems.

Fractals. One of the most important applications of fractals to graphics is in the representation of natural phenomena. For a definition of a fractal, see Section 22.3. They enable one to represent such phenomena in a realistic way with a small amount of data. The zooming problem also is no problem here. There is one caveat however. Fractals are typically used to represent "generic" trees, mountains, or whatever. They do not lend themselves easily to represent a specific tree or mountain. This is usually not an issue though.

Why are fractals so great for modeling certain natural phenomena? To begin with let us show how fractal curves and surfaces can be generated. The basic construction generalizes that of the well-known Koch curve (see Section 22.3).

In the one-dimensional case, the algorithm starts with a given initial polygonal curve and then generates a sequence of new curves, each of which adds more detail to the previous one. In every iteration we replace each segment of the old curve with

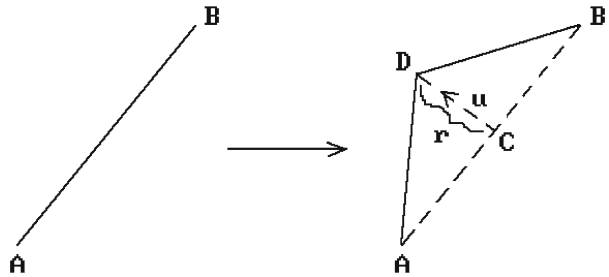


Figure 5.28. Curve midpoint displacements.

(a) Rough sketch (b) After random subdivisions

Figure 5.29. A fractal island.

a new curve segment. The simplest way to do this is to displace the midpoint of the segment by a random amount along the perpendicular bisector. See Figure 5.28. Given the segment **AB**, let **C** be its midpoint. Compute a unit normal vector **u** for it, choose a suitable random number r based on the current scale, and replace **AB** by the segments **AD** and **DB**, where **C** is the midpoint of **AB** and **D** = **C** + r**u**.

Now, to describe some natural shape such as the boundary of an island proceed as follows: Specify the rough outline of island with a polygonal curve and then apply the algorithm described above, that is successively replace each edge with an appropriate collection of edges. Figure 5.29 shows one possible result after starting with an approximation to the Australian continent. One does have to deal with the problem of self-intersections in the resulting curves.

In the two-dimensional case, we have more freedom. For example, for surfaces described as a collection of triangles one common approach is to do the following: "Subdivide" each triangle into smaller triangles obtained by connecting its vertices to appropriate random offsets of the midpoints of its sides. This replaces each triangle successively by seven new smaller triangles and the process can be repeated. In Figure 5.30(a) the midpoints of the edges of triangle **ABC** were offset to **D**, **E**, and **F**, and the triangle replaced by triangles **ABD**, **BDE**, **BEC**, **CEF**, **ACF**, **ADF**, and **DEF**. A similar construction works for quadrilaterals. There is one complication in the two-dimensional case, namely, if one is not careful, then gaps can appear in places where triangles used to be adjacent. Figure 5.30(b) shows the potential problem if we offset

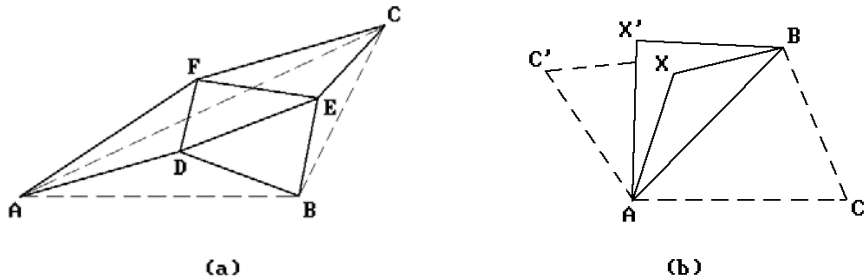


Figure 5.30. Surface midpoint displacements.

the midpoint of the edge **AB** to **X** and **X′** with respect to the triangles **ABC** and **ABC′**, respectively. One has to make sure that one uses the same random number and normal when offsetting an edge with respect to both triangles that have it in common.

Some good references on fractals are [Mand83] and [DevK89].

Iterated Function Systems. Iterated function systems are an elegant way to generate fractals. We refer the reader to Section 22.4 for a brief discussion. [Barn88] is a good reference.

Grammar-Based Models. Building on work of Lindenmayer ([Lind68]) on *parallel graph grammars*, Smith ([Smit84]) described a class of plant models that he called *graftals*. The modeling involved two stages: first one generates a formal string from an initial string using production rules and then the image is generated by interpreting this string as a geometric tree in a suitable way. Graftals were not necessarily fractals, but were very similar in that one could generate as much detail as desired. Very realistic plants and trees could be generated using botanical laws.

Particle Systems. Particle systems, introduced in [Reev83], were good at modeling phenomena that was "fluid-like" and changed over time by flowing or spattering, such as clouds, smoke, fire, water falls, and water spray. Typical particles were spheres and ellipsoids. They would be created in a random way with respect to size, position, shape, and other properties and then randomly deleted. During their lifetime their paths, which could be controlled by specified physical forces, would be rendered in appropriate colors. See also [ReBl85].

Most models above are what are called *procedural models*. We have more to say about this later in Sections 5.6 and 5.11.

5.5 Physically Based Modeling

The kind of modeling we have discussed so far has dealt mostly with static objects. Allowing for animation would not change that since animation is nothing but a matter of generating a sequence of views of static objects (rather than a single view). This static modeling is what the traditional CAD systems do and is quite adequate in aiding

users in the tasks for which they are used. The real world does not consist of isolated objects however. Objects interact due to a variety of forces. We need to broaden our outlook. Another goal should be to facilitate the modeling of the real world with its interactions. Geometric modeling, the modeling of isolated static objects, is an important step toward modeling real world scenes, but it is only a first step. The next step is to make it easier for users to include the interactions of the objects.

For example, if we wanted to model a ball in a scene with a cloth draped over it, we could do it with the standard modeling system, but it would take quite some effort. We would have to figure out the creases in the cloth on our own. These creases are determined by gravity and other physical forces associated to the particular material from which the cloth is made. We could use the relevant equations of physics to define the set of spline surface patches, or whatever, that would generate the correct picture. How much easier it would be if we only had to tell the CAD program the position and radius of the ball, the material properties of the flat cloth, the starting position of the cloth parallel to the floor at the top of the ball, and then let the program compute the final shape of the cloth after it has reached equilibrium with respect to the forces acting on it. Obviously, a program that could do this would have to have the relevant equations and algorithms programmed within it, but this would only have to be done once.

As another example, suppose that we wanted to show a ball bouncing on a floor. Again, we could do this animation ourselves with a traditional CAD system by determining by hand the series of positions of the ball along with the time intervals between those positions that made up the animation. Why can the CAD program not do this for us, so that all we had to input was an initial height from which the ball is dropped? Obviously, the CAD system could be programmed to do this. This would take some hard work, but again, it would only have to be done once, and then it could help many users in this and similar types of problems.

Modeling that also considers the dynamics of physical objects in addition to their static properties is called *physically based modeling*. The objects may be simple particles or rigid objects, but could also be much more complex, like cloth. As indicated earlier, we are not really dealing with a new representation scheme but rather an extension of "traditional" representation schemes. This is a relatively new branch of computer graphics, with the name "physically based modeling" being introduced for the first time in an ACM SIGGRAPH 87 course by A.H. Barr ([BarrA87]). To carry out its program involves a great deal of knowledge about physics and other sciences.

Physically based modeling can be interpreted quite generally to encompass the three main areas in computer graphics, modeling, rendering, and animation, but at its core, it deals with classical dynamics of rigid or flexible bodies, interaction of bodies, and constraint-based control. An active area of research is how a user can best control the models. There is a trade-off between realism and control. If the models perform realistically, they are typically controlled by differential equations and the only control a user has in initial conditions. On the other hand, giving the user more control might mean that objects will perform less realistically. Constraint-based techniques are a common way to deal with this problem. This includes constraints defined by equations in physics but also refers to situations where we would like the user to be able to say things like "move object A so that it touches both objects B and C," "let a ball roll down a hill following a given path," or "show a moving robot, figure, or object in a domain with obstacles." Unfortunately, if constraints are not chosen care-

fully, we may create underconstrained situations where there is no unique solution, or overconstrained situations where there is no solution.

For a more thorough discussion of physically based modeling see the references in that section of the bibliography.

5.6 Parametric and Feature-Based Modeling

We have mostly talked about various technical aspects of modeling systems, but a good modeler must also take the user's or designer's point of view into account. The difference between a machine representation and a user representation was briefly alluded to in Section 5.3. We also touched on this subject in our discussion of generative modeling in Section 5.3.5. Users should not have to be forced to adapt their way of describing geometry to any low-level abstractions due to technical requirements of a modeler. Defining nontrivial geometric models is usually a difficult task in the best of circumstances. If possible, the process should require no more expertise than the understanding of the final model. Of course, there are times when knowing the underlying mathematics is essential to building a model and so the option of taking advantage of it should be there. Nevertheless, for those times when it is not, we would like a modeler to have the ability to understand high-level, user-friendly concepts. Of course, what is considered user friendly depends on the user. In this section we are concerned with manufacturing environments, where, for example, designers often think of geometric objects in terms of important features that are associated to them, such as, "a block with a slot and rounded corners." Today's modelers have a long way to go in fully supporting that type of interface. This section will introduce the reader to what is referred to as *feature-based modeling*. [ShaM95] is a good reference for this subject. A brief survey can be found in [SodT94].

Systems using *parametric* or *variational models* were a first step toward feature-based modeling and making life easier for the designer. As is pointed out in [ShaM95], perhaps 80% of new designs are simply variations of old ones. Therefore, given the effort that is sometimes needed to create an individual model, why should one have to create each of these variants from scratch? As an oversimplified example of the type of interface that would be found in a modeler using parametric models, the object in Figure 5.31 might be defined by the following sorts of commands:

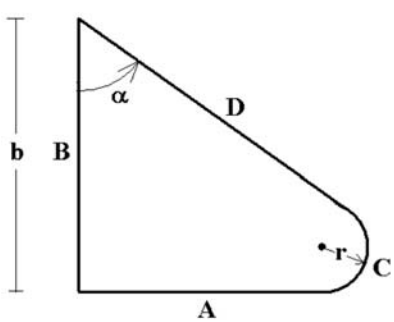


Figure 5.31. A parametric model.

(1) horizontal line **A** of length a
(2) line **B** perpendicular to line **A** of length b
(3) line **D** makes signed angle α with line **B**
(4) circular arc **C** tangent to lines **A** and **D**

It would then be up to the system to check if this description gives rise to a unique consistent model for particular values of a and b. The basic design process in such a modeling system is then that the user describes a list of geometric primitives to be used and the geometric relationships between them symbolically without any numbers. After the user inputs the actual geometric constraints, the system creates an actual instance of the object if possible. The user can subsequently input new data and get new instances of the object. Note that the "parametric" models considered here are higher-level constructs than those in the generative representation discussed in Section 5.3.5.

Although the terms "parametric" and "variational" are often used interchangeably, there is a subtle distinction between what are called parametric and variational methods. Parametric methods solve constraints by replacing symbolic variables by values that are computed sequentially from previously computed variables. Variational methods use equations to represent constraints and solve them simultaneously. The difference is captured by the difference between defining a variable via a formula or implicitly. For more on parametric and variational modeling see [ShaM95]. Some sample papers on constraint-based modeling with additional references are [LiHS02] and [Podg02].

An approach to geometric design based on functional programming that extends variational modeling is described in [PaPV95]. The authors discuss a high-level functional programming language (a language that manipulates functions) with the underlying geometric objects represented in a hierarchical manner much like in CSG. Elementary polyhedra are stored as inequalities and the basic Boolean set operations are supported. The language is such that all syntactically correct objects are valid. It is argued that the power of this functional approach is that it lets the user naturally generate new models from old ones and is similar to generative modeling in this respect.

Parametric and variational modeling is a start toward facilitating geometric design, but it still only deals with individual geometric primitives with no grouping capabilities and lacks a vision of the whole modeling process. Consider a manufacturing company. Its ability to deal with the design, planning, and manufacturing process in an integrated way is clearly of practical importance. To do this one needs to model the whole process. However, the models used by a designer should be allowed to be different from those used by the person manufacturing the product about which the designer may know little. Both may evolve over time and one only needs a way to map from one to the other. Feature modeling seems like a promising approach to an integrated solution. Again, the problem with the type of modelers we have been discussing up to now is that they dealt solely with the geometry of objects and ignored many of the other issues such as process planning, assembly planning, and inspection planning. Even in the case of just the geometry they were not totally adequate since they tended to be low-level and did not make it easy for a designer to make changes, although parametric modeling helped.

Feature-based modeling dates back to the mid-1970s when attempts were made to get data on manufacturing features for NC programming. Kyprianou [Kypr80] was the person who first introduced automated feature recognition into a modeler (BUILD). [PraW85] was one of the earliest studies of "design by features." So what exactly is a "feature?"

The term "feature" refers to a high-level abstract entity. It refers to some interesting aspect of, or associated to, an object and is usually a combining of details into one entity that is more meaningful for manipulation than the individual parts. For example, in a b-rep modeler a block with a hole through it might consist of a collection of surface patches with no explicit notion of the center and radius of the hole. Moving the hole might then involve moving and changing a subset of these patches – a tedious task. The term "feature" was first used in manufacturing but has since taken on a broader meaning. Machined parts typically can be described by things like holes, slots, pockets, and grooves. A relatively small collection of such features might have been adequate to describe a part in a particular manufacturing environment and with them one might then be able to create a manufacturing plan. Features are important to automating the design to manufacturing process because they help define the functionality of objects. [ShaM95] lists the following characteristics of a feature:

(1) It is a physical constituent of a part.
(2) It is mappable to a generic shape.
(3) It has engineering significance. (This may involve describing its function or how it "behaves.")
(4) It has predictable properties.

A *feature model* is a data structure representing a part or assembly mainly in terms of its constituent features. It is convenient to subdivide features into the following subtypes:

(1) Geometric features

(a) Form features:	They describe some idealized geometry.
(b) Tolerance features:	They describe variance constraints from the idealized geometry.
(c) Assembly features:	This is a grouping of various feature types into a larger entity and includes kinematic relationships.

(2) Functional features:	These describe the overall function of the part.
(3) Material features:	These give material names, specify treatments such as painting, etc.

Figure 5.32 shows a standard example of what one means by form features. It is a slightly modified version of the CAM-I ANC101 part that is not the picture of any real functioning object but is simply used to test geometric capabilities of modelers. (CAM-I is an abbreviation for Computer Aided Manufacturing, Inc., a nonprofit consortium in Arlington, Texas.) Form features can be primitive or compound. For example, one can talk about a specific pattern of holes rather than just an individual

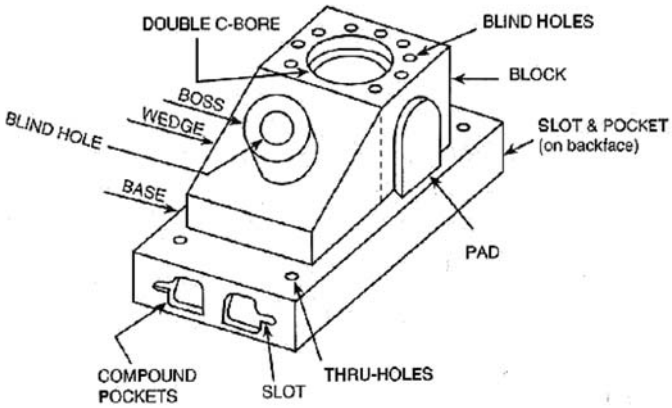


Figure 5.32. Modified ANC101 test part (CAM-I and [ShaM95]). (This material is used by permission of John Wiley & Sons, Inc. from Parametric and Feature-Based CAD/CAM, Shah, 1995 John Wiley & Sons, Inc.)

hole. Tolerance constraints are needed to ensure that parts will work as specified given the inevitability of inaccuracies in the manufacturing process. The three geometric features (form, tolerance, and assembly features) are the ones that are mostly supported by modelers. Support for functional features is currently still very weak because it assumes a lot more intelligence on the part of modelers than they currently have. Although there is no limit on the number of features one can define, attempts have been made to create taxonomies for them. This is important if one is to have any data exchange standards.

Just as one has to worry about the validity of geometry, one has to also make sure that features are created, modified, and deleted in a valid way. Unfortunately, this is not a mathematical process like it was in the case of geometry. [ShaM95] mentions four general classes of validity checks that are needed:

(1) Attachment validation:	A recess feature cannot be attached to an outside face of a block.
(2) Dimension limits:	A hole cannot be larger than the block that contains it.
(3) Location limits:	A hole should not get too close to the edge of a block.
(4) Interaction limits:	This is where two or more features change the geometry or semantics of features. The geometry may not necessarily be invalid. For example, a larger hole may delete a smaller hole.

As one moves from one stage to another in the manufacturing process, the features that are relevant to the persons involved may change. At the design stage, one may worry about the strength of a particular geometric configuration, whereas at the manufacturing stage this may no longer be relevant and the only features one may care about is where certain holes are to be drilled. This calls for mappings from one feature model to another and is referred to as *feature mapping*.

Modelers are typically not concerned with features as such, at least not explicitly. One either needs to add to them the ability to deal with features or add features as a set of primitive data structures. For example, CSG is not detailed enough for many features and b-rep is too detailed. So how do we add feature capability to modelers? The three standard approaches are

(1) Interactive feature definition:	A human determines the features either at model creation time or later.
(2) Automatic feature recognition:	Here one extracts feature information à posteriori if the geometric models are already defined.
(3) Design by feature:	Here one designs with features in the first place.

Approach (1) is easiest to implement, but it is up to the user to check for validity and it may be a tedious job if there are lots of features. Approach (2) is much more complicated than (1), but has been incorporated in some modelers (for example, BUILD). A number of different algorithms have been developed to get a handle on the recognition problem. One basically needs a program that looks for patterns. For example, to look for a pocket in a face one could look for a cycle of edges at which the solid has a convex corner. Difficult as automatic feature detection may be, one may need it if different features are used at different stages in the manufacture of an object. With regard to approach (2), some features could be defined at model creation time, as when one creates a slot by sweeping. However, these would by in large be purely geometric features and not all features are that. Furthermore, the primitive operations of a modeler may not directly correspond to the features that are of interest to someone and some may lead to ambiguous features (see Figure 5.33). A good overview of feature recognition techniques can be found in [JiMa97].

Approach (3) is probably the most attractive. A modeler might have a menu allowing a designer to create an object in terms of its features. One would be able to create an object with a slot or hole in essentially one step, or at least in a number of steps that depended on the number of varying parameters for that particular shape. Figure 5.34 shows 10 of the 13 steps needed to create the ANC101 part in Figure 5.32. Although this might make life easier for the designer, it would certainly make life much harder for the implementer of this modeler. The problem is validity. The modeler would have to make sure that the chosen features were consistent, a difficult task in general. For example, if someone defined a block with a hole, the modeler would have to make sure that the hole was not too close to the side so that it would break through. The bigger the collection of features, the more checking that would have to be done. Roller ([Roll95]) discusses designing with constrained parameterized features.

There are two ways to deal with feature definitions in a design-by-feature system, procedural or declarative, although these can be combined. In the procedural approach, features are defined by a collection of procedures in the programming language of the system. In the declarative approach a feature definition consists of a collection of constraint specifications, rules, or predicate logic statements. Satisfaction

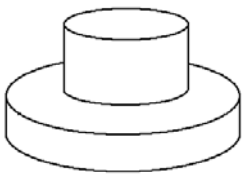


Figure 5.33. Ambiguous features: boss on disk or flange on cylinder?

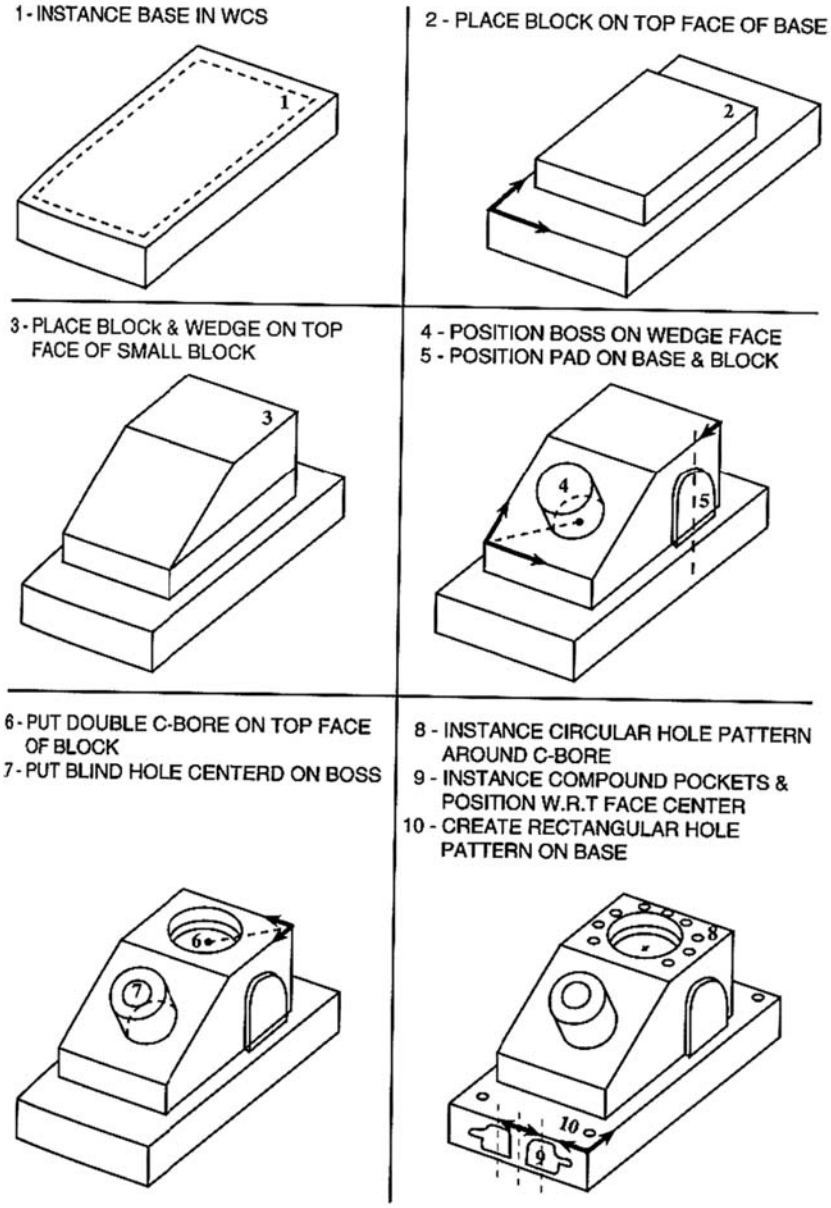


Figure 5.34. Designing with features ([ShaM95]). (This material is used by permission of John Wiley & Sons, Inc. from Parametric and Feature-Based CAD/CAM, Shah 1995; John Wiley & Sons, Inc.)

of the declarations is accomplished by a constraint satisfaction algorithm, an inference engine, or unification, respectively. Three basic approaches are used for solving the constraint problem: *constraint graphs* where the nodes are geometric entities and the edges are the constraints, logical assertions, or algebraic equations that are expressed symbolically and solved symbolically. None of the approaches are easy. See [LiON02] for an example of feature mapping in a design-by-feature system.

In conclusion, many modelers are now supporting features. Overall, b-rep modelers, in particular those that support nonmanifolds, seem better suited to the feature recognition task than CSG modelers. The only advantage of the latter is in editing but this can also be dealt with in a b-rep modeler. For example, consider an object with slots of varying lengths. Changing the length of the slots may cause them to intersect. With a CSG representation the history of any changes can be maintained relatively easily, whereas with a boundary representation such changes might cause radical changes in the relationships of facets. [Prat87b] mentions the following feature-specific advantages of a boundary representation:

(1) Features are usually best described in terms of faces, edges, etc.
(2) Dimensioning and tolerancing of features need these low-level entities.
(3) CSG representations can be ambiguous. See Figure 5.33 again.
(4) Local operations for feature manipulation are available in the design stage.

Feature recognition algorithms for b-rep modelers divide into two types: those that use purely surface information and those that use volume decompositions. The latter seem to be a better approach but are not as developed yet.

[Prat87a] and [Prat87b] have a nice survey of work in feature based modeling from 1980 to 1987.

Finally, since there are a number of feature-based modelers, it is important that one can exchange data between them. One might also want to input some feature data to an application program. STEP (**St**andard for **e**xchange of **p**roduct data) is a set of standards that resulted from an international effort to enable this exchange. See [ShaM95] for an overview and additional references.

5.7 Functions and Algorithms

In addition to modeling objects, modelers must also be able to perform a variety of operations on the objects that they have modeled. As indicated in the introduction, modeling involves modeling maps as well as objects. Here is a sample of some queries users may want to make and actions they may want to perform:

(1) Find physical propertie of objects:	center of mass, moments of inertia, area, volume, . . .
(2) Find geometrical properties of objects:	distances between objects, collision detection, intersections and other Boolean set operations, . . .
(3) Perform geometrical operations:	rigid motions, . . .
(4) Numerical control related operations:	milling, lathing, . . .

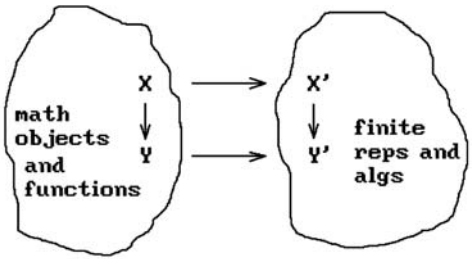


Figure 5.35. Relationship between functions and algorithms.

In a modeler mathematical functions get replaced by algorithms. We want a commutative diagram as shown in Figure 5.35. Some issues that arise are:

(1) A "correct" algorithm must perform the "correct" action on all **valid** inputs. This is a serious issue in the context of computers because of round-off errors. For example, finding the intersection of two rectangles should work if they only touch along a face.
(2) If an input to an algorithm is meaningless or invalid the algorithm should

    (a) inform the user, and
    (b) still return some answer which will not cause the system to crash

(3) An algorithm should be efficient. If objects have several representations it should be smart enough to pick the best one for its task.
(4) An algorithm should be as general as possible to allow extensions.

We can see from this discussion that modelers must deal with many tasks that fall into the field of computational geometry, which deals with finding geometric algorithms and analyzing them. In this regard we should mention the relatively new but growing area of research into genetic algorithms. These algorithms are a class of search algorithms that are proving to be very useful in the complex optimization problems encountered by modelers. An overview of this topic can be found in [RenE03].

5.8 Data Structures

5.8.1 Data Structures for Boundary Representations

Anyone interested in implementing a geometric modeling program will quickly face the problem of choosing appropriate data structures for the geometry that is to be modeled. As usual, one wants data structures that will be efficient in both space and execution times of algorithms. This section will briefly look at this issue in the case where we use a boundary representation for linear polyhedra. The next section will look at what one can do in the case of volume rendering.

Rendering linear polyhedra, or cell complexes in general, involves two parts: the abstract structure of the complex and the actual data. To describe the structure means describing various adjacency relations between the cells of the space.

Definition. A d-dimensional cell is said to be *adjacent* to an e-dimensional cell if

(1) $d \neq e$ and one is contained in the other,
(2) $d = e > 0$, and they have a (d–1)-dimensional cell in common, or
(3) $d = e = 0$, and they are the two ends of some 1-dimensional cell (edge).

In our discussion here we shall restrict ourselves to two-dimensional complexes. Algorithms dealing with such complexes typically need adjacency information such as the set of edges of a face or the set of faces that contain a given vertex. We shall use the following notation:

notation	what it means
$x \to y$	x is adjacent to y
$x \to Y$	the set of objects of type Y adjacent to x, that is, $\{y \mid y \text{ is an object of type Y and } x \to y\}$
$X \to Y$	the sets $\{x \to Y \mid x \text{ is an object of type X}\}$

In the context of this notation, capital letters such as X and Y will be one of the types V (vertex), E (edge), or F (face). $|X|$ will denote the number of objects of type X. We shall refer to $X \to Y$ as an *adjacency relation*.

The nine possible types of adjacency information between vertices, edges, and faces are shown in Figure 5.36. If a data structure contains one of these adjacency relations explicitly, then we shall say that one has *direct access* to that information. For example, the notation $E \to V$ means that each edge has direct access to both of its vertices and $V \to V$ means that each vertex has direct access to all of the vertices adjacent to it. Call a relation $X \to X$ a *self-relation*.

Two questions which need to be addressed when choosing a data structure are:

(1) Does the data structure adequately describe the topology of the spaces that are represented? If so, then we shall say that it is *topologically adequate*.
(2) What is the complexity of determining the truth of $x \to y$ or computing some $x \to Y$ given the adjacency relations defined explicitly by the data structure.

[Weil85] has an extensive discussion of question (1). The topologically adequate adjacency relationships are identified and proved to be that. We shall not repeat those

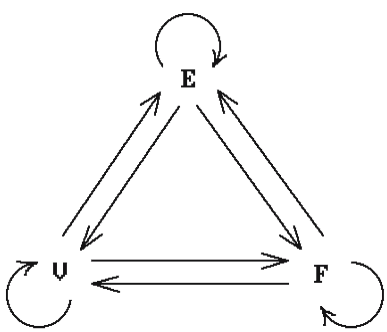


Figure 5.36. Possible face-edge-vertex adjacency data.

An algorithm for computing $Y \rightarrow X$ f rom $X \rightarrow Y$:

Let x_i , $1 \le i \le |X|$, and y_j , $1 \le j \le |Y|$, denote the objects of type X and Y, respectively.

```
for j:=1 to |Y| do
    begin
        Initialize a set X_j of objects of type X to empty;
        for i:=1 to |X| do
            begin
                One direct access gives us x_i → Y;
                If y_j ∈ (x_i → Y) then X_j = X_j ∪ { x_i };
            end
    end;
```

The collection of sets X_j now constitutes $Y \rightarrow X$.

Algorithm 5.8.1.1. Computing the inverse adjacency relation $Y \rightarrow X$.

arguments here but simply refer the interested reader to that paper. Rather, we want to concentrate on the answer to (2) and summarize the main results of [NiBl94]. It should be noted however, that all the adjacency relations that we mention later on as having the best complexity properties are also topologically adequate.

First of all, as Ni and Bloor point out, we need to distinguish between complexity in the context of a particular implementation and complexity at the abstract data structure level. [NiBl94] analyze the latter, which is implementation independent, and give answers in terms of the number of direct accesses of information and the number of set operations, such as union, intersection, etc., that are needed. A discussion of the costs involved in the context of some specific implementations, in particular, edge-based ones, can be found in [Weil85].

As an example of how one can compute the implementation independent cost of an adjacency relation that may be needed for an algorithm, suppose that a single non-self-relation $X \rightarrow Y$ is given. Algorithm 5.8.1.1 computes the inverse adjacency relation $Y \rightarrow X$. The cost of this algorithm is $|X|$ direct accesses, $|X|$ set membership tests, and at most $|X||Y|$ unions for a total of $2|X| + |X||Y|$ steps. Ni and Bloor analyze in a similar way the costs of all the other possible queries for any given set of adjacency relations. One relation, namely, the $E \rightarrow E$ relation, is treated as a special case. Sometimes data structures are used that do not store all objects adjacent to a given object. For historical reasons, due to the influential papers [Baum72] and [Baum75], the edge-edge adjacency relation has come to denote a more restricted relation in that only two of the edges adjacent to a given edge are listed.

Baumgart's Winged Edge Representation. In this representation each face is assumed to be bounded by a set of disjoint edge cycles. One of these is the outside

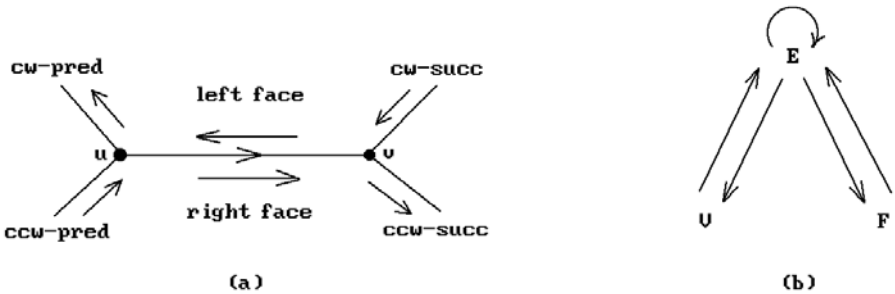


Figure 5.37. Winged edge representation.

boundary, and the others are holes in the face. In terms of implementing this, one uses a face table that consists of a list of edges where one has taken one edge from each edge cycle. Each vertex belongs to a circularly ordered set of edges. These sets are represented by one edge that is chosen from each. For each edge one lists

(1) the incident vertices,
(2) the left and right face,
(3) the preceding and succeeding edge in clockwise (cw) order with respect to the exterior of the solid, and
(4) the preceding and succeeding edge in counter-clockwise (ccw) order with respect to the exterior of the solid

See Figure 5.37(a). Figure 5.37(b) shows the relations which are involved, although $V \to E$ and $F \to E$ are only partial relations in this case.

Weiler ([Weil85]) showed that although the Baumgart structure represents the topology completely, certain algorithms for querying the data structure became complicated if self-looping edges were allowed (the two endpoints are the same point). He defined three improvements. One added a little to the structure and the other two split the edge structure into two.

Returning to [NiBl94], one finds the following conclusions:

(1) The best single relations are $V \to E$ and $F \to E$. These relations are also adequate to describe the topology completely provided that the sets $\{E_j\}$ are ordered in a circularly coherent manner.
(2) When one is given a pair of adjacency relations, this is the first time that one can, with an appropriate choice, answer all possible adjacency queries. The best pair of relations is $V \to E$ and $F \to E$.
(3) The two adjacency relation combinations shown in Figure 5.38(a) are the best combinations when one uses three adjacency relations.
(4) Figure 5.38(b) shows the best combination of four adjacency relations. This relation was discovered by [WooT85].
(5) The winged data structure shown in Figure 5.37(b) and some of its variants are worse than the combination in Figure 5.38(b).

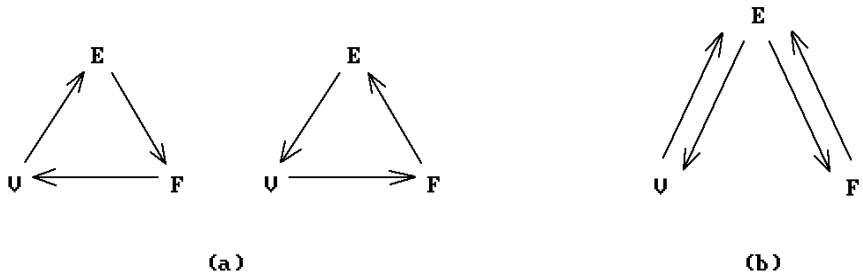


Figure 5.38. Optimal combinations of adjacency relations.

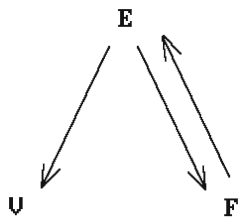


Figure 5.39. Another useful adjacency relation.

See [NiBl94] for additional observations. The authors point out that in some situations, the face-edge-vertex structure shown in Figure 5.39 is one worth considering because the algorithms used with it are simpler than the corresponding ones one would use with the related winged-edge representation.

Of course, as our final observation, in general the way one chooses a data structure for an algorithm is by **first** seeing what operations are needed by the algorithm and **then** choosing an optimal data structure for these operations. In the discussion above we evaluated data structures in terms of efficiency with respect to **all** possible adjacency queries which were possible with a given set of adjacency relations. In a particular context one may not need to answer all such queries however.

5.8.2 Data Structures for Volume Modeling

Encoding techniques based on tree structures have been used to cut down on the amount of data one needs when objects are represented by pixels or voxels. The recursive step in the basic algorithm for the general (n-dimensional) volume case is the following:

(1) If the volume is empty or completely covered by the object, then no further subdivision is necessary. Mark the volume as EMPTY or FULL, respectively.
(2) Otherwise, subdivide the volume and recursively repeat these two steps on each of the subvolumes.

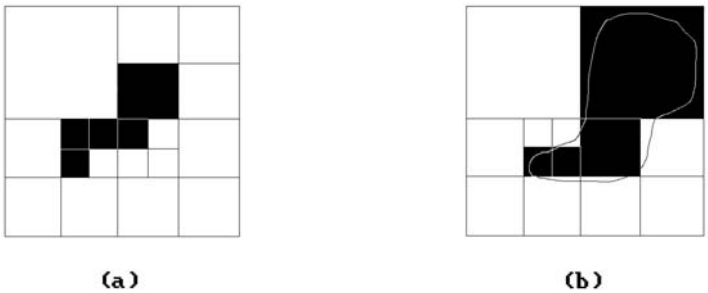


Figure 5.40. Quadtree examples.

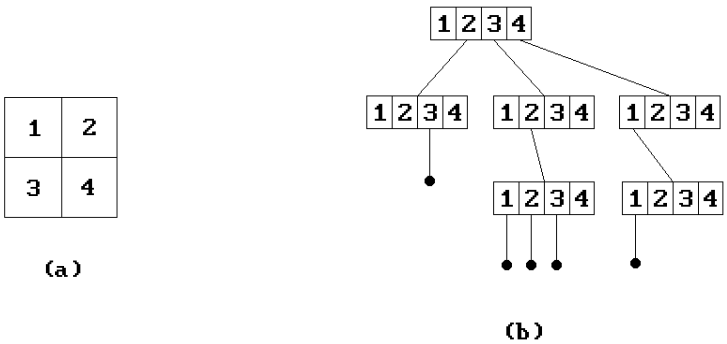


Figure 5.41. Quadtree structure.

The binary nature of the algorithm suggests a binary tree structure, but this is not as efficient as the quadtree in the two-dimensional case and octree in the three-dimensional case.

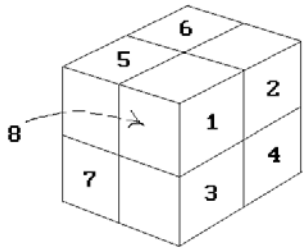
Quadtrees. Assume that a planar region is contained in a rectangle. For example, consider the shaded region **R** in Figure 5.40(a). Now change the general algorithm above to the following:

(1) If the region covers the current rectangle or is disjoint from it, then stop subdividing and return that fact; otherwise,
(2) divide the rectangle into four equal parts and repeat these two steps for each of the subrectangles.

For the region **R** in Figure 5.40(a) we shall end up with a subdivision as indicated in the figure. The region can therefore be represented by a tree structure, called a *quadtree*, where each node has up to four nonempty subtrees. If we label the four subrectangles of a rectangle as indicated in Figure 5.41(a), then the Figure 5.41(b) shows the tree structure for **R**.

Quadtrees can also be used to represent curved regions as shown in Figure 5.40(b) by modifying the criteria for when one quits subdividing in one of two ways:

Figure 5.42. Octree subdivision.



(1) We could specify a cutoff depth below which we do not subdivide, or
(2) rather than requiring that the region either misses or covers a rectangle completely, we can quit if it "almost" misses or quits, that is, one uses thresholds on the percentage of area covered.

Octrees. Octrees are the three-dimensional generalization of quadtrees. In the case of octrees we divide a cube into eight octants. See Figure 5.42. We can then encode objects with trees where each node has up to eight nonempty subtrees. Octrees are generated for objects analogous to how quadtrees are generated.

One can show that the number of nodes in a quadtree or octree are typically proportional to the size of the object's boundary, although it is possible to create some worse cases. The intuitively obvious reason for that is that the only time one needs to subdivide is when a cell is crossed by the boundary.

The quadtree and octree representations for objects have many nice properties and they have been studied extensively in order to store and process them efficiently. Boolean set operations are relatively easy. One traverses both trees in parallel and takes the appropriate steps at each corresponding pair of nodes. Algorithms exist for finding a pixel or voxel or finding neighboring pixels or voxels in the tree. These are operations that one often needs. Some simple transformations, such as rotations by 90 degrees, scaling by powers of 2, and reflections are easy. Other transformations are not. Aliasing is a big problem when performing these transformations. For more details see the references in the spatial data structure section of the bibliography.

5.9 Converting Between Representations

The need for algorithms that convert from one representation to another exists not only because modelers using different representations may want to exchange geometric data but especially because modelers increasingly seem to maintain multiple representations internally. By in large, the problem seems to have only been dealt with in an ad hoc way. We begin by addressing the two classical CSG-to-b-rep and b-rep-to-CSG problems. We end with some comments about the IGES data exchange standard between modelers.

The CSG-to-b-rep problem is the boundary evaluation problem in CSG that was first studied systematically in [ReqV85]. An early algorithm for polyhedral objects can

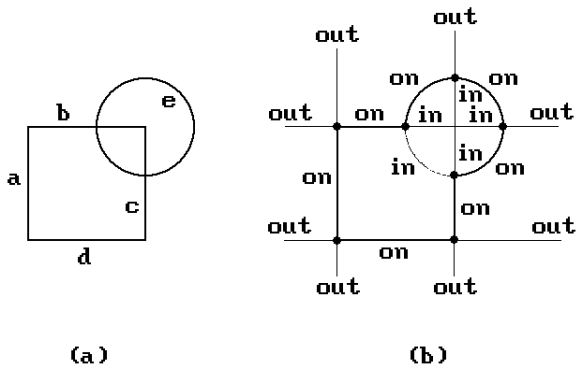


Figure 5.43. A CSG-to-b-rep conversion example.

be found in [LaTH86]. Ways to speed up the operation by only making computations for "active zones" (those parts of the CSG tree that actually affect the final answer) are described in [RosV89].

Let **X** be a CSG object. We clarify the basic approach to finding the boundary of **X** with the example in Figure 5.43(a) which is the union of a square (the intersection of four halfplanes defined by the edges **a**, **b**, **c**, and **d**) and the halfspace which is a disk. See [GHSV93].

Step 1: Determine the boundary of every CSG primitive halfspace $\mathbf{H}_i$ used in the definition of **X**. In our example this gives us four lines and a circle.

Step 2: We know that $\partial\mathbf{X} \subseteq \cup\, \partial\mathbf{H}_i$. Assuming that we have manageable definitions of the $\partial\mathbf{H}_i$, we now trim these boundaries against each other to get $\partial\mathbf{X}$. In our example, we would get four segments and three circular arcs. See Figure 5.43(b).

Step 3: To get a more compact representation one finally tries to merge adjacent faces into larger ones. In Figure 5.43(b) one would merge the three adjacent arcs into one arc.

The hard part in this algorithm is Step 2. We already mentioned some aspects of this problem in Section 5.3.3. We subdivide the step.

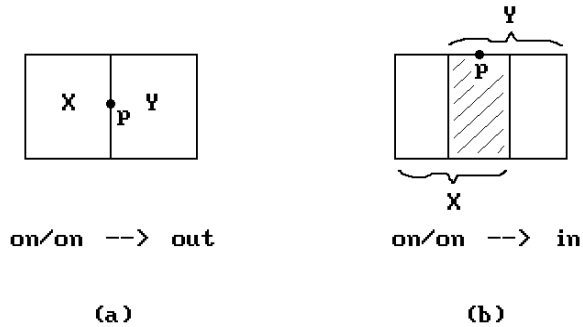
Step 2a: The points of each $\partial\mathbf{H}_i$ are divided into three subsets which consist of those points that are either **in**, **out**, or **on** with respect to **X**. See Figure 5.43(b).

Step 2b: The boundary $\partial\mathbf{X}$ consists of all those points which are **on**. This set is computed from the collection of **in**, **out**, and **on** sets using Boolean set operations together with some additional neighborhood information.

Essential to these computations is the point membership classification function, denoted by PMC. Assume that **X** is an r-set and **p** is a point. Define

$$\begin{aligned} \text{PMC}\,(\mathbf{p},\mathbf{X}) &= \mathbf{in}, && \text{if } \mathbf{p} \text{ lies in the interior of } \mathbf{X}, \\ &= \mathbf{out}, && \text{if } \mathbf{p} \text{ lies in the complement of } \mathbf{X}, \\ &= \mathbf{on}, && \text{if } \mathbf{p} \text{ lies on the boundary of } \mathbf{X}. \end{aligned}$$

Figure 5.44. An ambiguity in the on/on case for ∪*.



One computes PMC by using the CSG structure of **X**. What this means is that

(1) one has to know the value on the CSG primitives, and
(2) one needs to be able to combine values for the various set operations, that is, if **op** is one of the operations such as ∪* and ∩*, we need to express PMC (**p**,**X op Y**) in terms of PMC (**p**,**X**) and PMC (**p**,**Y**).

As an example, here is the answer, in table form, to the combine operation (2) for the operation ∩*:

X \ Y	in	on	out
in	in	on	out
on	on	on/out	out
out	out	out	out

There is a complication in the case where the point **p** is **on** both sets. Figure 5.44 shows the problem. We are unable to distinguish between the situations shown in Figure 5.44(a) and (b). To resolve this problem we need to store some neighborhood information N(**p**,**X**) whenever the point is on **X**. We therefore redefine the PMC function to

$$\begin{aligned} \text{PMC}\ (\mathbf{p},\mathbf{X}) &= \mathbf{in}, \\ &= \mathbf{out}, \\ &= (\mathbf{on},\ \text{N}(\mathbf{p},\mathbf{X})). \end{aligned}$$

Describing the neighborhoods N(**p**,**X**) can get complicated in general, depending on the kind of primitive spaces one allows. However, as an extremely simple two-dimensional example consider the case where the primitives are simply orthogonal rectangles. In that case it is possible to encode N(**p**,**X**) as a 4-tuple (a,b,c,d), where a, b, c, and d are T or F because the essential nature of a disk neighborhood of a point **p** on the boundary of **X** can be captured by considering the disk to be divided into quadrants and specifying which quadrant belongs to **X**. A quadrant is assigned a T if it belongs to **X** and an F, otherwise. See Figure 5.45. In Figure 5.44(a), we would have

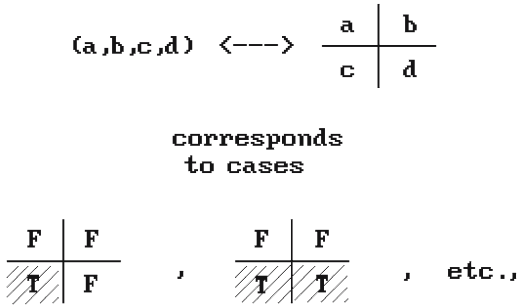


Figure 5.45. Neighborhood classification of an on/on point.

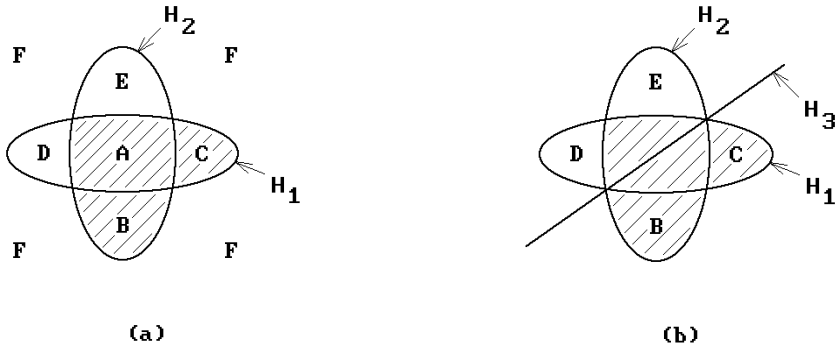


Figure 5.46. The need for separating planes.

N(**p**,**X**) = (F,F,T,T) and N(**p**,**Y**) = (T,T,F,F). In Figure 5.44(b), N(**p**,**X**) = (F,F,T,T) and N(**p**,**Y**) = (F,F,T,T). Simple Boolean operations on these representations would then determine the neighborhood of points in the **on**/**on** case. In a corresponding three-dimensional example of orthogonal blocks one would use an encoding based on octants. The reader is referred to [Tilo80] and [ReqV85] for a discussion of how one would handle more general cases.

Next, we consider the problem of converting from a b-rep to CSG, which is much more difficult than going in the opposite direction. The basic idea is to use the half-spaces associated to the faces of the b-rep to describe a CSG representation. Unfortunately, this may not work as the example in Figure 5.46(a) shows. The shaded region consisting of the three regions **A**, **B**, and **C** is our solid **X** and $\mathbf{H}_1$ and $\mathbf{H}_2$, the interiors of the horizontal and vertical ellipse, respectively, are the halfspaces associated to the faces of **X**. No CSG representation which only uses these two halfspaces will represent **X** because any space defined by such a representation that contains region **C** will also contain region **D**. We have to introduce some additional halfspaces, called *separating planes*. The separating plane and the halfplane $\mathbf{H}_3$ below it shown in Figure 5.46(b) will do the job for the space **X**. For example,

$$\mathbf{X} = (\mathbf{H}_3 \cap^* \mathbf{H}_1) \cup^* (\mathbf{H}_3 \cap^* \mathbf{H}_2) \cup^* (\mathbf{H}_1 \cap^* \mathbf{H}_2). \tag{5.5}$$

Therefore, before continuing, we need to answer the question: When can a space be described in terms of unions, intersections, and complements of halfspaces? Assume that we have a finite collection of halfspaces $H = \{\mathbf{H}_+(f_1), \mathbf{H}_+(f_2), \ldots, \mathbf{H}_+(f_k)\}$. Although there is an infinite number of ways that these spaces can be combined with the regular operators $\cup^*$, $\cap^*$, or $-^*$, the following is true:

5.9.1 Theorem. All possible combinations of the halfspaces in H using the operators $\cup^*$, $\cap^*$, or $-^*$ will generate only a finite number of regular spaces **X** and each of these can be represented by a unique expression in the form

$$\mathbf{X} = \bigcup_i{}^* \mathbf{\Pi}_i, \text{ where } \mathbf{\Pi}_i = \mathbf{h}_1 \cap^* \mathbf{h}_2 \cap^* \ldots \cap^* \mathbf{h}_k \tag{5.6}$$

and each $\mathbf{h}_j$ is either $\mathbf{H}_+(f_j)$ or $\mathbf{H}_-(f_j)$.

Proof. See [ShaV93].

As an example of this theorem, consider the space **X** in Figure 5.46(b) again. The unique decomposition of **X** guaranteed by the theorem is the one below:

$$\begin{aligned}\mathbf{X} = {} & (\mathbf{H}_3 \cap^* \mathbf{H}_1 \cap^* \mathbf{H}_2) \cup^* (\mathbf{H}_3 \cap^* \mathbf{H}_1 \cap^* c^*(\mathbf{H}_2)) \cup^* \\ & (\mathbf{H}_3 \cap^* \mathbf{H}_2 \cap^* c^*(\mathbf{H}_1)) \cup^* (\mathbf{H}_1 \cap^* \mathbf{H}_2 \cap^* c^*(\mathbf{H}_2)).\end{aligned} \tag{5.7}$$

(The decomposition in equation (5.5) does not have the right structure.)

In general, let us call any space like $\mathbf{\Pi}_i$ in equations (5.6) a *canonical intersection term* for the collection of halfspaces H. Note that the interior of every canonical intersection term is the intersection of the interior of the halfspaces that defined that canonical intersection. This leads to the main theorem about when a space admits a CSG representation based on a given set of halfspaces.

5.9.2 Theorem. Let $H = \{\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_k\}$ be a collection of halfspaces. A solid **X** with the property that $\partial\mathbf{X} \subset (\partial\mathbf{H}_1 \cup \partial\mathbf{H}_2 \cup \ldots \cup \partial\mathbf{H}_k)$ admits a CSG representation based on H if and only if the interior of every canonical intersection term based on H is either entirely contained in **X** or entirely outside of **X**.

Proof. See [ShaV91a].

Theorem 5.9.2 explains why the two halfspaces in Figure 5.46(a) were inadequate to describe the space **X**: the canonical intersection $\mathbf{H}_1 \cap^* c^*(\mathbf{H}_2)$ is half in **X** and half outside of **X**.

We clarify our discussion with another example. We desire a b-rep-to-CSG conversion for the solid **X** in Figure 5.47(a) ([GHSV93]). The b-rep of our solid **X** specifies five halfspaces associated to each face in the boundary: four halfplanes and one disk. Figure 5.47(b) shows the **in**/**out** classification of the canonical intersections for our halfspaces. We see that the regions **A** and **B** in the figure that belong to the same canonical intersection have a different **in**/**out** classification. They are both in the square and outside the disk, but one is inside our solid and the other is outside it. By Theorem 5.9.2 this means that **X** cannot be obtained from the given halfplanes and

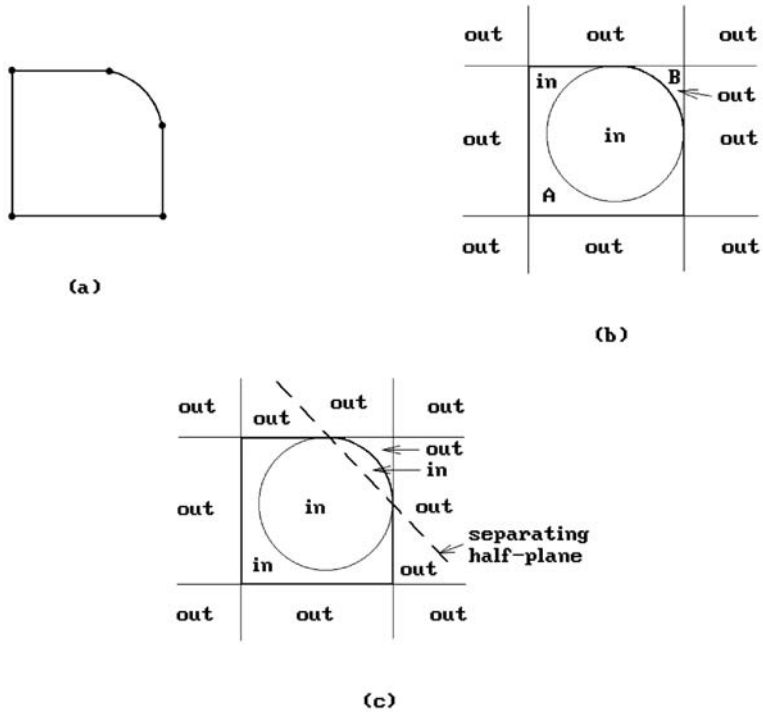


Figure 5.47. A b-rep to CSG conversion example.

disk by standard set operations. We need more halfspaces and the separating halfspace shown in Figure 5.47(c) resolves our problem.

We are now ready to describe the steps in a general b-rep-to-CSG conversion.

Step 1: We use the b-rep of our solid **X** to specify halfspaces associated to each face in the boundary.

Step 2: Since the halfspaces we get from Step 1 may not be adequate to describe **X** in a CSG way, we may have to introduce additional separating halfspaces. This is the hardest part of the conversion. What we do is to compute the canonical intersections of the halfspaces derived in Step 1. They divide the whole Euclidean space into a collection of cells. These cells and their interiors do not need to be connected. (See Figures 5.46 and 5.47.) Heavy use of the point membership classification function enables us to determine the **in**/**out** classification of these components. This classification tells us whether we need separating planes and is also used to find such planes if they are needed.

Step 3: The CSG decomposition is gotten from the cells in Step 2 by taking a union of all the cells that consisted of **in** points.

Step 4: The CSG decomposition we get in this way may not be very efficient and so as a last step one may try to perform some optimization which either minimizes the number of primitives or the number of set operations.

For more details see [ShaV91a], [ShaV91b], and [ShaV93]. [GHSV93] has a nice overview of the generic representation conversion process and the general principles that are involved. Step 4 above points to one of the stumbling blocks to having a nice theory of conversions between different representations. The answer may not be well-defined. In the b-rep-to-CSG case, we do not have a clear idea of which of the many possible primitives that one can use are best.

As we have seen, converting between different representations is potentially a hard problem. There is another related problem. Two modelers may use basically the same representation scheme but have implemented them differently. This is hardly surprising since each had a different team of programmers. What we have here is an implementation conversion problem. Because there are many commercial modeling systems in use, this is a real problem since many businesses are not able to assume that all needed data will be generated internally and there is a need to transfer data from one system to another. IGES (**I**nitial **G**raphics **E**xchange **S**pecification) was developed to solve this problem and enable different systems to exchange data between themselves. To use IGES, a modeling system must have translators that convert from their representations to those of IGES and vice versa. A person wishing to transfer data from system A to system B would then first use system A's IGES translator to translate the system A data into the IGES format and write this to a file. That file would then be read by system B's IGES translator that would convert the IGES data into system B's own internal format.

IGES is not perfect and it is easy to complain about its constraints. There is another more advanced data exchange format STEP that allows for much more high-level descriptions than the relatively simple annotated geometry formats of IGES, but we refer the reader to [ShaM95] for that. Nice features of a modeling system can get lost in the translation to and from IGES. A direct translation from one system's data structure into the other's would usually be more efficient. The latter approach would therefore be the way to go in certain dedicated situations. However, it is certainly much simpler to write one translator (for IGES) than writing many (one for each external modeling system). Writing translators is a nontrivial task. Furthermore, modeling systems continue to evolve and one would have to keep updating any direct translators. IGES also continues to change with the times, but at least only one update has to be written if one uses it. The bottom line is that IGES is a cost-effective solution to the geometric data transfer problem that works. See Appendix C for a summary of how various object types are represented by IGES and the format of an IGES file.

5.10 Round-off Error and Robustness Issues

Accuracy and robustness are important issues in numerical computations. For an overview of some common pitfalls see [McCa98]. Because geometric modeling involves a great many numerical algorithms, one wants to minimize the potential of round-off errors and avoid situations where small changes can lead to radically different results. To accomplish this one must choose carefully between different algorithms and representations for objects. In this section we only scratch the surface of

this difficult subject. Currently, solutions to numerical problems tend to be targeted to specific areas. What are still needed are general solutions. In the meantime it is largely up to users to be aware of potential problems and to take steps on their own to try to avoid them.

One of the first things one learns about real numbers when programming is that one almost never tests for equality but rather asks whether numbers are close enough. Modeling systems tend to use plenty of epsilons, small positive constants that are used to define when quantities are supposed to be considered equal. Getting correct yes/no answers when one only has approximations is not easy. So often the mathematical answer to a geometric problem is short and simple, but the computer program that implements it is much longer and messy. For example, to determine whether a point in the plane lies on a line is trivial mathematically. One can simply check whether the point satisfies the equation of the line, which involves checking whether some expression equals zero. In a computer program, testing for zero would probably be too strong a test and one would be satisfied if the expression is suitably small. This might not cause any problems by itself, but errors can propagate. Treating two lines as parallel if they are only almost parallel might be all right, but if a sequence of lines are almost parallel, then the first and the last might be quite far from being parallel. Furthermore, no matter how small the epsilons, there is a potential of getting inconsistencies in the geometric database. In Figure 5.48, if the segment **AB** is close to being parallel to the top face **f** of the solid, then in the process of intersecting it with the solid, one may conclude that **AB** intersects the face **f** in the segment **CD**. On the other hand, one may get a more accurate intersection with the side face **g** and conclude in that case that the line does not intersect **CD**. This might leave an incorrect description of **AB** in the database as a composition of three segments. The author has personally known of commercial CAD systems that (at least in their initial version) would crash in some constructions that involved solids that touched each other along essentially parallel faces.

Maintaining the orthogonality of orthogonal matrices is another problem in computer graphics. Orthogonal matrices may become nonorthogonal after numerous transformations. This is a serious problem in robotics where moving joints means transforming matrices. What one needs to do here is to maintain a count of the number of transformations that have been performed and then periodically reorthogonalize the matrices. Of course, since one does not know which values are incorrect, the new matrices, although orthogonal, are probably not what they should be mathematically.

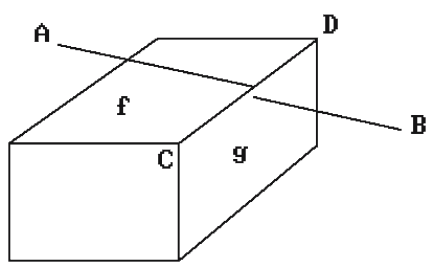


Figure 5.48. Intersection inconsistencies due to round-off errors.

Numerical analysis is clearly the first place to look for answers about accuracy in a world of approximations. A lot is known about the accuracy of the output of a computation given the accuracy of the input. For example, to get precise answers with linear problems one would have to perform computations using four to five times the precision of the initial data. In the case of quadratic problems, one would need forty to fifty times the precision. Sometimes there may be guidelines that help one improve the accuracy of results. Given the problems with floating point arithmetic, one could try other types of arithmetic.

Bounded Rational Arithmetic. This is suggested in [Hoff89] and refers to restricting numbers to being rational numbers with denominators that are bounded by a given fixed integer. One can use the method of continued fractions to find the best approximation to a real by such rationals.

Infinite Precision Arithmetic. Of course, there are substantial costs involved in this.

"Exact" Arithmetic. This does not mean the same thing as infinite precision arithmetic. The approach is described in [Fort95]. The idea is to have a fixed but relatively small upper bound on the bit-length of arithmetic operations needed to compute geometric predicates. This means that one can do integer arithmetic. Although one does not get "exact" answers, they are reliable. It is claimed that boundary-based **faceted** modelers supporting regularized set operators can be implemented with minimal overhead (compared with floating point arithmetic). Exact arithmetic works well for linear objects but has problems with smooth ones. See also [Yu92] and [CuKM99].

Interval Analysis. See Chapter 18 for a discussion of this and also [HuPY96a] and [HuPY96b].

Just knowing the accuracy is not always enough if it is worse than one would like. Geometric computations often involve many steps. Rather than worrying about accuracy only after data structures and algorithms have been chosen, one should perhaps also use accuracy as one criterion for choosing the data structures and algorithms.

One cause for the problem indicated in Figure 5.48 is that one often uses different computations to establish a common fact. The question of whether the line segment intersected the edge of the cube was answered twice – first by using the face **f** and second by using the face **g**. The problem is in the redundancy in the representation of the edge and the fact that the intersection is determined from a collection of isolated computations. If one could represent geometry in a nonredundant way, then one would be able to eliminate quite a few inconsistency problems. Furthermore, the problem shown in Figure 5.48 would be resolved if, after one found the intersection with face **f**, one would check for intersections with all the faces adjacent to **f** and then resolve any inconsistencies.

We give a brief overview of an approach to achieving robust set operations on linear polyhedra described in [HoHK89] and [Hoff89]. It involves both data structures

and algorithms. Their data structure allows for nonmanifold objects and specifies the following information about vertices, edges, and faces:

vertex: the adjacent edges and faces
edge: the bounding vertices and adjacent faces with the faces listed in a contiguous cyclical order with respect to their intersection with a plane orthogonal to the edge
face: the bounding edges and vertices as circular lists with the face to the right

Normal planes to edges and planes associated to planes are assumed to be oriented appropriately. To achieve irredundancy of information, planes are defined via equations that are oriented so that the associated normals point out of the associated solid. Vertices are defined as the intersection of the planes to which their adjacent faces belong. Edges are oriented and defined by their bounding vertices. The entire geometry is defined by a unique set of plane equations.

Since all Boolean set operations can be defined by the intersection and complement operation, the interesting operation is intersection. We consider the simplest, but most important, case of finding the intersection of two solids **X** and **Y** with connected boundaries. The initial overall strategy is

(1) If no pair of faces from **X** and **Y** intersect, check if one solid is contained in the other and quit.
(2) Intersect the boundaries of **X** and **Y**. For each face **f** of **X** that intersects **Y** find the cross-section of **Y** with respect to the plane of **f**. Determine the part of $\mathbf{X} \cap^* \mathbf{Y}$ contained in that cross-section. These regions will be defined by points and line segments lying in the boundary of **X**.
(3) The cells obtained in Step (2) will also lie in faces of **Y**. We use these cells to determine the subdivision of those faces of **Y**. Then using face adjacency information for **Y**, we find and add all the faces of **Y** lying in the interior of **X**.
(4) Assemble all the intersection faces into the solid $\mathbf{X} \cap^* \mathbf{Y}$.

The problem with Step (2) is that intersections are computed in isolation that can lead to the inconsistencies mentioned above, therefore, Step (2) and (3) are replaced by

(2′) For each intersecting pair of faces **f** and **g** from **X** and **Y**, respectively, determine the points and segments in which they intersect. Using three-dimensional neighborhood information for each intersection, the relevant parts are then transferred to all adjacent faces of **X** and **Y**.
(3′) Finally, those faces of either solid that are contained in the other are found using face adjacency information for the solids.

Bounding boxes are used for faces to speed up the algorithm and avoid unnecessary checks for face/face intersections along with efficient algorithms for determining whether intervals intersect. The most work takes place in Step (2). It is here that one makes sure that each element of the intersection is computed in a consistent way with respect to all adjacent cells. For intersecting faces one has to

(1) find and analyze the points and segments in the intersection,
(2) transfer the results to adjacent faces in **X** and **Y**, and
(3) link the various intersection pieces into complete face and edge descriptions.

These steps involve careful analysis of neighborhoods of cells. The six major cases arise from face/face, face/edge, face/vertex, edge/edge, edge/vertex, and vertex/vertex intersections.

The authors of [HoHK89] reported that when the algorithm we just described was implemented, very substantial improvements in robustness were realized compared with other modelers. Their test cases included such typically difficult cases as finding the intersection of two cubes, where the second is a slightly rotated version of the first. We refer the reader to that paper and [Hoff89] for additional ideas about dealing with accuracy and robustness that we do not have space to get into here. More papers on robustness can be found in [LinM96]. See also [DeSB92] and [EdaL99]. Often the problems we have talked about are caused by the fact that they are special cases or some sort of degeneracy. There is no problem determining whether two lines intersect if they are reasonably skew. Therefore, perhaps one can always arrange it so that they are or that objects are not almost touching, etc., by perturbing them slightly. This is the basis for an approach to robustness described in [EdeM90]. Objects are put into general position by a small perturbation; however, the perturbation is done symbolically. Nothing is actually ever perturbed.

Finally, because conics are such an important class of spaces in modeling, we finish this section with some facts about the robustness of some of their standard representations. Four common ways to define conic curves are:

(1) via the general quadratic equation
(2) in standard form at the origin along with a transformation
(3) via a few points and/or reals (For example, one can define an ellipse in terms of its center, major axis, and major and minor axes lengths.)
(4) via projective geometry type constructions

Which is best? It is well known that (1) is by far the worst representation. Changing coefficients even just slightly, can, in certain circumstances, lead to incorrect conclusions as to the type of the conic. According to [Wils87], (2) and (3) are the best with (3) slightly better.

5.11 Algorithmic Modeling

Sections 5.3.1–5.3.9 discussed various specific approaches to geometric modeling. This section takes a more global view and tries to identify some unifying principles behind some of the details. Specifically, the relatively recent generative modeling scheme and the natural phenomena and physically based modeling schemes are examples of what is referred to as *algorithmic* or *procedural modeling* in [GHSV93]. Algorithmic modeling refers to that part of geometric modeling where one uses algorithms to define and manipulate objects or functions rather than nonconstructive definitions.

Sometimes one has a choice, but even though the spaces we might get can be described by a function that parameterizes the space or defines it implicitly, there are reasons for why it might be useful to generate a space algorithmically:

(1) Not all spaces can be computed and sometimes, like in the case of fractals, one can only describe them by a limiting process.
(2) The geometry of a space sometimes changes over time or due to external influences and the best way to describe this may be algorithmically.
(3) Some complex geometric structures are too complicated to describe by functions.
(4) Algorithmic descriptions may give rise to added flexibility and power.

In [GHSV93] algorithmic models are divided into four main classes.

Geometry-based models: Generative modeling is an example. Its models are parameterized by properties and transformations

Functional-based models: These models are defined by functions and modified by other functions. Texture functions are an example how functions can modify geometry.

Grammar-based models: Here the structure is defined by a language and a grammar for that language. The grammars can be divided into geometric (fractals and their initiator/generator paradigm) and topological (graftals) grammars.

Physics-based models: The models are controlled by the laws of physics. See Section 5.5. Also included here are particle systems, deformable models (elastic/inelastic), and constraint systems.

For more details see [GHSV93].

Looked at abstractly, what the theory of algorithmic modeling adds to "standard" geometric modeling is a symbol generation mechanism that can be either deterministic or probabilistic. To formalize this we must generalize the notion of a Turing machine to one that can deal with continuous functions.

Although the theory of computation is a well-developed subject, the classical theory of Turing machines deals with discrete computations. Here, an analysis of the complexity of a computation takes into account the size of numbers in terms of the number of bits in their binary representation. On the other hand, when we deal with continuous geometry, as we do in geometric modeling, it would be nice to make our baseline the reals and to consider the cost of basic arithmetic operations, such as multiplication, independent of their "size." To put it another way, since geometric shapes are usually defined by parameterizations or implicitly, we would like to have a concept of computability based on reals and appropriate "computable" continuous functions. We would then like to ask the same types of questions as in the discrete Turing machine case. For example, which topological spaces are "computable" in this setting? In recent years, such a continuous theory of computation has

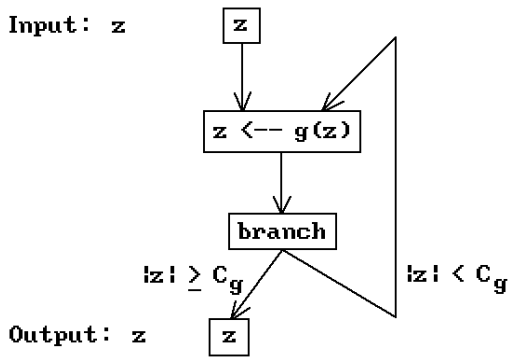


Figure 5.49. State diagram for Example 5.11.1.

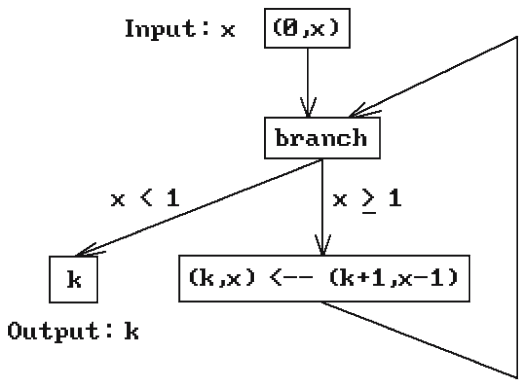


Figure 5.50. Computing the greatest integer in x.

been developed that formalizes the notion of a machine over the reals. See [BlSS89] or [GHSV93]. We would like to mention a few of the highlights of the theory in this section.

We motivate our definition of a machine over the reals with two examples from [BlSS89].

5.11.1 Example. Let $g: \mathbf{C} \to \mathbf{C}$ be a complex polynomial map. Since the highest order term of g dominates, one can show that there exists a positive constant C_g so that $|\mathbf{z}| > C_g$ implies that $|g^k(\mathbf{z})| \to \infty$ as k goes to infinity. Figure 5.49 shows a flowchart for an algorithm based on g. Right now we prefer to think of it as a state diagram for a machine M. Clearly, M will halt precisely on those inputs $\mathbf{z}$ for which $|g^k(\mathbf{z})| \to \infty$ as k goes to infinity.

5.11.2 Example. Figure 5.50 shows an algorithm that computes the greatest integer $\lfloor x \rfloor$ for $x \in \mathbf{R}$, $x \geq 0$. We again shall think of it as a state diagram of a machine that operates on a pair (k,x). To find the greatest integer in x, one inputs (0,x).

What is notable about the two examples here is the state diagram of the "machines" that carried out the associated algorithms. This is what we want to generalize now.

Let R be an ordered commutative ring with unity. Because there is no space to go into lots of details, the definitions we give here will not be as general as they could be but are aimed at the special cases R = **Z** or **R**.

Definition. A *machine M over R* consists of three sets $I = R^m$ (the *input space*), $O = R^n$ (the *output space*), and $S = R^k$ (the *state space*), along with a finite connected directed graph G whose nodes are one of four types:

Input node: There is only one node of this type and it will be denoted by 1. It has no incoming edge and only one outgoing edge to a node denoted by $\beta(1)$ and called the *next node*. Associated to this node is a linear injective map $\iota : I \to S$. One thinks of ι as just taking the input and "putting it into the machine."

Output node: These nodes have no outgoing edges. Associated to each such node o, there is a linear map $g_o : S \to O$.

Computation node: Each such node c has a unique outgoing edge to a node denoted by $\beta(c)$ and called the *next node*. Associated to the node is a polynomial map $g_c : S \to S$. If R is a field, then we can let g_c be a rational function.

Branch node: Each such node b has precisely two outgoing edges to nodes denoted by $\beta^-(b)$ and $\beta^+(b)$ called *next nodes*. Associated to b is a polynomial $h_b : S \to R$, such that $\beta^+(b)$ and $\beta^-(b)$ are associated to the conditions $h_b(x) \geq 0$ and $h_b(x) < 0$, respectively.

By expressing the definition of a machine over a ring graphically, it is not as complicated as it may sound. It leads to state diagrams just like in the Turing machine case. The two examples above are, in fact, special cases of a machine M over the reals. In Example 5.11.1, we identify the complex numbers **C** with $\mathbf{R}^2$. The input, output, and state spaces of M are all $\mathbf{R}^2$. The function ι associated to the input node is the identity map and the function at the branch node is $h(\mathbf{z}) = |\mathbf{z}|^2 - C_g$. In Example 5.11.2, the input, output, and state spaces are **R**, **R**, and $\mathbf{R}^2$, respectively. The function associated to the input node is the function from **R** to $\mathbf{R}^2$ that maps x to (0,x). At computation nodes, the associated function is $(x,y) \to (x + 1, y - 1)$. The output function is the map from $\mathbf{R}^2$ to **R** that maps (x,y) to y.

Now, given a machine M over a ring R with nodes N, define a partial map

$$G : N \times S \to N \times S$$

by $G(n,s) = (n',s')$ where

(1) G is not defined if n is an input or output node.
(2) If n is a computation node, then $n' = \beta(n)$ and $s' = g_n(s)$.
(3) If n is a branch node, then $s' = s$ and $n' = \beta^+(n)$ if $h_n(s) \geq 0$ and $n' = \beta^-(n)$ if $h_n(s) < 0$.

If we allowed rational functions in (2), then g_n may not be defined because its denominator vanishes on s. It turns out that it is easy to overcome this technical problem and so there is no loss in generality if we assume that g_n is always defined on any value where we want to evaluate it.

Definition. A *computation of length t* is a sequence

$$(1, \iota(x)), (n_1, s_1) = G(1, \iota(x)), (n_2, s_2) = G^2(1, \iota(x)), \ldots, (n_t, s_t) = G^t(1, \iota(x)), \tag{5.8}$$

where x is an element of I. The set

$$\Omega_M = \{x \in I \mid \text{there is a computation of the form (5.8) with } n_t \text{ an output node}\}$$

is called the *halting set* of M. The *input-output map*

$$\varphi_M : \Omega_M \to O$$

is defined as follows: Let $x \in \Omega_M$ and let (5.8) be the computation with $o = n_t \in O$. Then

$$\varphi_M(x) = g_o(s_t).$$

Definition. A map $f: A \subseteq R^m \to R^n$ is said to be *computable over R* if there exists a machine M over R so that $\Omega_M = A$ and $\varphi_M = f$. In this case we say that M *computes* f.

Definition. A set $A \subseteq R^m$ is *recursively enumerable* over R if $A = \Omega_M$ for some machine M over R. It is *decidable* if it and its complement are both recursively enumerable over R; otherwise it is said to be *undecidable*.

One can show that decidability of a set A is equivalent to its characteristic function being computable. Lots of interesting results about machines over a ring are proved in [BlSS89]. The ones most interesting for us here will now be mentioned.

5.11.3 Theorem. A recursively enumerable set over R is the countable union of semialgebraic sets.

Proof. See [BlSS89].

5.11.4 Corollary. The halting set of a machine over R is the countable union of disjoint semialgebraic sets. The input-output map for the machine is a piecewise polynomial map.

Since one can show that the Hausdorff-Besicovitch dimension of a semialgebraic set is an integer, we also get

5.11.5 Corollary. The halting set of a machine over R has integral Hausdorff dimension.

Corollary 5.11.4 indicates why it is reasonable to stay with semialgebraic sets in traditional geometric modeling. Corollary 5.11.5, on the other hand, shows that we do not get fractals in this way. The next result tells us something about the sets one encounters when trying to define fractals.

5.11.6 Theorem. The basis of attraction of a complex rational function $g: \mathbf{C} \to \mathbf{C}$ (which means the union of the basin of attraction of all attractive periodic points) is a recursively enumerable set over $\mathbf{R}$. In particular, it is the countable union of semialgebraic sets.

Proof. See [BlSS89]. It can be shown that g has only a finite number of attractive periodic points and that there is a polynomial h so that a point is in the basin of g if and only if $h(\mathbf{z}) < 0$ for some $\mathbf{z}$ in its orbit. The theorem is proved using this h and a machine very similar to the one in Example 5.11.1.

Not all basins of attraction are decidable. In fact, it is shown in [BlSS89] that the Julia set and most of its analogs are not recursively enumerable. On the other hand, one can compute semialgebraic set approximations to Julia sets.

5.12 Conclusions

In the 1970s and 1980s most modelers were based on either the boundary or CSG representations or both. Here is a summary of the differences between these two representations. Roughly speaking, the advantages of the boundary representation are disadvantages for the CSG representation and vice versa.

Advantages of b-reps:
(1) It is good for efficient rendering algorithms.
(2) It can handle general "free-form" or "sculptured" surfaces, that is, "curved" surfaces that typically are defined implicitly or via parameterizations.
(3) It is good for local modification of models.

Disadvantages of b-reps:
(1) It takes a lot of data structures and space to define objects.
(2) Object definitions tend to be complicated.
(3) Verification of validity is difficult.

Advantages of CSG:
(1) It is very compact.
(2) It is a natural way to define many objects and "perfect" for most mechanical engineering parts.
(3) Validity checking is "built in."

Disadvantages of CSG:
(1) It is not good for rendering because one needs a separate boundary evaluation algorithm.
(2) It may not be easy to define the motions that place objects in position for the Boolean set operations.
(3) It is impractical to use for sculptured surfaces or solids bounded by such surfaces except in the most

trivial cases. With the typical set of primitives, the best that one could do for such objects, as for example an airplane wing, is get a **very** inefficient approximation.

(4) The definition of what is a "face" is more complicated.

Some modelers were hybrid systems that used the best features of boundary and CSG representations. In fact, the user interfaces for modelers are reasonably standard and hide the actual representation that is used. It is like with a database program where the user usually does not really know (or care) whether it is truly relational or not. The only way one might get an inkling of the representation on which a modeler is based is by the speed and ease of completing certain queries. For example, boundary representations have an easier time with queries that deal with faces. A hybrid system does have problems however:

(1) It must be able to convert between the different representation and the b-rep-to-CSG conversion is very hard.
(2) It must maintain consistency between representations. This limits its coverage. For example, if a b-rep object came from a CSG representation and one modifies it using a parametric surface for blending, the original CSG structure can probably not be kept consistent with it. See [ShaV95].

Initially, the typical operators in b-rep modelers were the set operations basic to CSG, but gradually more and more operations were introduced into modelers, operations, such as local blending operations, that were not easy to implement in a CSG modeler. This has caused pure CSG modelers to disappear, probably also because of the many advantages to using spline surfaces, especially NURBS surfaces, and the fact that there is no general b-rep-to-CSG algorithm. The result is that most modelers are now b-rep based. Volume-based modelers will also probably become more prevalent in the future with faster computers and large amounts of memory. Nevertheless, CSG has had a fundamental impact on the way that one views geometric modeling. CSG can be viewed as an abstract description of objects and so, whether or not a modeler is based on it, the user interfaces will continue to support it. It should be pointed out that the parametric modeling systems that developed did not entirely avoid the problems found in the dual b-rep/csg-rep systems. When a slot in the middle of a block is moved to the edge of the block, it and the associated blend will disappear. Shapiro and Vossler ([ShaV95]) argue that such difficulties are caused by the fact that the concept "parametric family" is not well-defined. A designer may not be able to predict whether certain parameterizations will remain valid throughout the design process. A lot more work needs to be done in this area if one wants a design process that does not require human intervention in the parametric structure of an object.

A modeler's ability to edit a model is extremely important to a user. Finding a good way to do this is the basis of a lot of current research. We briefly discussed the medial axis representation. Another approach called Erep is described in [GHSV93]. Its goal is to be an editable, high-level textual representation for feature based solid modeling. It is a representation that is independent of the underlying modeler.

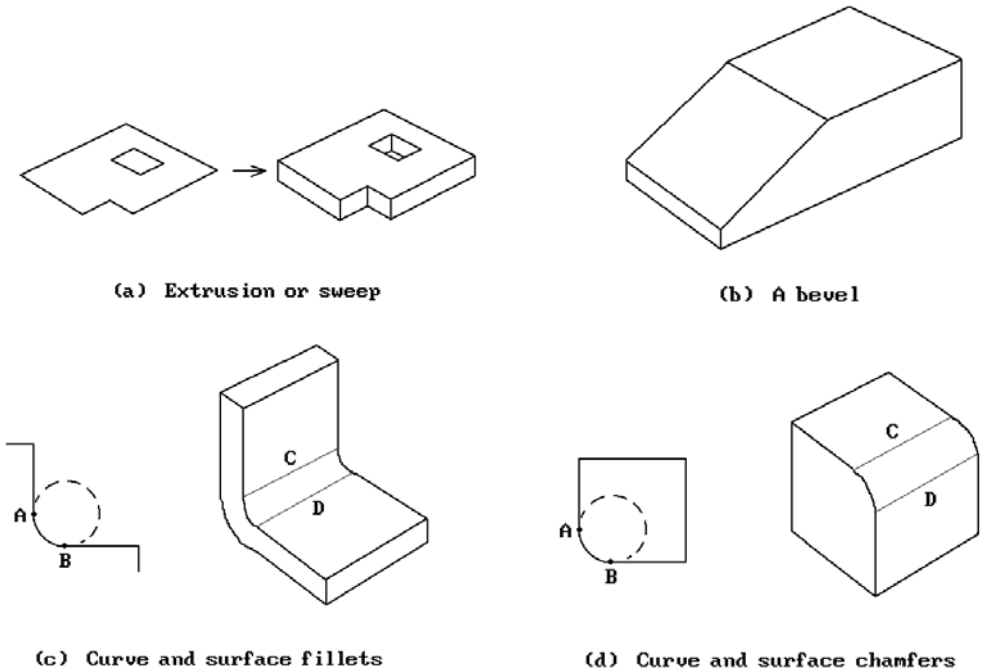


Figure 5.51. Various types of edits.

Editing involves performing operations on models. Below are some typical local operations supported by modeling systems:

(1) extrusions, or more generally sweeps (Figure 5.51(a))
(2) beveling or defining wedges (Figure 5.51(b))
(3) blending (see Section 15.6), which is a general operation that involves finding blending curves or surfaces.

A *blending curve* is one that smoothly connects the endpoints of two curves.
A *blending surface* is one that smoothly connects two surfaces along two given curves, one in each surface, and meets the surfaces tangentially at those curves.
Fillets are a special case of blending. See Figure 5.51(c). Filleting refers to a "rounding" along an edge of a **concave** corner. This could mean, e.g., matching a circle to a corner which is tangent to the edges.
Chamfering refers to "rounding" at a vertex or along a **convex** edge. See Figure 5.51(d).
The term "rounding" itself has sometimes been used to mean chamfering.

(4) local deformations of an edge or a face
(5) undoing one or more operations to restore a model to a previous state
(6) skinning (see Section 14.7)

(7) transforming shapes by scaling, mirroring, rotating, translating, . . .

In addition to the geometric operations, commercial modeling systems must also have the ability to annotate the geometry so that users can produce standard mechanical drawings. As a matter of fact, dimensioning and tolerancing is not an afterthought, but an important aspect of the total design and manufacturing process. Dimensions specify the "nominal" or perfect geometry of an object. Since one can never create the perfect object, one needs to be able to specify tolerances within which it would be considered acceptable. Other annotations relate to the type of material of an object and how it is to be rendered – in what color, etc. The representation schemes described in this chapter dealt with dimensions. The representation problem for tolerances is much harder. Coming up with a rigorous theoretical foundation for what humans have been doing manually for hundreds of years has been a nontrivial task and many problems remain as one moves towards the goal of automating annotations. See [Just92] for a brief overview of this topic.

Another property that can distinguish modelers is whether they use exact or faceted representations of objects in computations. Many modelers use a faceted representation for display purposes because the display hardware has been optimized to deal with that. The display itself is not the issue here, but rather, whether an algorithm, such as one that determines if two objects intersect, uses a faceted or exact representation of these objects. Finding the intersection of linear spaces is much easier than doing this for curved spaces and so many modelers did this in the 1980s. There are potentially serious problems with this however. To give a two-dimensional example, consider what happens when slightly overlapping circles are approximated by polygons. If they are rotated, an intersection algorithm would sometimes find an intersection and sometimes not, like in Figure 5.52. An incorrect determination of this type could cause problems if passed on to another operation.

This chapter has presented an overview of the evolution of geometric modeling. Although we have come a long way, the current state of the field is far from satisfactory. Attempts have been made to develop theoretical foundations so that one can talk about issues in a rigorous way, but by in large the field is still advancing in an ad hoc way. Some specific defects are:

(1) R-sets may be inadequate. One may want to be able to represent nonmanifold solids with dangling edges or faces. Simply enlarging the domain would still leave unanswered the question of which operations to allow and what they would preserve. See [GHSV93].
(2) In practice the definition of a representation scheme r is rather ad hoc. Usually only r^{-1} is well-defined and r itself is not. It is hard to compare different representation schemes. For example, when converting from a boundary to a CSG

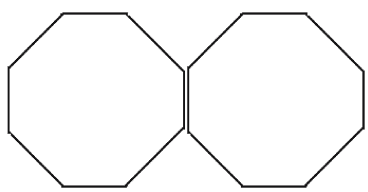


Figure 5.52. An intersection problem with faceted circles.

representation, which CSG representation of the object does one want? Which is the "correct" one? Ad hoc answers are not good enough. See [Shap91] and [GHSV93].

(3) A theoretical foundation for operations on objects is lacking or not well integrated into the representation scheme concept.

From a very global point of view however, a basic bottleneck is computer power. The fact is that (at least in some important aspects) there is a great body of known mathematics about geometry and topology that is central to doing geometric modeling "right" and which is simply waiting for the time that computers are fast enough to make dealing with it feasible. It is easy to see this relationship between progress in geometric modeling and the development of speedier computers. One simply has to look at the fantastic advances in rendering photorealistic images. This is not to say that no innovations are needed. The use of computers has brought along with it a host of additional problems that have to be solved while at the same time creating opportunities for finding new ways to understanding. An example of the former is the fact that computers do not have infinite precision arithmetic so that algorithms that are mathematically simple become very tricky to implement when one has to worry about round-off errors. An example of the latter is the ability to use computers to visualize data in ways that was not possible before. This by itself can lead to advances in knowledge. Coming up with good user interfaces is also a great challenge. Nevertheless, if computers could just deal with all the mathematics related to geometric modeling that is known **right now** we would be a giant step further along. Along these lines, two features that modeling systems should support but do not because the algorithms are too expensive computationally are:

(1) The ability to differentiate between objects topologically.
(One would need to implement the basic machinery of algebraic topology.)

(2) The ability to represent space and objects intrinsically.
Another aspect of this is that one should represent not only objects but the space in which they are imbedded. With the rise of volume rendering we are beginning to see some movement on that front.

These issues will be addressed again in Chapter 16. Given the certainty of rapid advances in hardware, geometric modeling has an exciting future.

Finally, here are some "philosophical" comments, alluded to before, having to do with the way that one should approach the subject of geometric modeling ideally. One of the things one learns from mathematics is that whenever one introduces a certain structure, be it that of a group, vector space, or whatever, it has always been fruitful to define maps that preserve that structure, like homomorphisms, linear transformations, and so on, respectively. The sets and maps are in a sense interchangeable. One could start with a class of maps and define a structure in terms of that which is left invariant by the maps. Furthermore, new structures and maps are usually studied by associating simpler invariants to them. A perfect example of this is the field of algebraic topology. See Chapter 7 in [AgoM05] for some simple examples of this (for example, the functor from the category of simplicial complexes and simplicial maps to the category of chain complexes and chain maps). In computer science one looks for "representations" and everything boils down to finding suitable representations

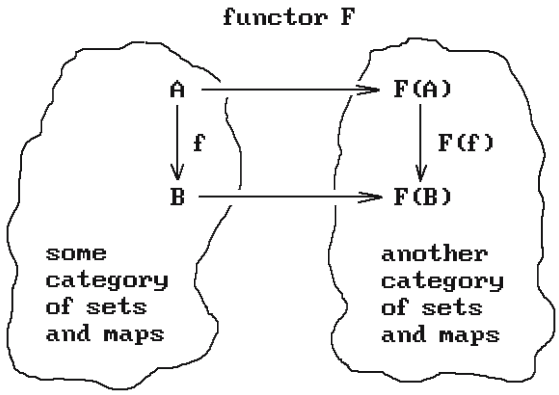


Figure 5.53. Commutative diagrams for representations.

for abstract concepts, representations that mimic the abstract concepts in every way. The point of these remarks is encapsulated by Figure 5.53. Ideally, one starts with abstract sets and transformations and then represents them by other sets and transformation in such a way that everything is preserved, meaning that one gets commutative diagrams. In the context of geometric modeling, there are objects, transformations of objects, and functions on these objects. Representations act on maps as well as sets and need to produce commutative diagrams. It would be nice if representations were what are called **functors** between **categories** but they are not. (The terms "functor" and "category" have a very definite meaning in mathematics. They help make precise the notion of commutative diagrams that we talk about at various places in this book, but that is another story that we cannot go into here.) Nevertheless it is worth reminding the reader of what one should at least strive for, or at least keep in mind, even if it is not possible to achieve in practice. Recall our discussion in Section 5.7 and Figure 5.35.

5.13 Exercises

Section 5.2

5.2.1 Explain why the unit square $[0,1] \times [0,1]$ is **not** an r-set in $\mathbf{R}^3$.

5.2.2 Prove that $r(r\mathbf{X}) = r\mathbf{X}$.

5.2.3 Give an intuitive but convincing argument for why the regularized set operators take r-sets to r-sets.

Section 5.3.3

5.3.3.1 Here is a slight variation of the usual linear polyhedron.

Definition. A set $\mathbf{X} \subseteq \mathbf{R}^n$ that can be obtained from a given collection of open linear halfplanes by a finite sequence of set operations consisting of the complement and intersection operator is called a *Nef polyhedron*.

Prove that a set $\mathbf{X} \subseteq \mathbf{R}^n$ is a Nef polyhedron if and only if it is the realization of a CSG-tree based on closed linear halfplanes and the set operations of ordinary union, intersection and difference. Because of this fact and other reasons, it has been argued that Nef polyhedra are an attractive primitive for CSG and geometric modeling in general. See [Bier95].

CHAPTER 6

Basic Geometric Modeling Tools

Prerequisites: Basic vector algebra

6.1 Introduction

This chapter describes some often-used mathematical tools and formulas in geometric modeling. The author highly recommends the *Graphics Gems* series of books to the reader (see the "Miscellaneous" section of the Bibliography). These books contain many insights into how one can make computations and algorithms more efficient.

We begin by discussing bounding objects, such as boxes, slabs, and spheres, and their uses in Section 6.2. Next, in Section 6.3 we look at tests for when a point is inside a region. Section 6.4 describes some facts that, in one way or another, are related to orientation. Some simple intersection algorithms are discussed in Section 6.5. Section 6.6 has some formulas for distances between objects, Section 6.7 has area and volume formulas, and Section 6.8 has formulas for circle constructions. We finish the chapter with some miscellaneous observations in Sections 6.9 and 6.10.

6.2 Bounding Objects and Minimax Tests

Checking for or finding intersections, as for example in clipping, visible surface determination, and collision detection, is a frequently performed task in graphics. It is also one that is usually very computation intensive, even for simple objects like polygons. Now, if complicated objects intersect, then one has to accept the fact that finding this intersection will take a lot of work. On the other hand, to make lengthy computations only to find out in the end that the objects do **not** intersect is very inefficient. One would like a quicker way to detect when objects are disjoint. A standard way to do this is to enclose objects in simpler ones and first check for intersections among these simpler objects. If they do not intersect, then by definition, the original objects will

not intersect. If they do intersect, well, then we will have to bite the bullet and check out the original objects.

Definition. A *bounding object* for an object **A** is any object **B** that contains **A**.

This section looks at some common types of bounding objects. Keep in mind the following three desirable properties that bounding objects should possess:

(1) They should be easy to compute.
(2) It should be easy to tell if they intersect.
(3) They should "fit" an object fairly closely.

The motivation behind property (3) is that we are trying to detect disjointness quickly, and the bigger that bounding objects are the more they will intersect, forcing us into lengthy computations we are trying to avoid. For an analysis as to whether bounding objects are really worth it see [SuHH99] and [ZhoS99].

Definition. A *box* in $\mathbf{R}^n$ is any subset of the form

$$[a_1,b_1]\times[a_2,b_2]\times\ldots\times[a_n,b_n],$$

where a_i, $b_i \in \mathbf{R}$. (It is convenient here to think of $[a_i,b_i]$ as **segments** and not require that $a_i \le b_i$.) A *bounding box* for an object is a box that contains the object.

Figure 6.1(a) shows a bounding box. It is easy to check for intersections of bounding boxes. For example, two segments $[a_1,b_1]$ and $[a_2,b_2]$ intersect if and only if

$$\max(\min(a_1,b_1),\min(a_2,b_2)) \le \min(\max(a_1,b_1),\max(a_2,b_2)),$$

and if they intersect, then the intersection is the **interval**

$$[\max(\min(a_1,b_1),\min(a_2,b_2)),\min(\max(a_1,b_1),\max(a_2,b_2))].$$

Notice how this formula shows that $[1,5] \cap [2,7] = [2,5]$ and $[1,5] \cap [7,9] = \phi$. In general, we have

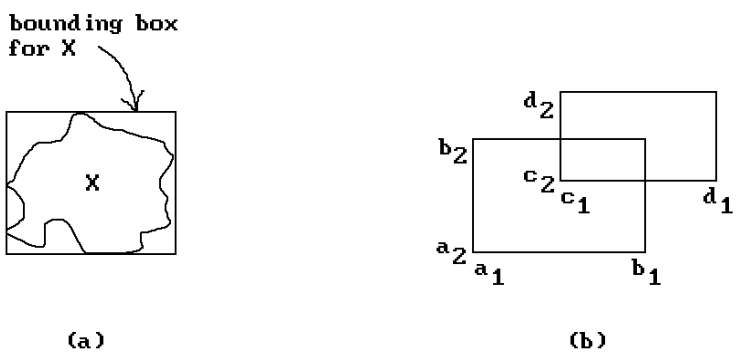


Figure 6.1. Bounding boxes.

6.2.1 Theorem. The boxes

$$\mathbf{X} = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n] \quad \text{and} \quad \mathbf{Y} = [c_1, d_1] \times [c_2, d_2] \times \ldots \times [c_n, d_n]$$

will intersect if and only if

$$lb_i = \max(\min(a_i, b_i), \min(c_i, d_i)) \le ub_i = \min(\max(a_i, b_i), \max(c_i, d_i))$$

for all i. If they intersect, then

$$\mathbf{X} \cap \mathbf{Y} = [lb_1, ub_1] \times [lb_2, ub_2] \times \ldots \times [lb_n, ub_n].$$

Proof. Easy. See Figure 6.1(b).

For obvious reasons, the intersection test in Theorem 6.2.1 is usually called the *minimax test*.

A bounding box for a polygon can be defined from the minimum and maximum values of all the coordinates of its vertices. Looking ahead, the bounding box for a spline curve or surface (defined in Chapters 11 and 12) is usually taken to be a bounding box for its control points. (This uses the important fact that splines lie in the convex hull of their control points.) Other objects may have to have their bounding boxes defined by hand.

Bounding boxes are probably the most common bounding objects that are used because it is so easy to work with them, but they are far from perfect. The main problem is that they may be a bad fit to the object. For example, the best bounding box for a line segment has that line segment as a diagonal. It is clearly easy to find lots of other objects for which rectangular boxes are a bad fit. For that reason, other bounding objects may be more appropriate for any given world of objects. There is nothing to keep us from using different types of bounding objects within one program to optimize fits. Unfortunately though, usually the better the fit, the more complicated it is to compute intersections.

A natural way to generalize bounding boxes is to allow bounding faces to be slanted and not just horizontal or vertical.

Definition. A *slab* in $\mathbf{R}^n$ is the region between two parallel hyperplanes.

Given a **bounded** set $\mathbf{X}$, a unit vector $\mathbf{n}$ determines a slab as follows: Starting arbitrarily far out on the two ends of the line through the origin with direction vector $\mathbf{n}$, slide two hyperplanes orthogonal to $\mathbf{n}$ towards the origin until they touch $\mathbf{X}$.

Definition. The region between the two touching planes is called the *slab for* $\boldsymbol{X}$ *induced by the unit vector* $\boldsymbol{n}$ and is denoted by Slab($\mathbf{X}$,$\mathbf{n}$).

See Figure 6.2(a). The two hyperplanes that bound the slab Slab($\mathbf{X}$,$\mathbf{n}$) can be written in the form

$$\mathbf{n} \bullet \mathbf{p} = d^{near} \quad \text{and} \quad \mathbf{n} \bullet \mathbf{p} = d^{far},$$

where $d^{near} \le d^{far}$. The plane that corresponds to the d^{near} will be called the *near plane* for the slab and the other one, the *far plane*. Note that if we were to project $\mathbf{X}$ orthogonally to the line through the origin with direction vector $\mathbf{n}$, then $\mathbf{X}$ would project **onto** the segment $[d^{near}\mathbf{n}, d^{far}\mathbf{n}]$.

We can use more than one vector $\mathbf{n}$.

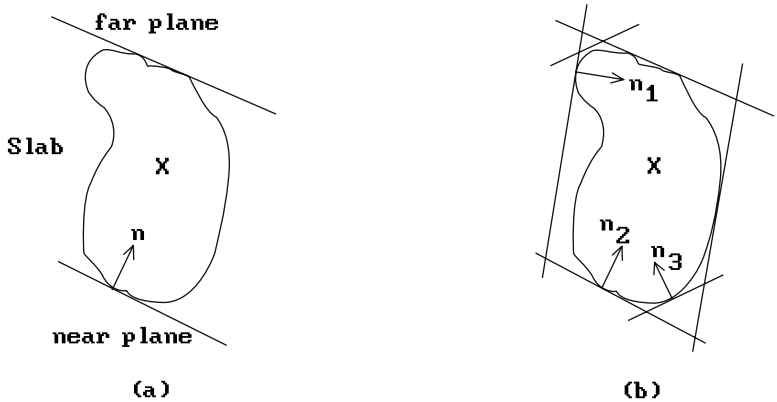


Figure 6.2. Bounding slabs.

Definition. The *generalized bounding box* for **X** determined by a fixed set, $\mathbf{n}_1$, $\mathbf{n}_2$, . . . , $\mathbf{n}_k$, of unit vectors is the intersection of the slabs that they determine. More precisely, if we denote this set by $B(\mathbf{X},\mathbf{n}_1,\mathbf{n}_2, \ldots ,\mathbf{n}_k)$, then

$$B(\mathbf{X},\mathbf{n}_1,\mathbf{n}_2,\ldots,\mathbf{n}_k) = \bigcap_{i=1}^{k} \mathrm{Slab}(\mathbf{X},\mathbf{n}_i). \tag{6.1}$$

See Figure 6.2(b). These parallelopiped-type boxes were introduced by [KayK86]. It turns out that they are not much more complicated to work with than simple boxes.

First of all, we show how the generalized bounding boxes are determined for three different types of objects in $\mathbf{R}^3$. Note that all we have to find is the d^{near} and d^{far}.

Linear Polyhedra. In this case we project all of its vertices $\mathbf{p}_j$ onto the $\mathbf{n}_i$ and use the minimum and maximum of those values, that is,

$$d_i^{near} = \min_j\{\mathbf{n}_i \bullet \mathbf{p}_j\} \quad \text{and} \quad d_i^{far} = \max_j\{\mathbf{n}_i \bullet \mathbf{p}_j\}. \tag{6.2}$$

Implicitly Defined Surfaces. Assume that a surface **S** is defined by an equation

$$f(x,y,z) = 0.$$

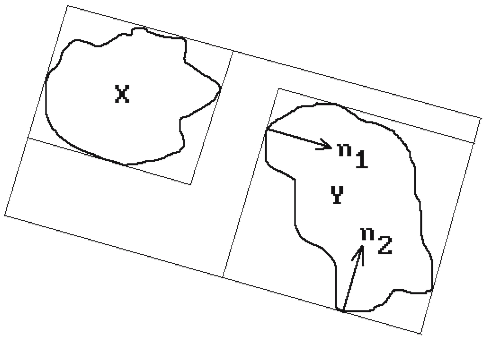
The d_i^{near} and d_i^{far} will be the minimum and maximum, respectively, of the linear function

$$g(x,y,z) = \mathbf{n}_i \bullet (x,y,z)$$

subject to the constraint above. These values can be solved for using the method of Lagrange multipliers.

Compound Objects. Assume that an object is defined by successive application of the operations of union, intersection, and difference of two objects starting with primitive objects of the type above. The d^{near} and d^{far} of the result is easily computed in

Figure 6.3. Combining bounding boxes.



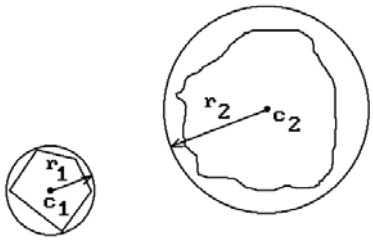


Figure 6.4. Bounding spheres.

terms of the d^{near} and d^{far} of the primitives that define it. For example, to get the d^{near} and d^{far} of the union of two objects **X** and **Y**, we need simply take the minimum of the two given d^{near} and the maximum of the two given d^{far}. See Figure 6.3.

Finding the intersection **B** of two generalized bounding boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ (defined with respect to the **same** set of normal vectors) is not hard. The formulas are really the same as those for boxes. The only difference is that instead of taking maxima and minima of coordinates (the orthogonal projections onto the standard axes $\mathbf{e}_i$) we now take maxima and minima of the orthogonal projections onto the normals $\mathbf{n}_i$, that is,

$$d_i^{near}(\mathbf{B}) = \max\left(d_i^{near}(\mathbf{B}_1), d_i^{near}(\mathbf{B}_2)\right) \quad \text{and} \quad d_i^{far}(\mathbf{B}) = \min\left(d_i^{far}(\mathbf{B}_1), d_i^{far}(\mathbf{B}_2)\right). \tag{6.3}$$

The generalized boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ are disjoint if and only if $d_i^{far}(\mathbf{B}) < d_i^{near}(\mathbf{B})$ for some i.

Other common types of bounding object are circles or spheres (we shall use the generic term "sphere" to refer to both). Such *bounding spheres* are also easy to deal with and may fit the objects better. See Figure 6.4. Two spheres S_i with centers $\mathbf{c}_i$ and radii r_i will intersect if and only if

$$|\mathbf{c}_1 - \mathbf{c}_2| \le (r_1 + r_2).$$

Finding the appropriate bounding sphere for an object is usually not hard if done by hand. Automating the process is not quite so easy. It involves finding the smallest

sphere containing a set of n points. There are optimal **O**(n) algorithms known for doing this using Voronoi diagrams. See [PreS85].

6.3 Surrounding Tests

Along with finding intersections, determining whether or not a point belongs to a two- or three-dimensional region is another common task. This section looks at some simple tests to answer the question

"Does the point **p** belong to the linear polyhedron **Q**?"

We call them "surrounding" tests because the question could also be thought of as one that asks whether a point is inside a closed curve or surface (the boundary of the polyhedron **Q** in this case). Surrounding tests fall naturally into two classes—those that deal with convex polyhedra and those that handle arbitrary polyhedra. Our discussion will also separate these two cases, but the reader should note the following: In either of these two cases it is usually a good idea to use a bounding box **B** for **Q** and first check whether **p** belongs to **B** or not. The reason is that it is very easy to check if a point belongs to a box. If **p** does not belong to **B**, then it will not belong to **Q** and there would be no need to do a lot of work to test **p** against **Q**.

We begin with tests for a convex polyhedron **Q**.

The Normals Test (Convex Q). A convex polyhedron **Q** is the intersection of a collection of halfplanes associated to the faces of **Q**. Suppose that we are given a normal vector $\mathbf{n}_i$ and vertex $\mathbf{q}_i$ for the ith face, so that the ith halfplane can be expressed in the form

$$\{\mathbf{q} \mid (\mathbf{q} - \mathbf{q}_i) \bullet \mathbf{n}_i \geq 0\}.$$

The point **p** will belong to **Q** if and only if

$$(\mathbf{p} - \mathbf{q}_i) \bullet \mathbf{n}_i \geq 0$$

for all i. As a trivial example, suppose that **Q** is the unit square $[0,1] \times [0,1]$. It has four faces. Let

$$\mathbf{n}_0 = (0,1), \mathbf{n}_1 = (-1,0), \mathbf{n}_2 = (0,-1), \mathbf{n}_3 = (1,0),$$
$$\mathbf{q}_0 = (0,0), \mathbf{q}_1 = (4,0), \mathbf{q}_2 = (4,4), \text{and } \mathbf{q}_3 = (0,4).$$

See Figure 6.5. If **p** = (x,y), then

$$\begin{aligned}
&(\mathbf{p} - \mathbf{q}_0) \bullet \mathbf{n}_0 \geq 0 && \text{means that } y \geq 0,\\
&(\mathbf{p} - \mathbf{q}_1) \bullet \mathbf{n}_1 \geq 0 && \text{means that } x \leq 4,\\
&(\mathbf{p} - \mathbf{q}_2) \bullet \mathbf{n}_2 \geq 0 && \text{means that } y \leq 4, \text{and}\\
&(\mathbf{p} - \mathbf{q}_3) \bullet \mathbf{n}_3 \geq 0 && \text{means that } x \geq 0.
\end{aligned}$$

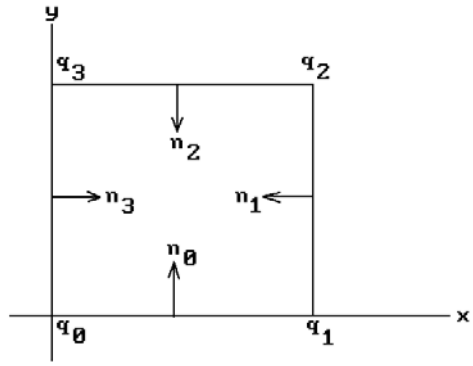


Figure 6.5. Surrounding test based on normals.

These constraints on x and y are clearly the correct ones.

The Equations Test (Convex Q). This test is really just a rewriting of the previous test. We shall describe it in the two-dimensional case. Let $\mathbf{n}_i = (a_i,b_i)$ and $c_i = -\mathbf{q}_i \cdot \mathbf{n}_i$. With this notation, the ith halfplane above is just the set

$$\{(x, y) | a_i x + b_i y + c_i \geq 0\}.$$

It follows that $\mathbf{p} = (x,y)$ will belong to the polygon if and only if

$$a_i x + b_i y + c_i \geq 0$$

for all i. What is the difference between the normals and equations test? Not much. Deciding which test to use basically depends on how data has been stored in a program. Did one store **vectors** $\mathbf{n}_i$ or coefficients a_i, b_i, and c_i?

The normals and equations tests can be generalized to a test for whether or not a convex polyhedron **P** is contained in **Q**. One simply checks if all the vertices of **P** belong to **Q**. If they do, then **P** will be contained in **Q**, otherwise not.

The Barycentric Coordinate Test (Convex Q). This test ([Bado90]) applies only to polyghedron in the plane. We think of $\mathbf{Q} = \mathbf{p}_0\mathbf{p}_1 \ldots \mathbf{p}_k$ as a fan of triangles $\mathbf{p}_0\mathbf{p}_i\mathbf{p}_{i+1}$ and then check each triangle to see whether **p** belongs to it by computing its barycentric coordinates with respect to the vertices. For example, in the case of the triangle $\Delta = \mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$, express $\mathbf{p}_0\mathbf{p}$ in the form

$$\mathbf{p}_0\mathbf{p} = a\,\mathbf{p}_0\mathbf{p}_1 + b\,\mathbf{p}_0\mathbf{p}_2.$$

The point **p** will belong to Δ if and only if $a,b \geq 0$ and $a + b \leq 1$. See Figure 6.6. If one keeps track of the number of triangles that cover the point, then one can extend the test to **nonconvex** polygons.

The Wedge Test (Convex Q). This test ([PreS85]) also applies only to polygons in the plane. One adds a central point **q** to the polygon $\mathbf{Q} = \mathbf{p}_0\mathbf{p}_1 \ldots \mathbf{p}_k$, say the centroid.

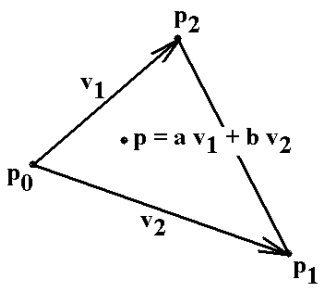


Figure 6.6. Surrounding test based on barycentric coordinates.

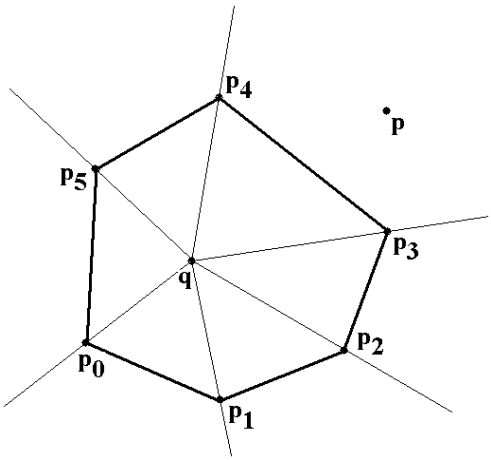


Figure 6.7. Surrounding test based on wedges.

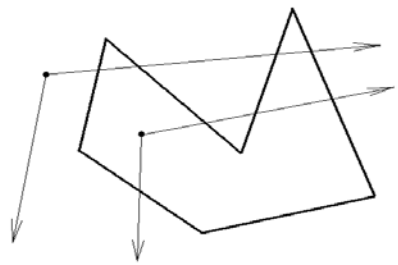
The rays from this point through all the vertices of **Q** then divide the plane into infinite wedges that are cut in half by the associated edge of **Q**. One can find the wedge that contains **p** by doing a binary search for the angle of **qp** among the angles of the vectors $\mathbf{qp}_i$. Finally one checks where **p** lies with respect to the edge of **Q** in the wedge. See Figure 6.7. Because binary search is fast, this can be a good algorithm.

Next, we look at surrounding tests for arbitrary (possibly nonconvex) polyhedra.

The Parity or Crossings Test (Arbitrary Q). For this test one checks how many times **any** ray starting at **p** will intersect the boundary of **Q**. If it intersects an even number of times, **p** is outside **Q**. If it intersects an odd number of times, **p** is inside **Q**. See Figure 6.8. For this to work though one must count an intersection twice if the ray is "tangent" to the boundary of **Q** at that point. In the two-dimensional case "tangent" means that the boundary edges containing the intersection point lie entirely to one side of the ray. In the three-dimensional case the boundary faces containing the point should lie entirely to one side of a "tangent" plane containing the ray. The polyhedron does **not** have to be convex for the parity test, but it can be made more efficient in the convex case.

The intersection tests and tests for tangency could make this a somewhat complicated test to implement without some tricks, especially in three dimensions. We

Figure 6.8. Surrounding test based on parity.



indicate a few details in the planar case. In this case the ray from **p** is usually chosen to be parallel to the x-axis with direction vector (0,1). To avoid the problem case where a vertex of **Q** actually lies on the ray, one pretends that all such points lie slightly above the ray. Next, one can easily tell if the ray intersects an edge. If the y-coordinates of the endpoints have the same signs, then the ray does not intersect. If they have opposite signs, then there will be an intersection if both x-coordinates are to the right of the x-coordinate of **p**. If the x-coordinates straddle the point, then one must compute the intersection and check on which side of **p** it lies.

The Angle Counting Test. This test, which applies only to planar polygons **Q**, is based on the topological concept of winding number that counts how many times one object "winds around" another. See Section 9.3 in [AgoM05]. The test, due to Weiler ([Weil94]), was motivated by the solution to a slightly different problem, which we shall describe first.

Let **W** be a rectangular "window" in the plane. Assume that **Q** also lies in the plane and that the boundaries of **W** and **Q** are disjoint. The question we want to answer is whether the two spaces are disjoint. Just because their boundaries are disjoint does not mean that the spaces are since one could contain the other. We present an algorithm ([Roge98]) that involves counting certain angles. In the topological case we would have to sum up infinitesimal angles, but here we do not have to be that accurate. Let us number the "octants" around **W** as shown below:

3	2	1
4		0
5	6	7

For a point **p**, let c(**p**) denote the number of the octant into which it falls. For each oriented polygon edge **e**, define the recursive angle increment function dθ(**e**) by

```
real function dθ (edge e)
begin
   real d;
   Assume that e = [p,q];
   d := c(q) − c(p);
   if d > 4 then d := d − 8;
   if d < −4 then d := d + 8;
```

```
      if |d| = 4 then
         begin
            Split e into two edges e1 and e2 at a window edge;
            d := dθ(e1) + dθ(e2)
         end;
      return d;
   end;
```

Define the *total angle* Ω by

$$\Omega = \sum_{\text{edge } \mathbf{e} \text{ of } \mathbf{Q}} d\theta(\mathbf{e}). \tag{6.4}$$

One can prove the following:

6.3.1 Theorem. The polygon **Q** will be disjoint from the window **W** if Ω is 0 and surround the window if $\Omega = \pm 8\,n$.

Figure 6.9 shows some examples. Figure 6.10 shows the need for the adjustment to $d\theta$ in the $|d\theta| = 4$ case.

We now return to our original problem about when a point **p** belongs to a polygon **Q**, Weiler's angle counting algorithm. Weiler basically takes the algorithm described above, shrinks the window **W** to a point **p**, and adjusts the algorithm accordingly. Now we classify the vertices of **Q** with respect to the quadrant into which they fall with respect to $\mathbf{p} = (x_0,y_0)$. See Figure 6.11. The quadrants are encoded via integers 0, 1, 2, or 3. Given a point (x,y), define

$$\begin{aligned}
\text{quadrant}((x,y)) &= 0 \quad \text{if} \quad x > x_0 \quad \text{and} \quad y > y_0 \\
&= 1 \quad \text{if} \quad x \le x_0 \quad \text{and} \quad y > y_0 \\
&= 2 \quad \text{if} \quad x \le x_0 \quad \text{and} \quad y \le y_0 \\
&= 3 \quad \text{if} \quad x > x_0 \quad \text{and} \quad y \le y_0.
\end{aligned}$$

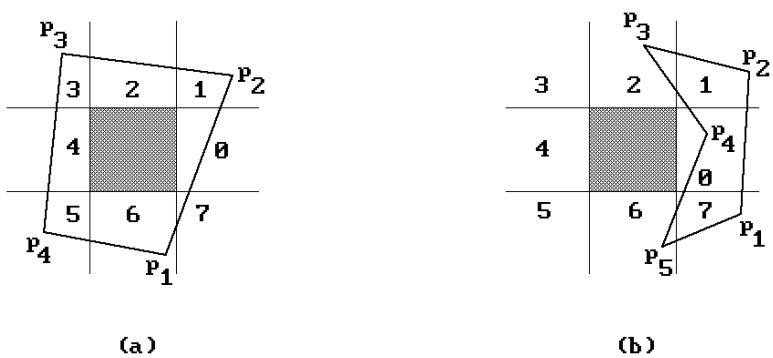


Figure 6.9. Window surrounding test based on angle counting.

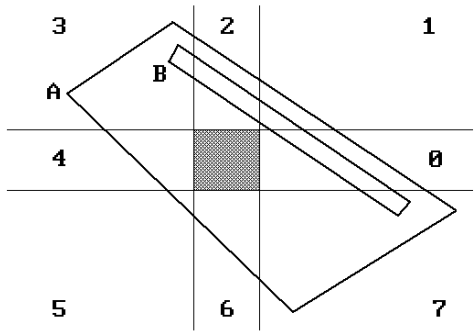


Figure 6.10. Example showing need for care in angle counting.

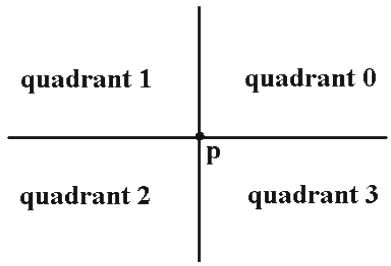


Figure 6.11. Surrounding test based on angle counting.

With our choice of inequalities, the point **p** falls into quadrant 2 and the other points on the vertical or horizontal axis in Figure 6.11 are encoded by the number of the quadrant to the left or below it, respectively. The actual program that computes the total angle Ω that is traced out by **Q** about **p** is quite simple. We start with a value of 0 and then add to Ω the difference dθ in quadrant values from one vertex to the next. The only problem is that we will again have to worry about moves between diagonal quadrants (too large of an angle). Therefore, increments will have to be adjusted using the function adjustDelta below. To compute the adjustment, one also needs a function that determines when a polygon edge passes to the left or right of **p** at y-level y_0. Here are the functions we need:

```
Assume that p = (x0,y0) and that e = [q1,q2] is an oriented edge of Q
and qi = (xi,yi).

{Find x-intercept of polygon edge e with horizontal line through y0.}
x_intercept (e) = x2 - (y2 - y0) * ((x1 - x2)/(y1 - y2))

{dθ = quadrant (q2) - quadrant (q1)}
adjustDelta (dθ,e) =
  case (dθ) of
    3     : dθ = -1; {we are crossing between quadrants 0 and 3}
    -3    : dθ = 1;  {we are crossing between quadrants 0 and 3}
    2, -2 : {we are crossing between diagonal quadrants}
            if (x_intercept (e) > x0) then dθ = - dθ;
  end;
```

The reader can find a C program for computing Ω in [Weil94]. The main result is then the following:

6.3.2 Theorem. If **p** does not lie on the boundary of **Q**, then it is outside polygon **Q** if Ω is 0 and inside if $\Omega = \pm 4$. If **p** lies on the boundary, then the algorithm will return 0 or ± 4 depending on whether the interior of **Q** was to the left or right of **p** with respect to the orientation of $\partial\mathbf{Q}$.

Weiler's angle counting algorithm extends to polygons **Q** with holes if one knows which is the outside boundary. The point **p** must be inside the outside boundary and outside the hole boundaries. There is also an extension to nonsimple polygons, such as polygons that intersect themselves.

Finally, we would to point the reader to the paper by Haines ([Hain94]) that analyzes a variety of "point-in-planar-polygon" tests with some detailed conclusions about which algorithm to use in which situation. Both the time and the amount of extra storage that is needed must be taken into account. Choosing efficient implementations of the algorithms is obviously also important. Haines has code for a parity algorithm, two versions of algorithms using normals, and an algorithm based on grids. A reader who is looking for the most efficient algorithm really needs to read Haines' paper, but a rough summary of his recommendations is the following:

No preprocessing or extra storage: use the parity test
Some preprocessing and extra storage:
- convex polygon:
  - few sides: use a normals type test on triangle fans
  - many sides: use the wedge test
- arbitrary polygon:
  - few sides: use a normals type test on triangle fans
  - many sides: use the parity test

Lots of preprocessing capability and extra storage: use a test based on grids (see [Hain94])

6.4 Orientation-Related Facts

When is a polygon **P** convex? The answer to this question is clear if **P** is defined as the intersection of halfplanes, but the more typical way that polygons are presented is via their boundary, that is, by a sequence of points. Therefore, the "real" question is "When does a sequence of points (in a plane) define a convex polygon?" One test is based on whether segments keep "turning" in one direction.

Definition. Two linearly independent vectors **u** and **v** in $\mathbf{R}^2$ are said to determine a *left turn* if the ordered basis (**u**,**v**) determines the standard orientation. Otherwise, we say that they determine a *right turn*. If the vectors are linearly dependent, then we will say that they determine **both** a left and right turn.

The notion of left or right turning vectors leads to the following intuitively obvious convexity test:

6.4.1 Theorem. (Convexity test for polygons) Assume that a planar polygon **P** is defined by a sequence of points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n, \mathbf{p}_{n+1} = \mathbf{p}_0$. The polygon **P** will be convex if and only if the vectors $\mathbf{p}_i\mathbf{p}_{i+1}$ and $\mathbf{p}_{i+1}\mathbf{p}_{i+2}$ either all determine left turns or all right turns.

Another way to express this test is to say that as one traverses the boundary of the polygon, successive edges either all make left or right turns. Alternatively, vertices of the polygon always lie on the same side of the previous edge as the one before.

There is a simple test for when two vectors **u** and **v** determine a left turn: It happens if and only if

$$\det\begin{pmatrix}\mathbf{u}\\ \mathbf{v}\end{pmatrix} = \mathbf{e}_3 \bullet (\mathbf{u} \times \mathbf{v}) \geq 0.$$

Therefore, the convexity test above is easy to program.

Next, suppose that the polygon **F** is the face of a solid **S** in $\mathbf{R}^3$ and that **p** is a point in the interior of **F**.

Definition. Let **n** be any normal for **F**. We say that **n** is an *inward-pointing normal* to **F** with respect to the solid **S**, if the segment $[\mathbf{p}, \mathbf{p} + \varepsilon\mathbf{n}]$ is entirely contained in the solid for some $\varepsilon > 0$. In that case, $-\mathbf{n}$ is called an *outward-pointing normal* for **F** with respect to **S**.

There is an easy way to determine if a normal is inward- or outward-pointing for a convex solid **S**. If **q** is any point in the interior of **S**, then **n** will be an outward-pointing normal for **F** if

$$\mathbf{n} \bullet \mathbf{qp} \geq 0,$$

otherwise it is inward-pointing.

Definition. If **P** is a polygon in a plane **X**, an *orientation* of **P** is an orientation of **X**. An *oriented polygon* is a pair (**P**,o), where **P** is a polygon and o is an orientation of **P**.

A choice of a normal vector **n** to a face **F** of a solid defines an orientation of the face. Choose an ordered basis (**u**,**v**) for the plane **X** generated by the face so that (**u**,**v**,**n**) induces the standard orientation of $\mathbf{R}^3$. The orientation of **X** induced by (**u**,**v**) is well-defined.

Definition. The orientation [**u**,**v**] of **X** is called the *orientation of **F** induced by **n***.

Conversely, an orientation o = [**u**,**v**] of the face determines the well-defined unit normal vector

$$\mathbf{n} = \frac{1}{|\mathbf{u} \times \mathbf{v}|}\mathbf{u} \times \mathbf{v}.$$

Definition. The vector **n** is called the *normal vector of **F** induced by the orientation o*.

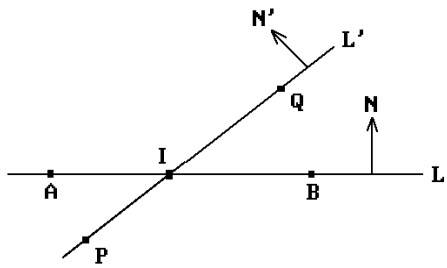


Figure 6.12. Finding the intersection of two segments.

Faces are usually defined by listing their vertices in some order. This ordering defines an orientation and normal vector in a unique way. These are called the *induced orientation* and *induced normal vector*, respectively. For example, if the face is defined by $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_k$, then the induced normal vector is $\mathbf{p}_0\mathbf{p}_1 \times \mathbf{p}_1\mathbf{p}_2$ (assuming that $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ are noncollinear). Conversely, an orientation or normal vector for a face defines a unique ordering of its vertices called the *induced ordering*.

All this extends to the case of an (n − 1)-dimensional face **F** of an n-dimensional solid **S** in $\mathbf{R}^n$, in particular, to the case n = 2 and edges of polygons in the plane.

Finally, if one has a set of either all outward- or all inward-pointing normals for a polygon, then another way to test for its convexity is to take successive cross products of the edges and their normals and see if the vectors we get all point the same way.

6.5 Simple Intersection Algorithms

6.5.1 Problem. Find the point **I** that is the intersection of two planar segments [**A**,**B**] and [**P**,**Q**].

Solution. Let **L** and **L′** be the lines determined by **A**,**B** and **P**,**Q**, respectively. See Figure 6.12. Since **I**, if it exists, must belong to both **L** and **L′**, we can express **I** in the form

$$\mathbf{I} = \mathbf{A} + s\mathbf{AB} = \mathbf{P} + t\mathbf{PQ}, \tag{6.5}$$

for some s and t. Assume that **N** and **N′** are two vectors which are perpendicular to **L** and **L′**, respectively. It follows that

$$\mathbf{A} \bullet \mathbf{N} = (\mathbf{A} + s\mathbf{AB}) \bullet \mathbf{N} = (\mathbf{P} + t\mathbf{PQ}) \bullet \mathbf{N} = \mathbf{P} \bullet \mathbf{N} + t(\mathbf{PQ} \bullet \mathbf{N}),$$

or

$$t = (\mathbf{PA} \bullet \mathbf{N}) / (\mathbf{PQ} \bullet \mathbf{N}). \tag{6.6}$$

Similarly,

$$\mathbf{P} \bullet \mathbf{N}' = (\mathbf{P} + t\mathbf{PQ}) \bullet \mathbf{N}' = (\mathbf{A} + s\mathbf{AB}) \bullet \mathbf{N}' = \mathbf{A} \bullet \mathbf{N}' + s(\mathbf{AB} \bullet \mathbf{N}').$$

or

$$s = (\mathbf{AP} \bullet \mathbf{N}')/(\mathbf{AB} \bullet \mathbf{N}'). \tag{6.7}$$

Of course, s and t may not be defined if the denominators in (6.6) and (6.7) are zero, but this happens precisely when **L** and **L′** are parallel.

This leads to the following solution to Problem 6.5.1:

Case 1. [**A**,**B**] and [**P**,**Q**] are not parallel.

In this case, use equations (6.6) and (6.7) to compute s and t. If

$$0 \le s, t \le 1,$$

then the segments [**A**,**B**] and [**P**,**Q**] intersect in the point

$$\mathbf{I} = \mathbf{P} + t\mathbf{PQ}.$$

Case 2. [**A**,**B**] and [**P**,**Q**] are parallel.

There are two steps in this case.

(1) Check to see if **L** and **L′** are the same line. This can be done by simply checking if they have a point in common. For example, if $\mathbf{AP} \cdot \mathbf{N} = 0$, then they do and **L** = **L′**. If **L** is not the same line as **L′**, then the segments will not intersect.
(2) If **L** = **L′**, then we still need to check if the segments intersect. One can reduce this to a problem of segments in **R**, because the problem is equivalent to asking if the segments $[0,|\mathbf{AB}|]$ and $[|\mathbf{AP}|,|\mathbf{AQ}|]$ intersect.

Unfortunately, although the two steps in Case 2 are a straightforward solution to the mathematical problem, implementing this in a way that produces correct results on a computer is not easy because of round-off errors.

One other issue must still be addressed before we leave the solution to Problem 6.5.1. How does one get the normal vectors **N** and **N′**. Since we are in the plane, this is easy. If **V** = (a,b) is a vector, then (–b,a) is a vector perpendicular to **V**. Finally, for a solution that is concerned with minimizing the number of arithmetic operations needed to solve the problem see [Pras91].

6.5.2 Problem. Find the intersection **I** of a line **L** and a plane **X** in $\mathbf{R}^3$. Assume that **L** is defined by two distinct points **P** and **Q** and that **X** contains a point **O** and has normal vector **N**.

Solution. Since **I** lies on **L** we can again express **I** in the form

$$\mathbf{I} = \mathbf{P} + t\mathbf{PQ}, \tag{6.8}$$

for some t. Furthermore, **OI** must be orthogonal to **N**. Therefore,

$$0 = \mathbf{OI} \bullet \mathbf{N} = (\mathbf{P} + t\mathbf{PQ} - \mathbf{O}) \bullet \mathbf{N}.$$

Solving for t leads to

$$t = \frac{\mathbf{PO} \bullet \mathbf{N}}{\mathbf{PQ} \bullet \mathbf{N}}. \tag{6.9}$$

Of course, t is only defined if **PQ** and **N** are not orthogonal, that is, **L** is not parallel to **X**. The parallel case is again a tricky case for a computer program. One needs to determine whether or not **L** lies in **X**.

6.5.3 Example. Find the intersection **I** of the line **L** containing points **P**(1,1,1) and **Q**(2,0,3) with the plane **X** which has normal vector **N** = (–1,2,0) and contains **O**(3,1,2).

Solution. Substituting into (6.9) gives

$$t = \frac{(2, 0, 1) \bullet (-1, 2, 0)}{(1, -1, 2) \bullet (-1, 2, 0)} = 2/3.$$

Therefore

$$I = \mathbf{P} + t\mathbf{PQ} = (1, 1, 1) + (2/3)(1, -1, 2) = (5/3, 1/3, 7/3).$$

Problem 6.5.2 easily generalizes to the case where **X** is a hyperplane in $\mathbf{R}^n$. It also generalizes to

6.5.4 Problem. Find the intersection of a line **L** with a k-dimensional plane **X** in $\mathbf{R}^n$. Assume that $\mathbf{N}_1$, $\mathbf{N}_2$, . . . , and $\mathbf{N}_{n-k}$ are orthogonal normal vectors for **X**. Assume also as before that **P** and **Q** are points on **L** and **O** is a point on **X**.

Solution. Define numbers t_i by

$$t_i = \frac{\mathbf{PO} \bullet \mathbf{N}_i}{\mathbf{PQ} \bullet \mathbf{N}_i}. \tag{6.10}$$

The t_i will be defined provided that **L** is not parallel to **X**, which is a special case that must be treated separately. If $t_1 = t_2 = \ldots = t_{n-k}$, then **L** intersects the plane **X** in the point

$$\mathbf{I} = \mathbf{P} + t_i\mathbf{PQ}. \tag{6.11}$$

6.5.5 Example. Find the intersection **I** of the lines **L** and **L′** in $\mathbf{R}^3$, where **L** contains the points **A**(0,3,1) and **B**(2,3,3) and **L′** contains the points **C**(2,1,1) and **D**(0,5,3).

Solution. A direction vector for **L′** is **CD** = (–2,4,2). Two orthogonal vectors normal to **L′** are $\mathbf{N}_1$ = (2,1,0) and $\mathbf{N}_2$ = (–1,2,–5). Then

$$t_1 = \frac{\mathbf{AC} \bullet \mathbf{N}_1}{\mathbf{AB} \bullet \mathbf{N}_1} = \frac{(2,-2,0) \bullet (2,1,0)}{(2,0,2) \bullet (2,1,0)} = 1/2$$

and

$$t_2 = \frac{\mathbf{AC} \bullet \mathbf{N}_2}{\mathbf{AB} \bullet \mathbf{N}_2} = \frac{(2,-2,0) \bullet (-1,2,-5)}{(2,0,2) \bullet (-1,2,-5)} = 1/2.$$

Since $t_1 = t_2$, the intersection **I** exists and

$$\mathbf{I} = (0,3,1) + (1/2)(2,0,2) = (1,3,2).$$

The reader may wonder where $\mathbf{N}_1$ and $\mathbf{N}_2$ came from. One can either assume that they were given or find two such vectors as follows: Take any two vectors orthogonal to **CD** and then apply the Gram-Schmidt orthogonalization algorithm to these. An alternate solution to this problem is to observe that **L** and **L′** intersect if and only if they lie in a plane **X**. A normal to this plane is **N** = **AB** × **CD**. Therefore the lines intersect if **C** and **D** satisfy the plane equation

$$\mathbf{N} \bullet (\mathbf{P} - \mathbf{A}) = 0.$$

To actually find the intersection, find a normal $\mathbf{N}_1$ to **L** in **X** (using, for example, the Gram-Schmidt algorithm on **N**, **AB**, **CD**). Now the problem is to find the intersection of a line **L′** with the hyperplane **L** in **X** with normal $\mathbf{N}_1$. The formula from Problem 6.5.2 applies to this variation of the intersection problem also.

We should point out that Formula 6.6.5 in Section 6.6 provides a more direct formula for the intersection of two lines in $\mathbf{R}^3$.

6.5.6 Problem. To find the intersection **I** of three planes $\mathbf{X}_i$, i = 1,2,3, which are defined by points $\mathbf{p}_i$ and normal vectors $\mathbf{N}_i$. We assume that the vectors $\mathbf{N}_i$ are linearly independent, that is, the planes are pairwise nonparallel.

Solution. One needs a simultaneous solution to the equations $\mathbf{N}_i \bullet (\mathbf{p} - \mathbf{p}_i) = 0$, i = 1,2,3. The solution is

$$\mathbf{I} = \frac{(\mathbf{p}_1 \bullet \mathbf{N}_1)(\mathbf{N}_2 \times \mathbf{N}_3) + (\mathbf{p}_2 \bullet \mathbf{N}_2)(\mathbf{N}_3 \times \mathbf{N}_1) + (\mathbf{p}_3 \bullet \mathbf{N}_3)(\mathbf{N}_1 \times \mathbf{N}_2)}{\mathbf{N}_1 \bullet (\mathbf{N}_2 \times \mathbf{N}_3)}. \tag{6.12}$$

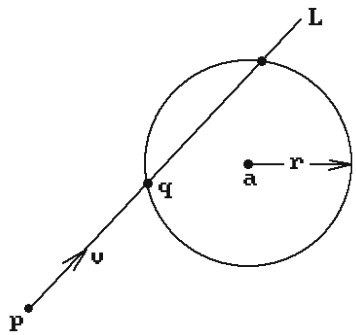


Figure 6.13. Finding the intersection of a ray and a circle.

Equation (6.12) is just a fancy way of writing the solution to this system of three equations. It is easy to check that **I** satisfies the equations. Note that $\mathbf{N}_1 \bullet (\mathbf{N}_2 \times \mathbf{N}_3)$ is just the determinant of that system.

The next two problems find the intersection of a ray with a circle. We shall use the following notation: **X** will denote a **ray** from a point **p** in a direction **v** and **L** will denote the **line** through **p** with direction vector **v**.

6.5.7 Problem. To find the intersection **q**, if any, of the ray **X** and the circle **Y** with center **a** and radius r.

Solution. See Figure 6.13. Now, **q** can be written in the form $\mathbf{q} = \mathbf{p} + t\mathbf{v}$ and so we need to solve for t satisfying

$$|\mathbf{p} + t\mathbf{v} - \mathbf{a}| = r,$$

or equivalently,

$$(\mathbf{p} - \mathbf{a} + t\mathbf{v}) \bullet (\mathbf{p} - \mathbf{a} + t\mathbf{v}) = r^2. \tag{6.13}$$

Let $A = \mathbf{v} \bullet \mathbf{v}$, $B = (\mathbf{p} - \mathbf{a}) \bullet \mathbf{v}$, and $C = (\mathbf{p} - \mathbf{a}) \bullet (\mathbf{p} - \mathbf{a}) - r^2$. Equation (6.13) can be rewritten in the form

$$At^2 + 2Bt + C = 0. \tag{6.14}$$

By the quadratic formula, the roots of (6.14) are

$$t = \frac{-B \pm \sqrt{B^2 - AC}}{A}. \tag{6.15}$$

Note that A cannot be zero because $\mathbf{v} \neq \mathbf{0}$. That leaves three cases:

Case 1: $B^2 - AC = 0$: The line **L** will intersect the circle in a single point and is tangent to it there. If $t \geq 0$, then the ray also intersects the circle at that point.
Case 2: $B^2 - AC < 0$: Both the line **L** and the ray miss the circle.
Case 3: $B^2 - AC > 0$: The line **L** intersects the circle in two points. Let t_1 and t_2 be the two distinct solutions to equation (6.13) with $t_1 < t_2$. If $t_1 \geq 0$, then the ray intersects the circle in two points. If $t_1 < 0 \leq t_2$, then the ray intersects the circle in one point. Finally, if $t_2 < 0$, then the ray misses the circle.

A special case of Problem 6.5.7 is

6.5.8 Problem. To find the intersection **q**, if any, of the ray **X** and the circle **Y** with radius r centered at the origin.

Solution. In this case we need to solve for t satisfying

$$|\mathbf{p} + t\mathbf{v}| = r,$$

or equivalently,

$$|\mathbf{v}|^2 t^2 + 2(\mathbf{p} \bullet \mathbf{v})t + |\mathbf{p}|^2 - r^2 = 0.$$

It follows that

$$t = \frac{-(\mathbf{p} \bullet \mathbf{v}) \pm \sqrt{(\mathbf{p} \bullet \mathbf{v})^2 - |\mathbf{v}|^2\left(|\mathbf{p}|^2 - r^2\right)}}{|\mathbf{v}|^2}. \tag{6.16}$$

The three cases in Problem 6.5.7 reduce to

Case 1: $(\mathbf{p} \bullet \mathbf{v})^2 - |\mathbf{v}|^2 (|\mathbf{p}|^2 - r^2) = 0$
Case 2: $(\mathbf{p} \bullet \mathbf{v})^2 - |\mathbf{v}|^2 (|\mathbf{p}|^2 - r^2) < 0$
Case 3: $(\mathbf{p} \bullet \mathbf{v})^2 - |\mathbf{v}|^2 (|\mathbf{p}|^2 - r^2) > 0$

with the same answers as before.

6.6 Distance Formulas

The next two sections describe a number of formulas that are handy for applications.

6.6.1 Formula. Let **L** be a line defined by a point **Q** and direction vector **v** and let **P** be a point. The point

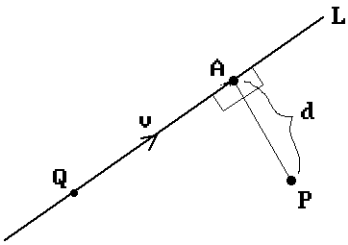


Figure 6.14. Computing the distance from a point to a line.

$$\mathbf{A} = \mathbf{Q} + \left(\mathbf{QP} \bullet \frac{\mathbf{v}}{|\mathbf{v}|}\right)\frac{\mathbf{v}}{|\mathbf{v}|} \tag{6.17}$$

is the **unique** point of **L** that is closest to **P**. If d is the distance from **P** to **L**, then

$$\mathrm{d} = \mathrm{dist}(\mathbf{P}, \mathbf{L}) = |\mathbf{PA}| = \left|\mathbf{QP} - \left(\mathbf{QP} \bullet \frac{\mathbf{v}}{|\mathbf{v}|}\right)\frac{\mathbf{v}}{|\mathbf{v}|}\right|. \tag{6.18}$$

Alternatively,

$$\mathrm{d} = \frac{|\mathbf{PQ} \times \mathbf{v}|}{|\mathbf{v}|}. \tag{6.19}$$

Proof. We shall only prove the first formula. The second is Exercise 6.6.1. Consider Figure 6.14. We seek the point **A** so that **AP** is orthogonal to **v** (Theorem 4.5.12 in [AgoM05]). The vector

$$\mathbf{w} = \left(\mathbf{QP} \bullet \frac{\mathbf{v}}{|\mathbf{v}|}\right)\frac{\mathbf{v}}{|\mathbf{v}|}$$

is the orthogonal projection of **QP** onto **L**. If **A** = **Q** + **w**, then **AP** = **QP** − **QA** = **QP** − **w** is orthogonal to **v**. Then d = |**AP**| = |**QP** − **QA**|. A solution that is concerned with minimizing the number of arithmetic operations needed to solve the problem can be found in [Morr91].

A straightforward generalization of Formula 6.6.1 is

6.6.2 Formula. The distance d from a point **P** to a plane **X** which contains a point **Q** and has orthonormal basis $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ is given by

$$\begin{aligned} \mathrm{d} &= \mathrm{dist}(\mathbf{P}, \mathbf{X}) \\ &= |\mathbf{QP} - (\mathbf{QP} \bullet \mathbf{v}_1)\mathbf{v}_1 - \ldots - (\mathbf{QP} \bullet \mathbf{v}_k)\mathbf{v}_k| \end{aligned} \tag{6.20}$$

Furthermore,

$$\mathbf{A} = \mathbf{Q} + (\mathbf{QP} \bullet \mathbf{v}_1)\mathbf{v}_1 + \ldots + (\mathbf{QP} \bullet \mathbf{v}_k)\mathbf{v}_k \tag{6.21}$$

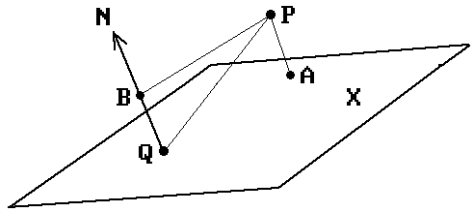


Figure 6.15. Computing the distance from a point to a plane.

is the **unique** point of **X** which is closest to **P**.

Proof. Exercise 6.6.2.

If one has a normal vector to a plane, then the formula for the distance of a point to it is much simpler.

6.6.3 Formula. The distance d from a point **P** to a plane **X** that contains the point **Q** and has normal vector **N** is given by

$$\mathrm{d} = \mathrm{dist}(\mathbf{P},\mathbf{X}) = \left|\mathbf{QP} \bullet \frac{\mathbf{N}}{|\mathbf{N}|}\right|. \tag{6.22}$$

The point

$$\mathbf{A} = \mathbf{P} - \left(\mathbf{QP} \bullet \frac{\mathbf{N}}{|\mathbf{N}|}\right)\frac{\mathbf{N}}{|\mathbf{N}|} \tag{6.23}$$

is the **unique** point of **X** that is closest to **P**.

Proof. See Figure 6.15. The vector

$$\mathbf{w} = \left(\mathbf{QP} \bullet \frac{\mathbf{N}}{|\mathbf{N}|}\right)\frac{\mathbf{N}}{|\mathbf{N}|}$$

is the orthogonal projection of **QP** onto **N**. Therefore, d = |**w**|. Define

$$\mathbf{B} = \mathbf{Q} + \left(\mathbf{QP} \bullet \frac{\mathbf{N}}{|\mathbf{N}|}\right)\frac{\mathbf{N}}{|\mathbf{N}|}.$$

Then **A** = **Q** + **BP** = **Q** + **QP** − **QB** = **P** − **w** is the unique point of **X** that is closest to **P** because **AP** is orthogonal to the plane (Theorem 4.5.12 in [AgoM05]).

Formula 6.6.3 can be restated in terms of coordinates as follows:

6.6.4 Formula. If **X** is a hyperplane in $\mathbf{R}^n$ defined by equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n + d = 0,$$

then the distance from **X** to the origin is

$$\frac{|d|}{\sqrt{a_1^2 + a_2^2 + \ldots + a_n^2}}.$$

Proof. Note that $(a_1, a_2, \ldots, a_n)$ is a normal vector to plane **X**. Therefore, the formula is basically equation (6.22) written in coordinate form. For an efficient formula that avoids square roots see [Geor92].

Two special cases of Formula 6.6.4 are worth noting. The distance from the origin to the line in the plane with equation ax + by + c = 0 is

$$\frac{|c|}{\sqrt{a^2 + b^2}}. \tag{6.24}$$

The distance from the origin to the plane in $\mathbf{R}^3$ with equation ax + by + cz + d = 0 is

$$\frac{|d|}{\sqrt{a^2 + b^2 + c^2}}. \tag{6.25}$$

6.6.5 Formula. Let $\mathbf{L}_1$ be the line defined by a point **P** and direction vector **v**. Let $\mathbf{L}_2$ be the line defined by a point **Q** and direction vector **w**. Assume that the lines are not parallel. The distance d between $\mathbf{L}_1$ and $\mathbf{L}_2$ is given by

$$d = \mathrm{dist}(\mathbf{L}_1, \mathbf{L}_2) = |\mathbf{PQ} - s\mathbf{v} + t\mathbf{w}|, \tag{6.26}$$

where

$$\begin{aligned} s &= (-(\mathbf{PQ} \bullet \mathbf{w})(\mathbf{w} \bullet \mathbf{v}) + (\mathbf{PQ} \bullet \mathbf{v})(\mathbf{w} \bullet \mathbf{w}))/D, \\ t &= ((\mathbf{PQ} \bullet \mathbf{v})(\mathbf{v} \bullet \mathbf{w}) - (\mathbf{PQ} \bullet \mathbf{w})(\mathbf{v} \bullet \mathbf{v}))/D, \text{and} \\ D &= (\mathbf{v} \bullet \mathbf{v})(\mathbf{w} \bullet \mathbf{w}) - (\mathbf{v} \bullet \mathbf{w})^2. \end{aligned}$$

Furthermore, the point **A** = **P** + s**v** on $\mathbf{L}_1$ is the unique point of $\mathbf{L}_1$ which is closest to $\mathbf{L}_2$. Similarly, the point **B** = **Q** + t**w** on $\mathbf{L}_2$ is the unique point of $\mathbf{L}_2$ which is closest to $\mathbf{L}_1$. If the lines intersect, then **A** = **B** and we have formulas for the intersection point.

Proof. See Figure 6.16. Let **A** = **P** + s**v** and **B** = **Q** + t**w** be typical points on $\mathbf{L}_1$ and $\mathbf{L}_2$, respectively. Clearly, d = d(s,t) = dist(**A**,**B**), where the vector **AB** = **PQ** − s**v** + t**w** is

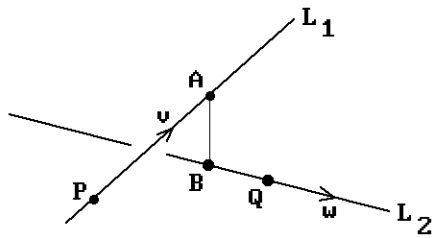


Figure 6.16. Computing the distance between two lines.

orthogonal to both **v** and **w** (Theorem 4.5.12 in [AgoM04]). Expanding the two conditions $\mathbf{AB} \bullet \mathbf{v} = 0$ and $\mathbf{AB} \bullet \mathbf{w} = 0$ reduces to the equations

$$t\mathbf{w} \bullet \mathbf{v} - s\mathbf{v} \bullet \mathbf{v} = -\mathbf{PQ} \bullet \mathbf{v}$$
$$t\mathbf{w} \bullet \mathbf{w} - s\mathbf{v} \bullet \mathbf{w} = -\mathbf{PQ} \bullet \mathbf{w}$$

with the indicated solutions. By the Cauchy-Schwarz inequality, the denominator D is zero precisely when the vectors **v** and **w** are parallel.

6.7 Area and Volume Formulas

This section contains some more useful formulas. The "proofs" of these formulas will rely on simple-minded geometric observations and will not be very rigorous. For rigorous proofs one would need to use a theory of areas and volumes. The most elegant approach would be via differential forms. See [Spiv65] or Section 4.9 in [AgoM05]. Finally, the formulas below are "mathematical" formulas. For efficient ways to compute them see [VanG95].

6.7.1 Formula. The area A of a parallelogram defined by two vectors **u** and **v** in $\mathbf{R}^2$ is given by

$$A = |\mathbf{u} \times \mathbf{v}| = \left|\det\begin{pmatrix}\mathbf{u}\\ \mathbf{v}\end{pmatrix}\right|. \tag{6.27}$$

Proof. The first equality follows from properties of the cross product and the fact that A is the product of the height of the parallelogram and the length of its base. See Figure 6.17(a). The second follows from direct computation of the cross product and the indicated determinant.

6.7.2 Formula. The volume V of a parallelopiped defined by vectors **u**, **v**, and **w** in $\mathbf{R}^3$ is given by

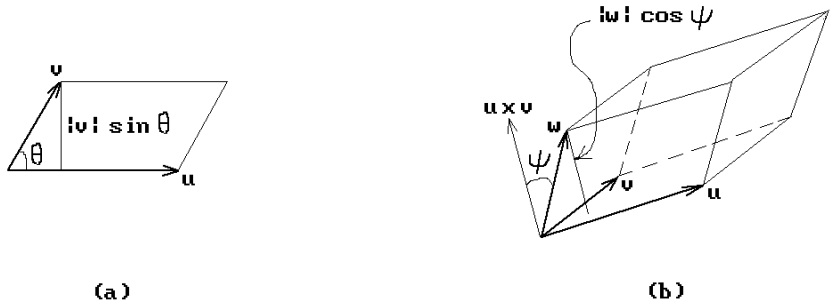


Figure 6.17. The area of a parallelogram and volume of a parallelopiped.

$$V = |(\mathbf{u} \times \mathbf{v}) \bullet \mathbf{w}| = \left| \det \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{pmatrix} \right|. \tag{6.28}$$

Proof. By a rigid motion we can arrange it so that $\mathbf{u}$ and $\mathbf{v}$ lie in the plane $\mathbf{R}^2$. The first equality follows from the fact that V is the product of the height of the parallelopiped and the area of its base. See Figure 6.17(b). The second equality is a property of the triple product.

Note that Formula 6.7.1 is a special case of Formula 6.7.2 where we let $\mathbf{w} = \mathbf{e}_3$.

6.7.3 Formula. The area A of a triangle defined by two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbf{R}^2$ is given by

$$A = (1/2)|\mathbf{u} \times \mathbf{v}| = (1/2)\left| \det \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix} \right|. \tag{6.29}$$

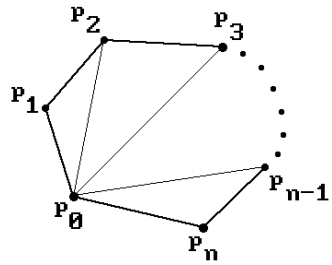
Proof. This is an immediate corollary of Formula 6.7.1. Of course, we could also use the well-known formula

$$\begin{aligned} A &= (1/2)\,(\text{length of base})\,(\text{height}) \\ &= (1/2)|\mathbf{u}| \left| \mathbf{v} - \left(\mathbf{v} \bullet \frac{\mathbf{u}}{|\mathbf{u}|} \right) \frac{\mathbf{u}}{|\mathbf{u}|} \right|. \end{aligned}$$

(The height is just the length of the orthogonal complement of $\mathbf{v}$ with respect to $\mathbf{u}$.)

6.7.4 Formula. The area A of a polygon defined by points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n, \mathbf{p}_{n+1} = \mathbf{p}_0$ in $\mathbf{R}^2$ is given by

Figure 6.18. Computing the area of a polygon.



$$A = \left| \sum_{i=2}^{n} (1/2) \det \begin{pmatrix} \mathbf{p}_0\mathbf{p}_{i-1} \\ \mathbf{p}_0\mathbf{p}_i \end{pmatrix} \right|. \tag{6.30}$$

Proof. Consider Figure 6.18 and use Formula 6.7.3. The argument also works for nonconvex polygons.

6.7.5 Formula. The signed volume V of a tetrahedron with vertices $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and $\mathbf{p}_4$ is given by

$$V = (1/6)\det \begin{pmatrix} \mathbf{p}_1 - \mathbf{p}_2 \\ \mathbf{p}_1 - \mathbf{p}_3 \\ \mathbf{p}_1 - \mathbf{p}_4 \end{pmatrix} = (1/6)\det \begin{pmatrix} p_{11} & p_{12} & p_{13} & 1 \\ p_{21} & p_{22} & p_{23} & 1 \\ p_{31} & p_{32} & p_{33} & 1 \\ p_{41} & p_{42} & p_{43} & 1 \end{pmatrix}, \tag{6.31}$$

where $\mathbf{p}_i = (p_{i1}, p_{i2}, p_{i3})$. The value for V will be positive if we order the points so that the ordered basis $(\mathbf{p}_1 - \mathbf{p}_2, \mathbf{p}_1 - \mathbf{p}_3, \mathbf{p}_1 - \mathbf{p}_4)$ induces the standard orientation of $\mathbf{R}^3$.

Proof. The first equality follows from Formula 6.7.2 and the fact that the parallelolopiped defined by $\mathbf{p}_1 - \mathbf{p}_2$, $\mathbf{p}_1 - \mathbf{p}_3$, and $\mathbf{p}_1 - \mathbf{p}_4$ can be decomposed into six tetrahedra of equal volumes. The second equality in formula (6.31) follows from basic properties of the determinant, in particular, the fact that the determinant of a matrix is unchanged if a row of the matrix is subtracted from another row.

Definition. Let $\gamma: [a,b] \to \mathbf{R}^2$ be a differentiable curve in the plane. Let $\mathbf{p} \in \mathbf{R}^2$. The set

$$\bigcup_{t \in [a,b]} [\mathbf{p}, \gamma(t)]$$

is called the *region subtended by the curve* γ *from the point* $\mathbf{p}$. See Figure 6.19(a).

6.7.6 Formula. Let $\gamma: [a,b] \to \mathbf{R}^2$, $\gamma(t) = (\gamma_1(t), \gamma_2(t))$, be a curve in the plane. The signed area A of the region subtended by the curve γ from the origin is given by

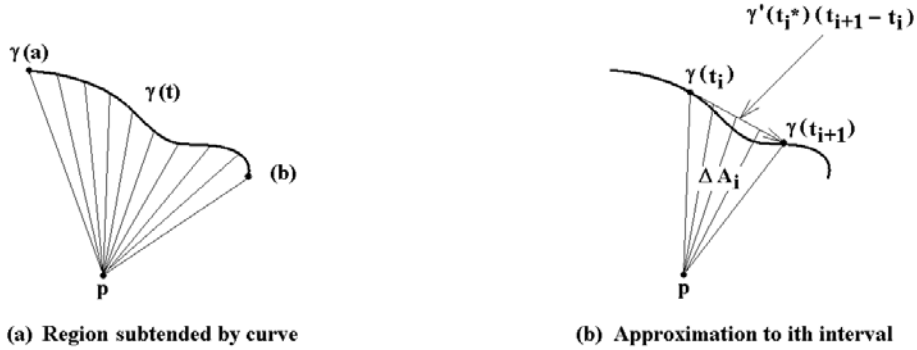


Figure 6.19. Area subtended by curve from origin.

$$A = \int_a^b (\gamma_1\gamma_2' - \gamma_1'\gamma_2). \tag{6.32}$$

Proof. See Figure 6.19. Let $a = t_0, t_1, \ldots, t_k = b$ be a partition of $[a,b]$. Let ΔA_i be the signed area of the triangle $\mathbf{0}\gamma(t_i)\gamma(t_{i+1})$. Since

$$\gamma(t_{i+1}) - \gamma(t_i) = \gamma'(t_i^*)(t_{i+1} - t_i),$$

for some $t_i^* \in [t_i, t_{i+1}]$, we get that

$$\Delta A_i = (1/2)\det\begin{pmatrix} \gamma(t_i) \\ \gamma'(t_i^*) \end{pmatrix}(t_{i+1} - t_i)$$

by Formula 6.7.3. The sign of ΔA_i will be positive if the curve is moving in a counterclockwise direction about the origin at $\gamma(t_i)$ and negative otherwise. Adding up all the ΔA_i is clearly an approximation to A and a Riemann sum that converges to the integral in the formula as the norm of the partition goes to zero.

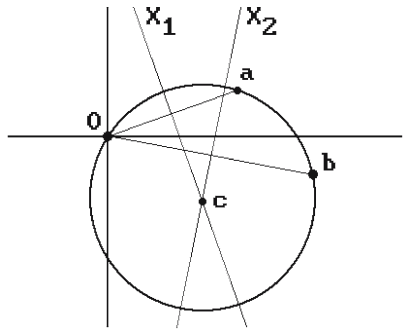
6.8 Circle Formulas

6.8.1 Formula. Let $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ be noncollinear points in $\mathbf{R}^3$. Let $\mathbf{a} = \mathbf{p}_2 - \mathbf{p}_1$ and $\mathbf{b} = \mathbf{p}_3 - \mathbf{p}_1$. The **unique** circle that contains the points $\mathbf{p}_i$ has center $\mathbf{o} = \mathbf{p}_1 + \mathbf{c}$, where

$$\mathbf{c} = \frac{(\mathbf{b}\bullet\mathbf{b})(\mathbf{a}\bullet\mathbf{a} - \mathbf{a}\bullet\mathbf{b})\mathbf{a} + (\mathbf{a}\bullet\mathbf{a})(\mathbf{b}\bullet\mathbf{b} - \mathbf{a}\bullet\mathbf{b})\mathbf{b}}{2|\mathbf{a}\times\mathbf{b}|^2}, \tag{6.32}$$

and radius r, where

Figure 6.20. Circle through three points.



$$r = \frac{|\mathbf{a}||\mathbf{b}||\mathbf{a}-\mathbf{b}|}{2|\mathbf{a}\times\mathbf{b}|}.$$

In the special case where $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ are points in the plane, these formulas can be simplified. Let $\mathbf{a} = (a_1,a_2)$, $\mathbf{b} = (b_1,b_2)$, and $\mathbf{c} = (c_1,c_2)$. Then

$$c_1 = \frac{|\mathbf{a}|^2 b_2 - |\mathbf{b}|^2 a_2}{2(a_1b_2 - b_1a_2)} \quad \text{and} \quad c_2 = \frac{|\mathbf{b}|^2 a_1 - |\mathbf{a}|^2 b_1}{2(a_1b_2 - b_1a_2)}. \tag{6.33}$$

Proof. We shall give essentially two proofs for this formula. We give a geometric argument in the general case and an algebraic one in the planar case. In either case, the solution to this problem will be easier if we first move the point $\mathbf{p}_1$ to the origin and solve the problem of finding the center $\mathbf{c}$ of the circle through the points $\mathbf{0}$, $\mathbf{a}$, and $\mathbf{b}$. Consider the following planes and their point-normal equations:

plane	equation
the plane $\mathbf{X}$ containing the points $\mathbf{0}$, $\mathbf{a}$, and $\mathbf{b}$ (and $\mathbf{c}$)	$(\mathbf{a}\times\mathbf{b})\bullet(\mathbf{x}-\mathbf{c}) = 0$
the plane $\mathbf{X}_1$ that is the perpendicular bisector of the segment $[\mathbf{0},\mathbf{a}]$	$\mathbf{a}\bullet(\mathbf{x}-(1/2)\mathbf{a}) = 0$
the plane $\mathbf{X}_2$ that is the perpendicular bisector of the segment $[\mathbf{0},\mathbf{b}]$	$\mathbf{b}\bullet(\mathbf{x}-(1/2)\mathbf{b}) = 0$

Basic geometric facts about circles imply that $\mathbf{c}$ is the intersection of these three planes $\mathbf{X}$, $\mathbf{X}_1$, and $\mathbf{X}_2$. See Figure 6.20 where we have identified $\mathbf{X}$ with the xy-plane. Letting $\mathbf{x}$ be $\mathbf{0}$ in the equation for $\mathbf{X}$ and $\mathbf{c}$ in the other two equations gives us that

$$(\mathbf{a}\times\mathbf{b})\bullet\mathbf{c} = 0, \quad \mathbf{c}\bullet\mathbf{a} = (1/2)|\mathbf{a}|^2, \quad \text{and} \quad \mathbf{c}\bullet\mathbf{b} = (1/2)|\mathbf{b}|^2.$$

Applying Formula 6.5.6 for the intersection of three planes now gives us our formula (6.32). The formula for the radius r is gotten by substituting formula (6.32) into the equation $\mathbf{r}^2 = \mathbf{c}\bullet\mathbf{c}$ and simplifying.

Next, assume that our points $\mathbf{p}_i$ lie in the xy-plane. The equation for the circle with center $\mathbf{c}$ and radius r is

$$(x-c_1)^2 + (y-c_2)^2 = r^2. \tag{6.34}$$

Since $r^2 = c_1^2 + c_2^2$, equation (6.34) reduces to

$$2xc_1 + 2yc_2 = x^2 + y^2. \tag{6.35}$$

Substituting points **a** and **b** into equation (6.35) gives two equations in two unknowns c_1 and c_2:

$$\begin{aligned} 2a_1c_1 + 2a_2c_2 &= |\mathbf{a}|^2 \\ 2b_1c_2 + 2b_2c_2 &= |\mathbf{b}|^2. \end{aligned} \tag{6.36}$$

The solution to this system of equations leads to the formula (6.33). This finished the proof of Formula 6.8.1.

The next two formulas are useful in blending computations.

6.8.2 Formula. Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be intersecting lines in the plane defined by equations

$$a_1x + b_1y + c_1 = 0 \quad \text{and} \quad a_2x + b_2y + c_2 = 0,$$

respectively. The circles of radius r that are tangent to $\mathbf{L}_1$ and $\mathbf{L}_2$ have centers (d,e) defined by

$$\begin{aligned} d &= \frac{b_1c_2 - c_1b_2 \pm r\left(b_2\sqrt{a_1^2 + b_1^2} - b_1\sqrt{a_2^2 + b_2^2}\right)}{a_1b_2 - a_2b_1}, \\ e &= \frac{a_2c_1 - a_1c_2 \pm r\left(a_1\sqrt{a_2^2 + b_2^2} - a_2\sqrt{a_1^2 + b_1^2}\right)}{a_1b_2 - a_2b_1}. \end{aligned}$$

Proof. From Figure 6.21(a) one can see that there are four solutions in general. Let $\mathbf{L}_1'$ and $\mathbf{L}_2'$ be lines that are parallel to and a distance r from lines $\mathbf{L}_1$ and $\mathbf{L}_2$, respectively. See Figure 6.21(b). There are four such pairs of lines and it is easy to see that the intersection of these lines defines the centers (d,e) of the circles we seek. Now the lines $\mathbf{L}_1'$ and $\mathbf{L}_2'$ are a distance r closer or further to the origin than the lines $\mathbf{L}_1$ and $\mathbf{L}_2$. Therefore, it is easy to see from equation (6.24) that the equations for $\mathbf{L}_1'$ and $\mathbf{L}_2'$ are

$$a_1x + b_1y = -c_1 \pm r\sqrt{a_1^2 + b_1^2} \tag{6.37a}$$

$$a_2x + b_2y = -c_2 \pm r\sqrt{a_2^2 + b_2^2}. \tag{6.37b}$$

Solving equations (6.37) gives our answer.

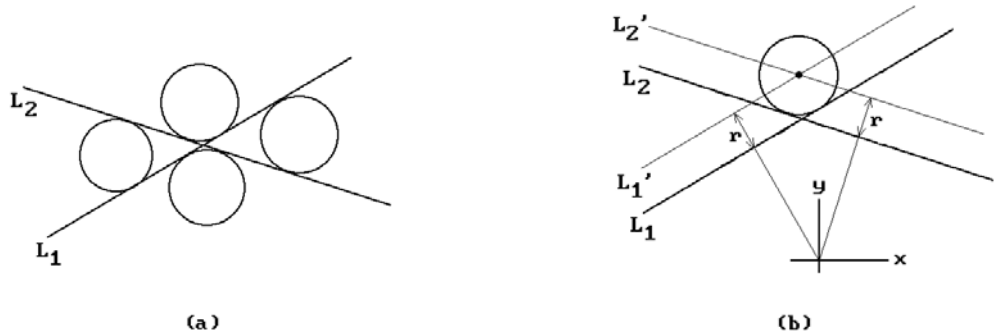


Figure 6.21. Circles of fixed radius tangent to two lines.

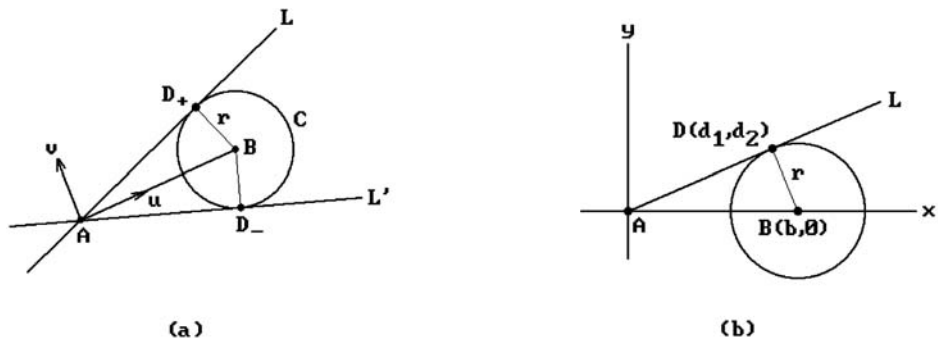


Figure 6.22. Lines through a point tangent to a circle.

6.8.3 Formula. Let **A** be a point and **C** a circle with center **B** and radius r. Assume that $|\mathbf{AB}| > r$. There are two lines through **A** that are tangent to **C** and they intersect **C** in the points $\mathbf{D}_\pm$ defined by

$$\mathbf{D}_\pm = \mathbf{A} + \frac{|\mathbf{AB}|^2 - r^2}{|\mathbf{AB}|}\mathbf{u} \pm \frac{r}{|\mathbf{AB}|}\sqrt{|\mathbf{AB}|^2 - r^2}\,\mathbf{v},$$

where **u** and **v** are the orthonormal vectors

$$\mathbf{u} = \frac{\mathbf{AB}}{|\mathbf{AB}|} = (u_1, u_2) \quad \text{and} \quad \mathbf{v} = (-u_2, u_1).$$

Proof. Figure 6.22(a) shows the two lines **L** and **L′** that pass through **A** and are tangent to **C**. By switching to the coordinate system defined by the frame (**A**,**u**,**v**) we may assume that $\mathbf{A} = (0,0)$ and $\mathbf{B} = (b,0)$. Let $\mathbf{D} = (d_1,d_2)$. See Figure 6.22(b). The following equations are satisfied by **D**:

$$\mathbf{BD} \bullet \mathbf{BD} = r^2$$
$$\mathbf{D} \bullet \mathbf{BD} = 0.$$

In other words,

$$(d_1 - b)^2 + d_2{}^2 = r^2$$
$$d_1(d_1 - b) + d_2{}^2 = 0.$$

If follows that

$$d_1 = \frac{b^2 - r^2}{b} \quad \text{and} \quad d_2 = \pm\frac{r}{b}\sqrt{b^2 - r^2}.$$

Since b corresponds to $|\mathbf{AB}|$ in the original problem, our solution translates into the stated one in world coordinates.

6.8.4 Formula. Consider two circles in the plane centered at points **A** and **B** with radii r_1 and r_2, respectively. Assume that $|\mathbf{AB}| > r_1 + r_2$ and $r_1 > r_2$. Let $\mathbf{D}_{i,\pm}$ and $\mathbf{E}_{i,\pm}$ be the points where the four lines $\mathbf{L}_i$ that are tangent to both of these circles intersect the circles. See Figure 6.23(a). Then

$$\mathbf{D}_{i,\pm} = \mathbf{A}_i + \frac{r_i(r_1 - r_2)}{|\mathbf{AB}|}\mathbf{u} \pm \frac{r_i}{|\mathbf{AB}|}\sqrt{|\mathbf{AB}|^2 - (r_1 - r_2)^2}\,\mathbf{v},$$
$$\mathbf{E}_{i,\pm} = \mathbf{A}_i + \varepsilon_i\frac{r_i(r_1 + r_2)}{|\mathbf{AB}|}\mathbf{u} \pm \frac{r_i}{|\mathbf{AB}|}\sqrt{|\mathbf{AB}|^2 - (r_1 + r_2)^2}\,\mathbf{v},$$

where $\mathbf{A}_1 = \mathbf{A}$, $\mathbf{A}_2 = \mathbf{B}$, $\varepsilon_1 = +1$, $\varepsilon_2 = -1$, and **u** and **v** are the orthonormal vectors

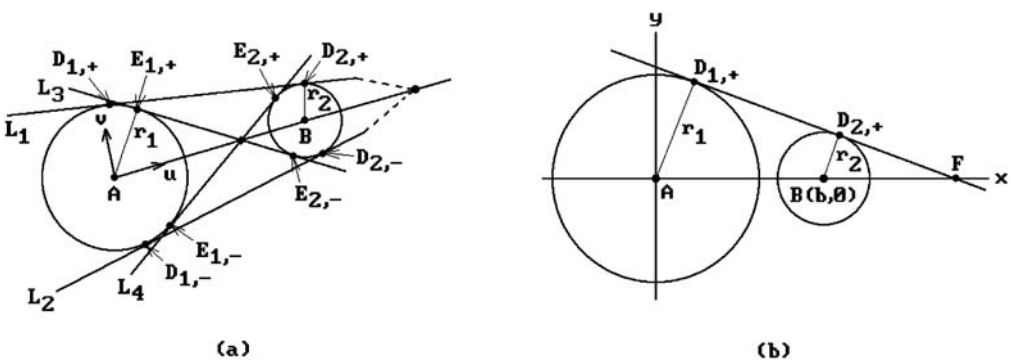


Figure 6.23. Lines tangent to two circles.

$$\mathbf{u} = \frac{\mathbf{AB}}{|\mathbf{AB}|} = (u_1, u_2) \quad \text{and} \quad \mathbf{v} = (-u_2, u_1).$$

Proof. Note that the associated pairs of lines ($\mathbf{L}_1$,$\mathbf{L}_2$) and ($\mathbf{L}_3$,$\mathbf{L}_4$) intersect on the line through the points **A** and **B**. We again switch to the coordinate system defined by the frame (**A**,**u**,**v**) and assume that **A** = (0,0) and **B** = (b,0), b > 0. See Figure 6.23(b). Let **F** = **A** + s**AB** be the point where $\mathbf{L}_1$ intersects the x-axis. Then

$$\begin{aligned} |\mathbf{AF}| &= |s||\mathbf{AB}| \\ |\mathbf{BF}| &= |s-1||\mathbf{AB}|. \end{aligned} \tag{6.38}$$

Because the triangles **ADF** and **BEF** are similar, we get that

$$\frac{r_1}{r_2} = \frac{|\mathbf{AF}|}{|\mathbf{BF}|} = \frac{|s||\mathbf{AB}|}{|s-1||\mathbf{AB}|} = \frac{|s|}{|s-1|}. \tag{6.39}$$

Case 1: s > 1. In this case equation (6.39) implies that

$$s = \frac{r_1}{r_1 - r_2}.$$

Case 2: 0 < s < 1. In this case equation (6.38) implies that

$$s = \frac{r_1}{r_1 + r_2}.$$

In either case, since we now know **F**, we can now use Formula 6.8.3 to compute $\mathbf{D}_{i,\pm}$ and $\mathbf{E}_{i,\pm}$. For example, in Case 1,

$$\mathbf{D}_{2,\pm} = \mathbf{F} - \frac{(s-1)^2|\mathbf{AB}|^2 - r_2^{\,2}}{|s-1||\mathbf{AB}|}\mathbf{u} \pm \frac{r_2}{|s-1||\mathbf{AB}|}\sqrt{(s-1)^2|\mathbf{AB}|^2 - r_2^{\,2}}\,\mathbf{v}.$$

Note that to use Formula 6.8.3 we must use the frame (**F**,–**u**,**v**). It is now a simple matter to rewrite this formula in the form stated earlier.

6.8.5 Formula. Consider two circles in the plane centered at points **A** and **B** with radii r_1 and r_2, respectively. Assume that $r_1 + r_2 < |\mathbf{AB}| < r_1 + r_2 + 2r$. The circles of radius r that are tangent to these circles have center **C** defined by

$$\mathbf{C} = \mathbf{A} + e\mathbf{u} \pm \sqrt{(r_1 + r)^2 - e^2}\,\mathbf{v},$$

where

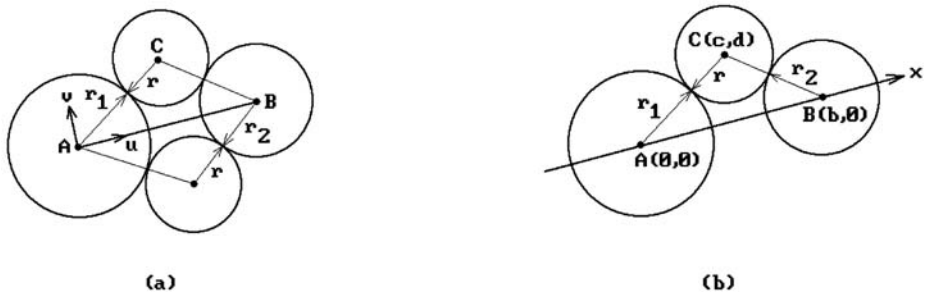


Figure 6.24. Circles of fixed radius tangent to two circles.

$$e = \frac{|\mathbf{AB}|}{2} + \frac{(r_1 + r)^2 - (r_2 + r)^2}{2|\mathbf{AB}|}, \quad \mathbf{u} = \frac{\mathbf{AB}}{|\mathbf{AB}|} = (u_1, u_2), \quad \text{and} \quad \mathbf{v} = (-u_2, u_1).$$

Proof. We can see from Figure 6.24(a) that there are precisely two circles of radius r which are tangent to both circles. What we have to do is find the intersection of two circles: one has center **A** and radius $r_1 + r$ and the other has center **B** and radius $r_2 + r$.

We switch to the coordinate system defined by the frame (**A**,**u**,**v**). Let **A** = (0,0) and **B** = (b,0). See Figure 6.24(b). We must solve the equations

$$x^2 + y^2 = (r_1 + r)^2$$
$$(x - b)^2 + y^2 = (r_2 + r)^2.$$

We get that

$$x = \frac{b}{2} + \frac{(r_1 + r)^2 - (r_2 + r)^2}{2b}$$
$$y = \pm\sqrt{(r_1 + r)^2 - x^2}.$$

This translates to the desired formula in world coordinates since b corresponds to |**AB**|.

This concludes our list of formulas involving circles. Other formulas can be found in [Chas78] and [BowW83].

6.9 Parametric or Implicit: Which Is Better?

Two of the most common ways to present a geometric object **X** is via a parameterization or implicitly as the zeros of an equation. It is natural to ask which is better. The advantages and disadvantages of these two representations are seen best in the context of the following two tasks:

Task 1: Generate some points that belong to $\mathbf{X}$.
Task 2: Determine if a point $\mathbf{q}$ belongs to $\mathbf{X}$.

Assume that $\mathbf{X} \subseteq \mathbf{R}^n$. Suppose that

$$p : \mathbf{A} \to \mathbf{X} \tag{6.40}$$

is a parameterization of $\mathbf{X}$, where $\mathbf{A} \subseteq \mathbf{R}^k$. Suppose also that

$$\mathbf{X} = \{\mathbf{q} | f(\mathbf{q}) = 0\}, \tag{6.41}$$

where f: $\mathbf{R}^n \to \mathbf{R}$.

Advantage of a parameterization:	Task 1 is easy because all one has to do is to evaluate the function p in (6.40) at different values.
Disadvantage of a parameterization:	Task 2 is hard because one has to find a value t that satisfies the equation p (t) = $\mathbf{q}$.
Advantage of an implicit definition:	Task 2 is easy because all one has to do is check if f($\mathbf{q}$) = 0 for the f in equation (6.41).
Disadvantage of an implicit definition:	Task 1 is hard because one has to find values $\mathbf{q}$ for which f($\mathbf{q}$) = 0. Solving equations is usually not easy.

Since both Tasks 1 and 2 are usually handy to be able to carry out in a modeler, it would be nice if one could maintain both a parametric and an implicit representation for an object. The implicit function theorem implies that locally a smooth manifold in $\mathbf{R}^n$ has both a parametric and implicit representation, at least in terms of C^∞ functions. On the other hand, in computer graphics, for computability reasons, one prefers polynomial functions (sometimes rational functions are acceptable) and then the question becomes harder. With the exception of a few well-known spaces, such as conics, finding such representations is difficult in general and falls into the domain of algebraic geometry. See Chapter 10 in [AgoM05] for some answers to the question of how one can convert from parameterizations to implicit representations and vice versa.

6.10 Transforming Entities

This section makes two simple but useful observations about transformations. The first has to do with how vectors transform.

Let M: $\mathbf{R}^n \to \mathbf{R}^n$ be an affine map. The map M can be written uniquely in the form $M = TM_0$, where T is a translation and M_0 is linear transformation with $M_0(\mathbf{0}) = \mathbf{0}$. Let $\mathbf{v} \in \mathbf{R}^n$, where we think of $\mathbf{v}$ as a **vector**. Let $\mathbf{v}'$ be the **vector** to which $\mathbf{v}$ is transformed by the map M. Then

$$\mathbf{v}' = M_0(\mathbf{v}) \quad (\textbf{not } M(\mathbf{v})). \tag{6.42}$$

Mapping a vector is different from mapping a point. The justification of equation (6.42) is that to map a vector one really should map its two "endpoints." To put it another way, if **p** and **q** are two points in $\mathbf{R}^n$, then the direction vector **pq** transforms to the direction vector

$$M(\mathbf{p})M(\mathbf{q}) = M(\mathbf{q}) - M(\mathbf{p}) = T(M_0(\mathbf{q})) - T(M_0(\mathbf{p})) = M_0(\mathbf{q}) - M_0(\mathbf{p}) = M_0(\mathbf{pq}).$$

It follows that translations leave vectors unchanged. For example, if M is defined by equations

$$\begin{aligned} x' &= ax + by + m \\ y' &= cx + dy + n, \end{aligned}$$

then to transform a vector we can drop the translational part and it will transform using the equations

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy. \end{aligned}$$

Next, we look at how parameterizations and implicit representations of a space **X** in $\mathbf{R}^n$ change under a transformation

$$T : \mathbf{R}^n \to \mathbf{R}^n.$$

Let **Y** = T(**X**). Assume that **X** is defined by a parameterization

$$p : \mathbf{A} \to \mathbf{R}^n$$

and is the set of zeros of

$$f : \mathbf{R}^n \to \mathbf{R}.$$

Clearly, **Y** is parameterized by the composition $T \circ p$. If T has an inverse, then **Y** can also be defined implicitly. In fact,

$$\mathbf{Y} = \{\mathbf{q}' \mid f(T^{-1}(\mathbf{q}')) = 0\}. \tag{6.43}$$

Figure 6.25 explains the validity of equation (6.43). The point $\mathbf{q}'$ belongs to **Y** if and only if the point $\mathbf{q} = T^{-1}(\mathbf{q}')$ lies in **X** and satisfies the equation $f(\mathbf{q}) = 0$.

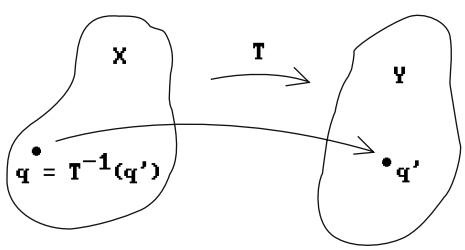
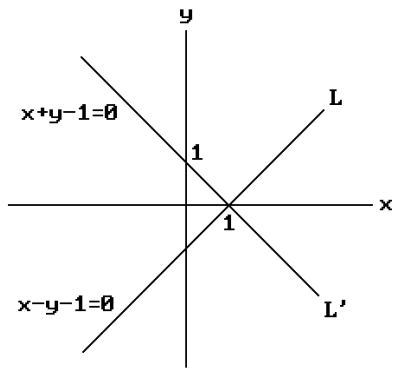


Figure 6.25. Transforming an implicit representation.

Figure 6.26. Transforming a line.



6.10.1 Example. Let **L** be the line in the plane defined by

$$x - y - 1 = 0 \tag{6.44}$$

and let T be the rotation about the origin through the angle of ninety degrees. The equations for T and T^{-1} are

$$\begin{aligned} T : x' &= -y \qquad & T^{-1} : x' &= y \\ y' &= x & y' &= -x. \end{aligned}$$

If **L′** = T (**L**), then from what we just said above, the equation for **L′** is gotten from equation (6.44) by substituting y and –x for x and y, respectively. This gives

$$y - (-x) - 1 = x + y - 1 = 0 \tag{6.45}$$

That this is the correct answer can easily be checked. See Figure 6.26. Simply take two points on **L** and find the equation of the line through the image of these two points under T. For example, the points (0,–1) and (1,0) map to (1,0) and (0,1), respectively, and equation (6.45) contains these two points.

6.11 Exercises

Section 6.5

6.5.1 Find the intersection of the segments [(2,1),(6,–2)] and [(–1,–3),(7,1)].

6.5.2 Find a formula for the intersection of a ray with a segment in the plane.

6.5.3 Find the intersection of the line **L** defined by

$$\begin{aligned} x &= 2 + t \\ y &= 3t \\ z &= -5 + 2t \end{aligned}$$

and the plane **X** with equation $x + y - z = 2$.

6.5.4 Find all intersections of the ray starting at the point $\mathbf{p} = (2,1)$ and direction vector $\mathbf{v} = (1,-3)$ with the circle defined by $x^2 - 2x + y^2 = 0$.

Section 6.6

6.6.1 Prove equation (6.19).

6.6.2 Prove Formula 6.6.2.

Section 6.7

6.7.1 Show that Formula 6.7.4 works for nonconvex polygons.

Section 6.8

6.8.1 Find the equation of the circle through the points $\mathbf{p}_1 = (2,1)$, $\mathbf{p}_2 = (3,3)$, and $\mathbf{p}_3 = (7,1)$.

Section 6.10

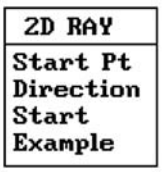
6.10.1 Find the equation of the parabola $y = x^2$ after it is rotated about the origin through an angle of $\pi/3$.

6.12 Programming Projects

Section 6.5

6.5.1 Two-dimensional ray tracing

This project involves implementing a simple 2d ray-tracing program. Create the following submenu for your main program



When the user enters this menu, he/she should be presented with a view looking orthogonally down on the plane. Then the menu items should cause the following actions to be taken when activated:

Start Pt: This should let the user to graphically pick a point **p** on the screen.
Direction: This should ask the user for an angle θ.
Start: This should start drawing the ray from **p** in the direction defined by θ. The ray should continue, reflecting off of any polygonal curves and circles in the world, until the user presses a key. If a ray does not meet any object, that is, it is about to escape to "infinity," draw a short segment in that direction and then stop after telling the user what has happened. Alternatively, rather than waiting for a key to be pressed ask for a number k and quit drawing the ray after k reflections.

Be sure to include a delay each time the ray hits an object so that one can watch the ray trace out its path in the world. Without this, the whole process will happen so fast that one would only see a "final" picture.

Example: This would display a sample world of objects for the user to use in this ray tracing exercise.

An additional nice option in your program would be to allow the user to create his/her own world of circles and closed or open polygons in the plane.

CHAPTER 7

Visible Surface Algorithms

Prerequisites: Parts of Chapter 4 in [AgoM05], mainly Sections 4.5 and 4.7, for Section 7.10

7.1 Introduction

After modeling the geometry, the next basic step in a realistic rendering of a scene from a given viewpoint involves determining those surface patches that will be visible from this viewpoint. This is the *visible surface determination* problem. In the past this was called the *hidden surface removal* problem, but the terminology is beginning to change in order to emphasize the positive rather than the negative aspect of the problem. Many visible surface determination algorithms have been developed over the years. This chapter describes several well-known such algorithms. Although the emphasis is on algorithms for linear polyhedra, we also discuss the additional complexities encountered in curved object cases. The ray tracing approach to visible surface determination, where one simply sends out rays from the camera (or eye) and checks which surface is hit first, is treated in detail in Chapter 10.

An excellent overview of some of the very early algorithms can be found in [SuSS74]. In general, visible surface determination algorithms can be classified as belonging to one of three types: *object precision*, *image precision*, or *list priority*. (We shall follow [FVFH90] and use the more descriptive term "**precision**" rather than the older term "space," as in "object precision algorithm" rather than "object space algorithm".)

Object precision algorithms are of the form

```
for each  object O in the world  do
    begin
        Find the part A of O that is visible;
        Display A appropriately;
    end;
```

whereas image precision algorithms are of the form

```
for each  pixel on the screen  do
    begin
        Determine the visible object O that is pierced
           by the ray from the viewer determined by the pixel;
        if  there is such an O
           then  display the pixel appropriately
           else  display the pixel in the background color;
    end;
```

Both types of algorithms do all their computations to the same precision in which the objects were defined. The main difference between them is that the former accurately compute **all** the visible parts of the world whereas the latter only determine visibility in a **sampled** number of directions. The complexity of image precision algorithms depends on the resolution of the screen, whereas the complexity of object precision algorithms does not (except at the last stage when regions have to be scan converted). In that sense, with the former one has to worry about aliasing. Pure object precision algorithms were used only in the early days of graphics, mainly on vector graphics devices. Given a raster graphics device it is only natural to use the discreteness of the pixels to help make the algorithm efficient.

Ray tracing captures the essence of image precision algorithms, which can be further subdivided into whether they deal with areas or points (*area* versus *point sampling*). The Warnock algorithm is an example of the former. The Z-buffer algorithm, the Watkins algorithm, and ray tracing are examples of the latter.

List priority algorithms fall somewhere in between object and image precision algorithms. They differ from pure image precision algorithms in that they precompute, in object space, a visibility ordering **before** scan converting objects to image space in a simple back-to-front order. Obtaining such an ordering may involve splitting objects. The Schumacker, Newell-Newell-Sancha, and BSP tree algorithm are examples of this type of algorithm.

Like in Chapter 3, each algorithm described in this chapter was selected with certain considerations in mind, which were:

(1) It is currently an acceptable approach in its intended application.
(2) The algorithm was interesting for historical reasons and easy to describe.
(3) It involved the use of some interesting techniques, even though it itself is no longer a recommended method.

This led to the following list categorized by (1)–(3) above:

Algorithm	Category	Comments
Schumacker-Brand-Gilliland-Sharp	(2)	The first list priority algorithm.
Newell-Newell-Sancha	(2), (3)	A depth sort list priority algorithm.
The BSP algorithm	(1), (2)	A list priority algorithm.
Warnock	(2), (3)	An area subdivision image precision algorithm.
Weiler-Atherton	(2), (3)	A more sophisticated area subdivision algorithm.
The Z-buffer algorithm	(1)	**The** algorithm used originally on high-end graphics work stations but now in most graphics systems.
Watkins	(1), (2)	A scan line algorithm that is still worthy of consideration in certain applications because it is **much** faster than a ray-tracing–type algorithm.
Ray tracing	(1)	**The** approach used in conjunction with radiosity methods if one wants the highest quality realistic images. It is discussed at length in Chapter 10.
Octree	(1)	A list priority algorithm useful for volume rendering
The Blinn curved surface algorithm	(2), (3)	

The algorithms above, except for the ray-tracing algorithm, will be discussed in Sections 7.3–7.10. Section 7.2 will describe a preprocessing step called back face removal, which is easy to carry out and saves the algorithms a lot of needless work, but can also be used all by itself to provide some rudimentary realism to scenes. When we discuss the visible surface determination algorithms, the reader needs to be aware about one assumption that is made because it is convenient. Unless explicitly stated otherwise, we shall assume the following:

> **The orthogonal projection assumption:** We assume that the world has been transformed into a coordinate system where the eye is at infinity, so that the projection to the view plane is an orthogonal projection (rather than a central projection) and corresponds to simply dropping the z-coordinate.

Finally, it should be mentioned that there are also *visible line determination* or *hidden line removal* algorithms. These algorithms are used mainly in the context of wireframe displays. Every visible surface determination algorithm can be modified into a visible line determination algorithm, but not vice versa. For an example of this see [PokG89]. The earliest visible line determination algorithm was developed by Roberts in [Robe63]. It assumed all lines came from edges of convex polyhedra. First, back edges were removed. (A *back edge* is a common edge of two back faces.) Each remaining edge was then checked against all the polyhedra that might obscure it. [Roge98] describes this algorithm in great detail. A more general purpose visible line determination algorithm was described by Appel in [Appe67]. Appel introduced a notion of the *quantitative invisibility of a point*, which was the number of front-facing polygons that obscured the point. A point was visible if and only if its quantitative invisibility was zero. Visible line determination lost its importance as computers became more powerful.

7.2 Back Face Elimination

The simplest way to render in a geometric modeling program is to display the world in wireframe mode. Such displays might be quite adequate for the task at hand and they have the added advantage that they are fast. An extremely easy way to make them look more realistic is to eliminate edges common to "back faces."

A back face of a solid object is a face that is facing "away from" the camera. To explain what this means and derive a test for it we need to look at normal vectors. Recall the discussion in Section 6.4. The faces of a solid have a natural outward-pointing normal vector associated to them. On the other hand, a choice of a normal vector for a face is equivalent to having an orientation for the face. Therefore, a general definition for a back face is the following (see Figure 7.1):

Definition. An oriented face is called a *back face* with respect to a vector $\mathbf{v}$ (typically the view direction of a camera) if the angle between its normal vector $\mathbf{n}$ and $\mathbf{v}$ is between 0 and 90 degrees. Mathematically, this simply means that $\mathbf{n} \bullet \mathbf{v} \geq 0$. If $\mathbf{n} \bullet \mathbf{v} \leq 0$, then it is called a *front face* for $\mathbf{v}$.

This definition of back face also handles the case of faces in an arbitrary oriented surface patch that may not be the boundary of a solid, such as the upper hemisphere of the unit sphere or a bicubic patch.

Removing edges on the back faces in wireframe mode works pretty well on boundaries of solids, such as a sphere. On the other hand, some viewers may not be happy with the result in other cases. For example, if one looks diagonally down on a cylindrical surface, back face removal would only show half of it even though the back part would actually be "visible" from the viewpoint. For that reason, a modeling program should have the ability to flag a face as "two-sided," meaning that it should be treated as a "front" face no matter where the viewpoint is.

Finally, it is important to realize that back face removal is based on a local decision. It does not guarantee that the faces that are left are in fact visible. They might be obscured by some other object or even another part of the same object if it is not convex. Therefore, back face removal in no way saves us from a subsequent visible surface determination algorithm. If one wants to display only visible surfaces, one will still basically have to check each face against every other face to see if one obscures the other. What it does do however is cut down the number of faces that have to be looked at by a factor of two on the average, which is a substantial savings, and so visible surface determination algorithms usually have this built in as a preprocessing step because it is so easy to do.

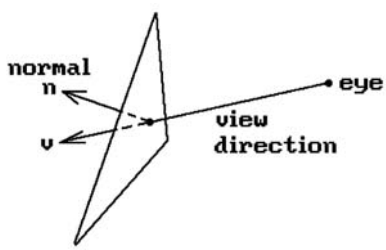


Figure 7.1. Defining a back face.

7.3 The Schumacker List Priority Algorithm

The original list priority algorithm is due to [SBGS69]. Its major contribution was the idea that faces of objects can sometimes be given a priory ordering from which their visibility can be computed independently of the viewpoint. Figure 7.2 shows an example of this. Figure 7.2(a) shows possible priority numbers for the faces. Figure 7.2(b) shows how one uses these numbers. Once an **actual** viewpoint is specified, one eliminates back faces (shown with dotted lines) and then uses the priority numbers on the remaining faces to tell which face is in "front" of another one. More accurately, if one face has a lower priority number than another one, then this means that the second will never obscure the first. In the figure, we would conclude that the face with priority number 2 does not obscure the face with priority number 1. Given the priority numbers we can then use the so-called *painter's algorithm*, Algorithm 7.3.1, to draw the scene. The painter's algorithm gets its name from the way a painter paints an oil painting. As the brush draws over other paint it covers it up.

Of course, the priority numbering that Schumacker was looking for does not always exist for the whole world. One can, however, divide the world into prioritizable *clusters*, that is, collections of faces within which one **can** assign priority numbers to each face with the meaning above. These clusters will then themselves have to be prioritized. One big problem with Schumacker's approach was that too

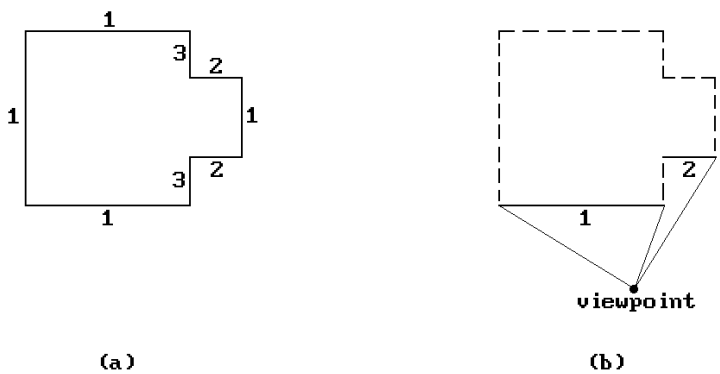


Figure 7.2. A priority ordering for faces.

Given: The faces of a scene listed in 'back" to "front" order, meaning that if face **A** is in back of face **B**, or equivalently, face **B** is in front of face **A**, then **A** will not obscure **B** in any way.

Draw the scene by writing the faces to the frame buffer in back to front order.

Algorithm 7.3.1. The painter's algorithm.

much had to be done by hand, but it was useful in situations where the objects in the scene did not change much and only the viewpoint changed, as, for example, in Schumacker's own application to a flight simulation program.

7.4 Newell-Newell-Sancha Depth Sorting

The Newell-Newell-Sancha visible surface algorithm ([NeNS72]) is an early list priority algorithm that sorts objects by depth and then uses a painter's algorithm to display them. One important difference between it and the Schumacker algorithm is that it computes the depth ordering on the fly and does not rely on an a priori ordering like the Schumacker algorithm. For that reason it is a more versatile algorithm. It introduced some interesting ideas on how to obtain a depth ordering.

The Newell algorithm does an initial crude ordering of the polygon faces of objects based on the z-value of that vertex of a face that is furthest from the viewer. Next, starting with the last polygon **P** in the list (the one furthest from the viewer) and the next to the last polygon **Q**, one checks if **P** can be safely written to the frame buffer by asking whether **P** and **Q** separated by depth, that is, whether

$$(\text{minimum z-value of a vertex of } \mathbf{P}) > (\text{maximum z-value of a vertex of } \mathbf{Q})?$$

If yes, then **P** can never obscure any part of **Q** and we can write **P** to the frame buffer. See Figure 7.3(a). If no, then one considers the set {**Q**} of polygons that overlap **P** in

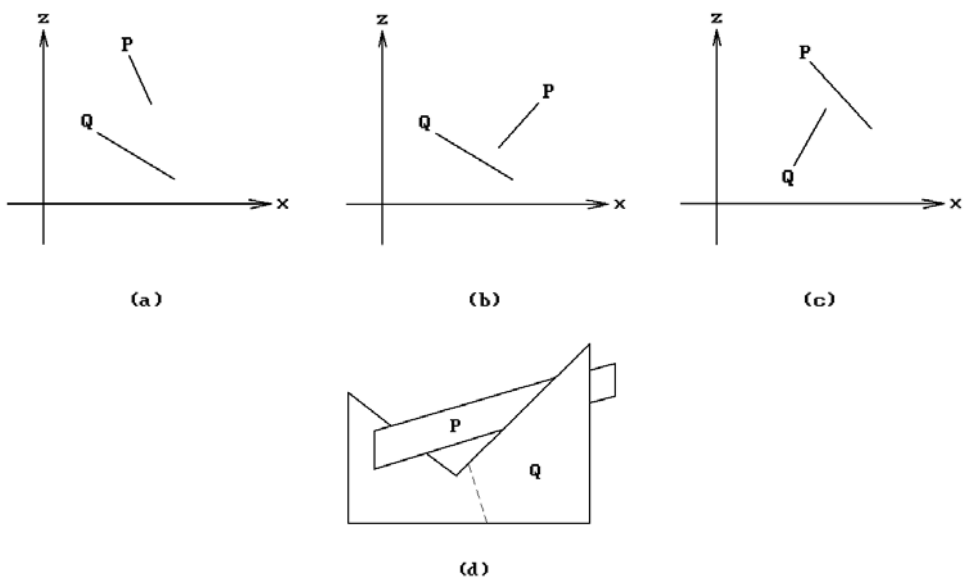


Figure 7.3. Some relative positions of faces.

depth. Although there is this overlap in z, **P** may in fact not obscure any part of any of the **Q**s, and so the algorithm performs a series of tests, ordered by complexity. These tests involve answering the following questions:

(1) Can one separate **P** and the **Q**s in x?
(2) Can one separate **P** and the **Q**s in y?
(3) Is **P** on the farther side of the **Q**s? See Figure 7.3(b).
(4) Are the **Q**s on the near side of **P**? See Figure 7.3(c).
(5) Do **P** and the **Q**s project to disjoint sets?

Test (5) is clearly the most complicated and the hope is that one does not have to perform it very often. If the answer to all these tests is "no," then one swaps **P** with a **Q** and repeats the tests. One has to mark the **Q** to prevent getting into an infinite loop. An attempt to swap an element already swapped at a previous stage implies that one has encountered a "cyclical overlap." An example of this is shown in Figure 7.3(d). In that case one would cut **Q** at the dotted line and replace the old **Q** with the two parts. Eventually one would be able to write the polygons to the frame buffer.

The Newell algorithm handled transparency effects by overwriting the frame buffer only partially. However, aside from the historical interest, the interesting aspect of the algorithm comes from tests (1)–(4), which can be useful in other situations.

7.5 The BSP Algorithm

The *Binary Space Partitioning* (BSP) algorithm ([FuKN80] and [FuAG83]) is a visible surface algorithm that improved on Schumacker's work by automating the division into clusters. The basic BSP algorithm consists of two steps:

(1) A one-time preprocessing step that converts an input polygon list into a binary tree structure called a *BSP tree*
(2) A traversal algorithm that traverses this tree and outputs the polygons to the frame buffer in a back-to-front order

A key idea here (like in Schumacker's work) is that of a *separating plane*. The main condition that this plane has to satisfy is that *no* polygon on the viewpoint side of the plane can be obstructed by a polygon on the other side. To construct the BSP tree, one proceeds as follows:

(1) Select any polygon **P** from the current list of polygons and place it at root of tree.
(2) Each remaining polygon in the list is then tested to see on which side of **P** it lies. If it lies on the same side as the viewpoint one puts it in the left (or "front") subtree list, otherwise one puts it in the right (or "back") subtree list. If a polygon straddles **P**s plane, divide it along that plane and put each of the pieces in the appropriate subtree list.
(3) Repeat this procedure recursively on the two subtree lists.

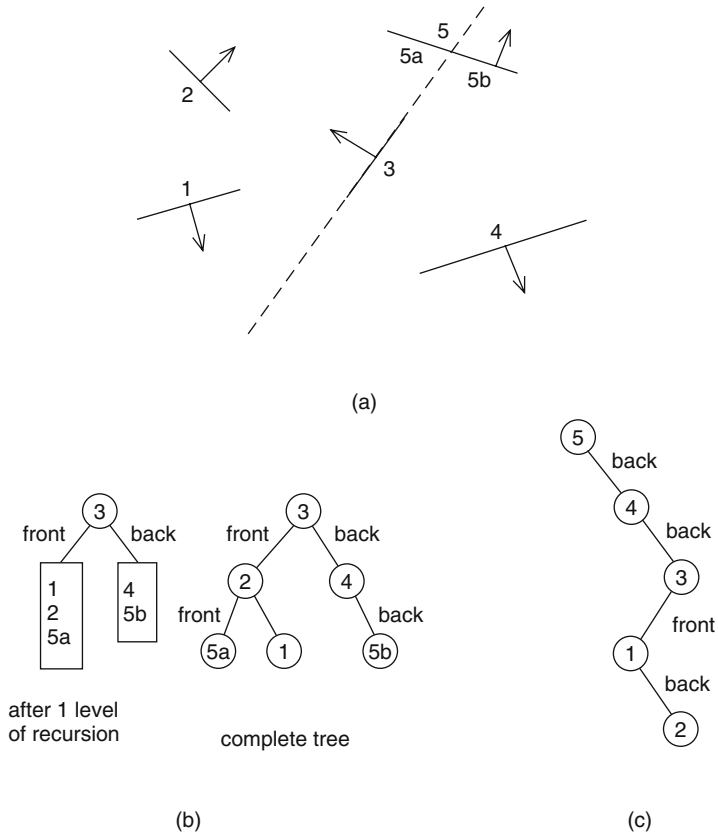


Figure 7.4. A BSP example.

Consider the example shown in Figure 7.4. There are five polygons numbered from 1 to 5 in Figure 7.4(a). The arrows are supposed to indicate the side containing the viewpoint. (In this example, the chosen directions do not correspond to a possible situation.) We assume that polygon 3 is the one chosen first, then 2 and finally 4. The stages of the BSP tree are shown in Figure 7.4(b). Note that choice of polygons can greatly influence the outcome. Figure 7.4(c) shows the tree that we would get if the polygons were chosen in the following order 5, 4, 3, 1, and 2.

Once the BSP tree is generated, it is easy to generate the view by traversing the tree in in-order fashion. At each node of the tree determine whether the eye is in front or in back of the current root polygon. Traverse the opposite side of the tree, output the root polygon to the frame buffer, and then traverse the remaining side.

Algorithm 7.5.1 is a more precise description of how the BSP tree is built and traversed. It was originally feared that the BSP tree would be significantly larger than the input polygon list, but this did not turn out to be the case in practice. To cut down on the number of polygons that are split, the following heuristic was used: Select that polygon to be the root of the next subtree that cuts the fewest other polygons. It was discovered that it was not necessary to look over **all** of the remaining polygons. Rather,

```
BSPtree function  BuildBSPTree  (polygon list plist)
begin
    polygon list  frontList, backList;
    polygon       root, poly, frontPart, backPart;

    if  IsEmpty (plist)  then  return (Empty BSP tree);

    frontList := nil;    backList := nil;
    root := SelectAndRemoveOne (plist);
    for  poly in plist  do
        if  InFrontOf (poly,root)
            then  Insert (poly,frontList)
            else  if  InBackOf (poly,root)
                then  Insert (poly,backList)
                else
                    begin
                        SplitPolygon (poly,root,frontPart,backPart);
                        Insert (frontPart,frontList);
                        Insert (backPart,backList);
                    end;

    return  (CreateBSPTree (BuildBSPTree (frontList),root,BuildBSPTree (backList)));
end;

Procedure TraverseBSPTree (BSPTree T)
{ The procedure Display is assumed to do all the necessary transformations,
  clipping, shading, etc. }
begin
    if  IsEmpty (T)  then  return;
    if  InFrontOf (eye,rootPolygon (T))
        then
            begin
                TraverseBSPTree (backSubtree (T));
                Display (rootPolygon (T));
                TraverseBSPTree (frontSubtree (T));
            end
        else
            begin
                TraverseBSPTree (frontSubtree (T));

                { If back faces are not to be displayed, remove the next statement }
                Display (rootPolygon (T));

                TraverseBSPTree (backSubtree (T));
            end
end;
```

Algorithm 7.5.1. The BSP tree algorithm.

it was sufficient to choose the best from a few chosen **at random** (five seemed to work well).

We have just described the original BSP algorithms. Variants have been developed since then. For example, Kaplan ([Kapl85]) chose his separating planes to be parallel to the coordinate planes of the reference coordinate system. Such an algorithm is sometimes called an *orthogonal BSP algorithm*. This property of separating planes can simplify computations, especially when objects have bounding boxes. We shall mention other BSP algorithms later on when discussing ray tracing in Section 10.2.

To summarize, the BSP algorithm is a good algorithm where the world model does not change much and only the viewpoint changes. Examples of such situations are flight simulators, architects walking through a house, and biochemists viewing complicated molecules. Chin ([Chin95]) describes an efficient way to implement and use BSP trees.

7.6 Warnock and Weiler-Atherton Area Subdivision

The Warnock visible surface determination algorithm [Warn69] is an image space algorithm that attempts to find rectangular regions (here called *windows*) of the same intensity on the screen (*area coherence*). Algorithm 7.6.1 is an outline of the algorithm. The polygons referred to in the algorithm are the **projected** polygons. See Figure 7.5.

Essential to this algorithm is the ability to perform the following tests on any polygon **P**:

Initialize a list L of windows to consist of a single window that is the entire screen;

For each window W in the current list L of windows look for one of the following "trivial" cases:

(1) All polygons are disjoint from W. In this case one draws W in the background color.
(2) Only one polygon **P** intersects W. In this case draw the intersection of **P** and W in the color of **P** and the rest of W in the background color. In practice, this case is divided into three subcases: **P** is contained in W, **P** surrounds W, **P** and W have a nontrivial intersection.
(3) At least one surrounding polygon was found and it is in front of all other polygons that intersect the window. In that case draw the window in the color of that polygon.

Otherwise, divide the window W into four equal smaller windows, add them to the list L of windows, and repeat the process until one gets down to a window the size of a pixel. At that point one checks directly which polygon is in front of all the others at that pixel.

Algorithm 7.6.1. Outline of the Warnock algorithm.

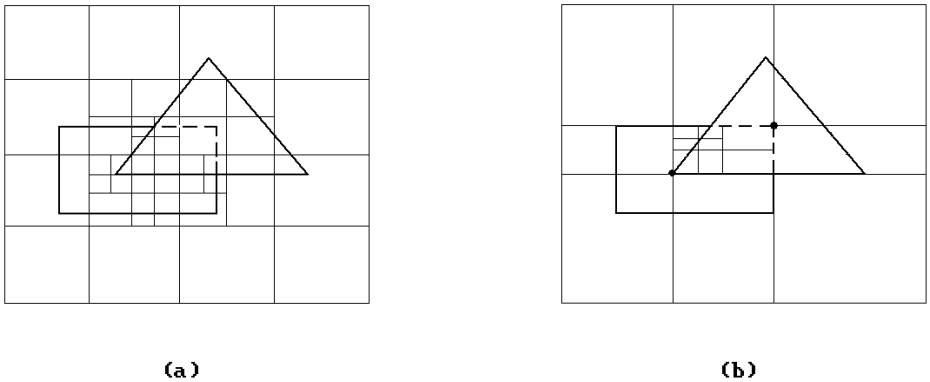


Figure 7.5. Area subdivision examples.

Test 1: Is **P** disjoint from a window?
Test 2: Does **P** surround a window?
Test 3: Does **P** partially meet a window?
Test 4: Does **P** fall inside a window?
Test 5: Is **P** in front of other polygons?

For a quick test for disjointness, one usually uses bounding boxes. One way to test if the window falls inside a polygon is to substitute the vertices of the window into the equations for the edges of the projected polygons. If these tests fail, then one needs to check if the boundary of the polygon intersects the window by checking each edge of the polygon against each edge of the window. If the boundaries are disjoint, then one still has to distinguish between the case where the regions are disjoint or where one contains the other. We discussed some tests for doing this in Section 6.3. One approach is to use a parity test and count the number of times that, say, any horizontal ray starting at a window vertex intersects the polygon. If the number is even, then the two are disjoint, otherwise the polygon surrounds the window. Another approach is to use an angle counting argument.

Tests 1–4 above dealt with view plane issues. Test 5 involves depth calculations. As mentioned earlier, we are assuming an orthogonal projection (with the camera at $-\infty$) and so testing if one point is in front of another amounts to checking that its z-value is less than that of the other point. Assume that two polygons **P** and **Q** meet the window. We give a test for whether **P** is in front of **Q**. In Warnock's algorithm this test is only needed for the case **P** is a surrounding polygon, but the fact that it is surrounding is not important. Here is the test: if the depth of the plane of **P** is less than the depth of the plane of **Q** at the four corners of the window, then **P** is in front of **Q** (Figure 7.6(a)). This is a sufficient but not necessary condition for that to happen (Figures 7.6(b) and (c)). Warnock subdivides the window if the test fails.

There are many variations of Warnock's algorithms. The windows need not be rectangular. The problem with rectangular windows is that, being bad matches to most polygons, the algorithm has to recurse down to the pixel level a lot. The Weiler-Atherton algorithm ([WeiA77]) uses subwindows that match the shape of the polygons. See Figure 7.7, which shows a list of polygons being clipped against a polygon

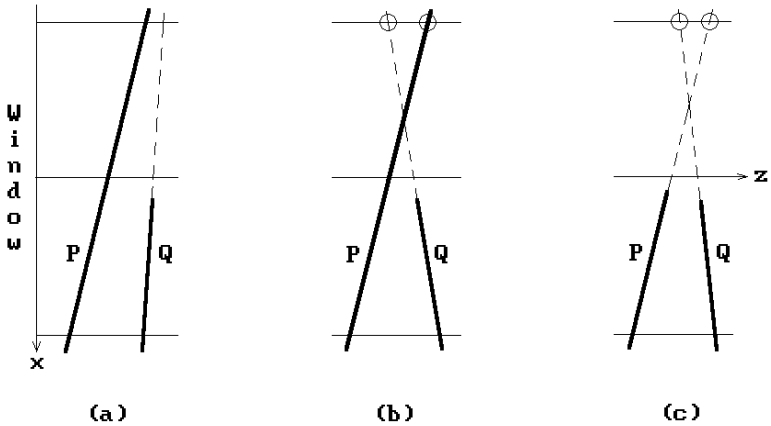


Figure 7.6. Examples of a face in front of another face.

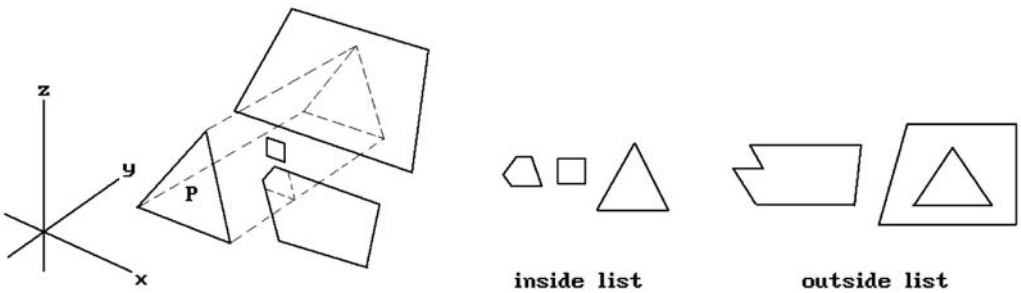


Figure 7.7. Weiler-Atherton type area subdivision.

P. The resulting pieces are separated into two lists, those that lie inside **P** and those that lie outside. This is more complicated to implement and the authors had to develop a new clipping algorithm, which was discussed briefly in Section 3.3.2, but it is more efficient.

Both the Warnock and Weiler-Atherton algorithms are not scan line oriented but jump around as they draw the scene. Their complexity is approximately the complexity of the final display (not of the scene).

7.7 Z-buffer Algorithms

Z-buffer algorithms record "current" depth information for each pixel. A "real" Z-buffer is a two-dimensional array of real numbers of the **same** size as the frame buffer. The real numbers record current depth information. A Z-buffer algorithm then scan converts an entire face at a time into both the frame and the Z-buffer. The high-level outline of such algorithms is extremely simple and is shown in Algorithm 7.7.1. There

```
Let Depth(p) denotes the "depth" of a point p, that is, the z-coordinate of p in the
camera or eye coordinate system. The array DEPTH below holds these z-values of
object points nearest to the eye.

color array FRAMEBUF[XMIN..XMAX,YMIN..YMAX];
real array  DEPTH[XMIN..XMAX,YMIN..YMAX];

begin
    Initialize FRAMEBUF to the background color;
    Initialize DEPTH to ∞;

    for all faces F in the world do
        for each point p of F do
            if p projects to FRAMEBUF[i, j] for some i, j then
                if Depth (p) < DEPTH[i, j] then
                    begin
                        FRAMEBUF [i, j] := color of F at p;
                        DEPTH [i, j]    := Depth (p);
                    end
end;
```

Algorithm 7.7.1. The Z-buffer algorithm.

are variants of the Z-buffer algorithm that incorporate antialiasing. An early scan line approach described in [Catm78] was computationally expensive. A more efficient approach, called the A-buffer algorithm, is described briefly in Section 7.11.

A Z-buffer takes up a lot of memory. By taking a scan line approach, one only needs an array as long as a scan line. This leads to a slightly modified Z-buffer algorithm shown in Algorithm 7.7.2. See [Roge98]. Such Z-buffer algorithms all use the notion of "segment" and "span" in a scan line (where a face intersects the scan line) and ask the question "Which segments are visible?" One uses the x-z plane. The dotted lines in Figure 7.8(a) divide the scan lines into "spans." Within each span one determines the visibility of segments by checking their depths using **plane equations**. Different choices of spans are possible because the only criterion is that one can unambiguously order the segments within it by their depths. Therefore, there is the issue of how to best choose the spans? For example, Figure 7.8(b) shows a better choice of spans than Figure 7.8(a).

The scan line algorithms all do

(1) a Y sort to limit attention of algorithm to edges or faces which intersect scan line
(2) an X sort
(3) Z depth search

With regard to (1), one brings in new edges, etc., as one proceeds to the next scan line and discards those that are no longer needed. One uses coherence and a list of "active"

Assume that the world has been transformed to the coordinate system in which the pixels on the screen correspond to the integer coordinate points of the rectangle [XMIN,XMAX] × [YMIN,YMAX] × 0 and we are doing an orthogonal projection (the camera is at −∞ on the z-axis). The function Depth(**p**) returns the z-coordinate of the point **p**, and ix_p will denote the x-coordinate of **p** rounded to the nearest integer.

```
color array COLOR[XMIN..XMAX];  { holds color of pixels on current scan line }
real array  DEPTH[XMIN..XMAX];  { holds z-value of point nearest to eye }

for iy:=YMIN to YMAX do                { for each scan line of the screen }
    begin
        Initialize COLOR to the background color;
        Initialize DEPTH to ∞;

        for each projected face F do
            if F intersects the plane y = iy then
                begin
                    Let L be the line segment which is this intersection;
                    for each p ∈ L do
                        if Depth (p) < DEPTH [ix_p] then
                            begin
                                COLOR [ix_p] := color of S;
                                DEPTH [ix_p] := Depth (p);
                            end;
                end;

        Display COLOR;
    end;
```

Algorithm 7.7.2. A scan line Z-buffer algorithm.

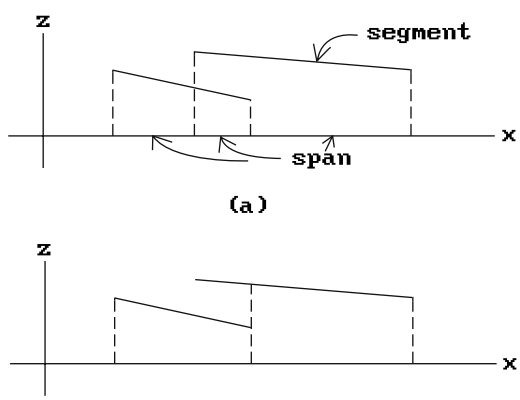


Figure 7.8. Z-buffer spans.

edges. For (2) one divides the scan line into "sample spans" within which the same face is visible (one is using "point-to-point" coherence along the line). Finally, at step (3) each sample span must then be processed. How this is done depends on how they were chosen. If we use the method indicated in Figure 7.8(a), then this is straightforward, although if we allow penetrating faces there are extra modifications needed: At each vertex or span endpoint one must compute the depths of the segments that fall into the span.

7.8 The Watkins Scan Line Algorithm

The Watkins algorithm ([Watk70]) is a scan line image space algorithm that is especially efficient. The idea is to let the right end of the current span "float" and keep the left end fixed. One starts at the extreme right. As new segments are taken from the x-sorted list, the right end moves left until one gets a span that is simple enough to compute which segment is visible.

Algorithm 7.8.1 gives an outline of a Watkins-type algorithm. There are two things to note. One is that for parity to work, one needs to shorten the edges appropriately as described in Section 2.9.1. The other is that we assume that polygons have been clipped.

To help clarify the algorithm we shall work through an example. Consider the scene shown in Figure 7.9(a). Figures 7.9(b) and (c) show its projection onto the x-z and x-y plane, respectively. Suppose that we are at scan line y as shown in Figure 7.9(c). The critical points along the x-axis are 0, a, b, c, and d. Table 7.7.1 shows how the x scan progresses. We show the values of the indicated variables at the locations in the procedure ProcessActiveEdgeList marked "LA" or "LB." Note how the active flag for each polygon toggles back and forth between **true** and **false** based on a parity-type argument.

The objects in Figure 7.9(a) did not intersect. Figure 7.10 shows a case where one polygon penetrates another. In such a case we have to find intersections and keep moving the right end of the span to the left, saving the current values as we go along, until we find an intersection-free segment. In the case shown there is one intersection with x value x_I in the segment [b,c]. We need to check **all** active polygons against **all** other active polygons when testing for intersections. We can tell if two polygons intersect by looking at their z-depth values at the endpoints of the span, call these zl_1,zr_1 for the first polygon and zl_2,zr_2 for the second polygon. The polygons will intersect if

$$((zl_1 < zl_2)\ \textbf{and}\ (zr_1 > zr_2))\ \textbf{or}\ ((zl_1 > zl_2)\ \textbf{and}\ (zr_1 < zr_2)).$$

If we find an intersection, then certain branches in the code of the procedure LastVisiblePolygonColor will now be executed. The new x scan will now pass statements marked "LC" and "LD," as we can see from Table 7.7.2 in our example. First we save c on the stack and deal with the segment [b,x_I]. Then we set spanLeft to x_I, pop c from the stack, and deal with [x_I,c]. The rest proceeds as before.

Notice that we always check if segments intersect in an endpoint of a span. If so, then we use the opposite endpoint's x value to do a depth check to see which polygon is closer.

```
Assume that the viewport is [XMIN,XMAX]×[YMIN,YMAX] .

{ The data record we need for each polygon in the world. }
polydata = record
    integer      y,              { first scan line crossed by poly }
    edgedata list edges;         { each edge of polygon has an associated edgedata record }
    real         a, b, c, d;     { ax + by + cz + d = 0  is plane equation for polygon }
    color        hue;
    boolean      active;         { updated as the algorithm proceeds }
end;

{ horizontal span endpoint record }
edgedata = record
    polydata pointer polyP;      { pointer to data for polygon to which edge belongs }
    real     x,                  { where the edge intersects the current scan line }
             dx;                 { change in x from one scan line to the next }
    integer dy;                  { the number of scan lines left to cross by the edge }
end;

{ Global variables: }
polydata list activePolys;       { the list of currently active polygons }
edgedata list activeEdges;       { the list of currently active edges }
polydata list array buckets[YMIN..YMAX];
    { buckets[y] holds all polygons which "start" at scan line y }
real spanLeft, spanRight;        { used by procedure ProcessActiveEdgeList }

procedure WatkinsAlgorithm ()
begin
    integer y;

    InitializeData ();

    for  y:=YMIN to YMAX  do
        begin
            Add any polygons in buckets[y] to active polygon list activePolys;

            Scan edges of polygons in activePolys and add those that start at y
                to active edge list activeEdges;

            Sort edges of activeEdges by increasing x;
            (Sorting is needed because when the list gets updated below the ordering
              is destroyed if edges cross.)

            ProcessActiveEdgeList ();
            UpdateActiveEdgeList ();
            UpdateActivePolygonList ();
        end
end;
```

Algorithm 7.8.1. A Watkins visible surface algorithm.

```
procedure  InitializeData ()
begin
    integer i;

    activePolys  := nil;
    activeEdges := nil;

    { Initialize all buckets to nil }
    for  i:=YMIN to YMAX  do  buckets[i] := nil;

    for each  polygon P in the world  do
        begin
            Create a new polydata record pData for P (active field is set to false);
            Add pData to buckets[pData.y];
        end
end;

procedure ProcessActiveEdgeList ()
begin
    color        spanColor;
    integer      polyCount;
    edgedata  E;
    polydata   P;

    spanLeft   := XMIN;
    polyCount := 0;

    for  E in activeEdges  do
        begin
            spanRight := E.x;
{LA}            case  polyCount  of
                0    :  spanColor := backgroundColor;
                1    :  spanColor := ColorOf (OnlyMemberOf (activePolys));
                >1  :  spanColor := LastVisiblePolygonColor;
            end;

            P := PolydataOf (E);
            ToggleActive (P);    { if active field true, set to false and vice versa }
            if  IsActive (P)
                then  polyCount := polyCount + 1
                else  polyCount := polyCount – 1;

{LB}        Display ([spanLeft,spanRight],spanColor);
            spanLeft := spanRight;
        end;

    if  spanLeft < XMAX  then  Display ([spanLeft,XMAX],backgroundColor);
end;
```

Algorithm 7.8.1. *Continued*

```
Procedure CheckForSegIntersections (ref boolean intersected;  ref real xint)
if   two segments in the activeEdges list intersect at an x-coordinate xint with
                        spanLeft < xint < spanRight
     then  intersected := true
     else
          begin
               intersected := false;
               if  two segments in activeEdges touch at spanLeft
                    then  xint := spanRight
                    else   xint := spanLeft;
          end;

polydata function MinActiveZvaluePolygon (real x, y);
{ Scan active polygon list and return the data for the one with minimum zvalue at
  (x,y).  The z values are determine by the equation  z = –(a*x + b*y + d)/c   where
   a,b,c, and d are the coefficients of the planes of the polygons. }

color function LastVisiblePolygonColor ()
{ Checking for intersections is where we may have to do a lot of work }
begin
     real stack   spanStack;
     boolean      intersected;
     real         xint;

     Reset stack of spans spanStack to empty;
     repeat forever
          CheckForSegIntersections (intersected,xint);
          if  intersected
               then
                    begin
                         { There was an intersection.  Push current spanRight,
                            divide the span, and process the left half [spanLeft,xint] }
                         Push (spanright,spanStack);
                         spanRight := xint;
{LC}                 end
               else
                    begin
                         segcol := ColorOf (MinActiveZvaluePolygon (xint,y));

                         if  Empty (spanStack)  then  return (segcol);

                         Display ([spanLeft,spanRight],segcol);
{LD}                     spanLeft   := spanRight;
                         spanRight := Pop (spanStack);
                    end
     end
end;  { LastVisiblePolygonColor }
```

Algorithm 7.8.1. *Continued*

```
procedure UpdateActiveEdgeList ()
begin
    edgedata E;

    for  E in activeEdges  do
        if  E.dy = 0
            then  Delete E from activeEdges
            else
                begin
                    E.dy := E.dy – 1;
                    E.x   := E.x + E.dx;
                end;
end;

Procedure UpdateActivePolygonList ()
begin
    polydata P;

    for  P in activePolys  do
        if  P.dy = 0  then  Delete P from activePolys
                      else  P.dy := P.dy – 1;
end;
```

Algorithm 7.8.1. *Continued*

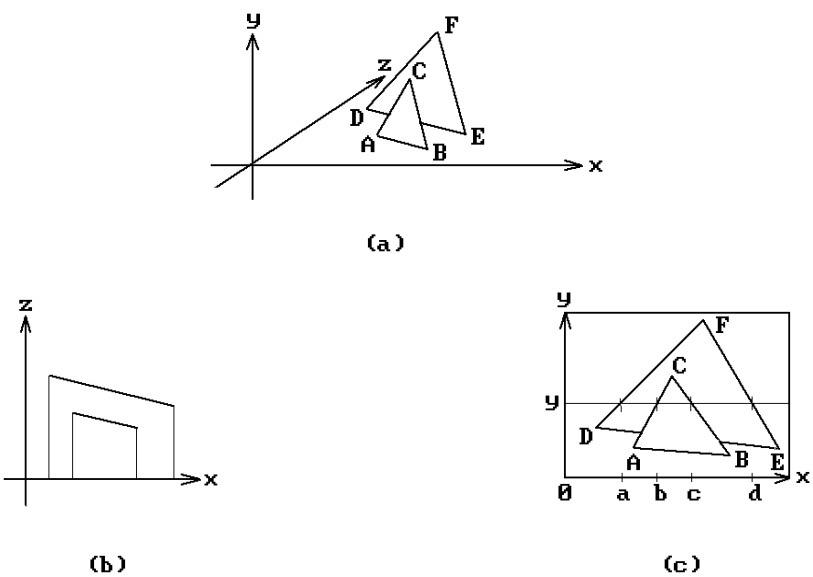


Figure 7.9. A Watkins scan line algorithm example.

Table 7.7.1 The x-scan data at scan line y for Figure 7.9(c).

Label	polyCount	spanLeft	spanRight	ABC.active	DEF.active
LA	0	0	a	**F**	**F**
LB	1	0	a	**F**	**T**
LA	1	a	b	**F**	**T**
LB	2	a	b	**T**	**T**
LA	2	b	c	**T**	**T**
LB	1	b	c	**F**	**T**
LA	1	c	d	**F**	**T**
LB	0	c	d	**F**	**F**

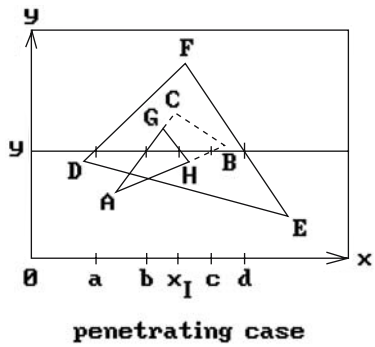


Figure 7.10. The Watkins scan line algorithm with penetrating objects.

Table 7.7.2 The x-scan data at scan line y for Figure 7.10.

Label	polyCount	spanLeft	spanRight	ABC.active	DEF.active	spanStack
LA	0	0	a	**F**	**F**	**nil**
LB	1	0	a	**F**	**T**	**nil**
LA	1	a	b	**F**	**T**	**nil**
LB	2	a	b	**T**	**T**	**nil**
LA	2	b	c	**T**	**T**	**nil**
LC	2	b	x_I	**T**	**T**	c
LD	2	x_I	c	**T**	**T**	**nil**
LB	1	x_I	c	**F**	**T**	**nil**
LA	1	c	d	**F**	**T**	**nil**
LB	0	c	d	**F**	**F**	**nil**

7.9 Octree Algorithms

Because of the regular geometric structure underlying octrees, it is fairly easy to devise list priority visible surface algorithms for objects represented in this way if one is using a parallel projection. One lists the voxels in a back-to-front order. For example,

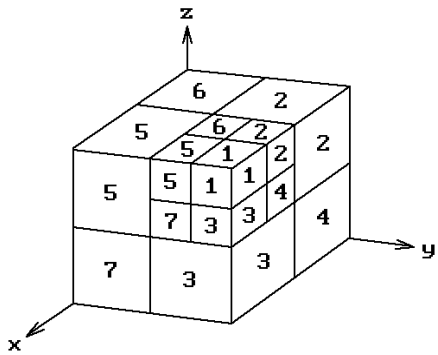


Figure 7.11. Determining a back-to-front order for voxels.

consider Figure 7.11 and assume that the camera is in the first octant looking toward the origin. In that case, the sequence 8, 7, 4, 6, 5, 2, 3, 1 is one order in which to list the voxels so that no voxel in the list will ever be obscured by a voxel appearing earlier in the list. As each voxel is subdivided, we would use the same order for the new subdivided voxels.

Assuming that the voxel faces are parallel to the coordinate planes, it is straightforward to define a back-to-front order for the voxels for an arbitrary orthographic projection based simply on knowing into which of the eight octants the view plane normal is pointing. This can be generalized to an arbitrary parallel projection, not just orthographic projections. Having decided on a back-to-front order of the voxels (there is more than one), one then simply writes them to the frame buffer in that order like in the painter's algorithm. For more details and references to papers see [FVFH90] and [Roge98].

For a visible surface algorithm based on ray tracing in the volume rendering case see Chapter 10. One has to define discrete 3d rays first, which is done in Section 10.4.1.

7.10 Curved Surface Algorithms

One of the earliest curved surface visible surface algorithms is Catmull's z-buffer algorithm described in [Catm75]. It recursively subdivided surface patches until they covered a single pixel and then updated the z-buffer appropriately. In this section we describe a scan line approach that is due to Blinn ([Blin81]). Before we start, however, we need to point out that curved surface algorithms like Blinn's are hardly, if ever, used anymore. Nowadays, one simply generates a sufficiently close polygonal approximation to the surface and displays those polygons. We will have more to say about this in Chapter 14. High-performance graphics systems now have hardware support to make this feasible. Nevertheless, it is worthwhile discussing Blinn's algorithm anyway because it brings out some interesting aspects of curved surfaces.

Let us begin with a review of the polygonal case. The top level scan line algorithm was:

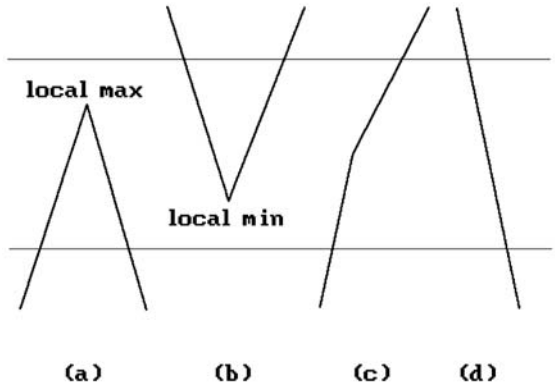


Figure 7.12. Basic cases to watch for in scan line algorithms.

```
for  iy := MINY to MAXY  do
  begin
    intersect plane  y = iy  with objects to get segments;
    for  ix := MINX to MAXX  do
      show point (ix,iy) in segments, if any, which are
        closest to viewer;
  end;
```

Of course one had to make this practical using "coherence." We made incremental computations and had "active edge lists." Figure 7.12 shows the basic cases for which we had to watch out as we moved from scan line to scan line. We did various sorts to speed things up.

As we pass to curved surfaces we need to handle new cases. Assume that the surface is parameterized by a function

$$f(u,v) = (X(u,v), Y(u,v), Z(u,v)).$$

We need to look at level curves. Again assuming an orthogonal projection with the y-axis corresponding to scan lines, these level curves are defined by equations

$$Y(u,v) = c.$$

In the linear case all the geometric information was contained in the edges and these could be represented by their endpoints. Now this is not enough because of "silhouettes." A *silhouette* is formed by points where the normal to the surface is orthogonal to the z-axis (or line of sight for a nonorthogonal view).

Let us keep track of points on level curves that are endpoints on patches or silhouettes. We shall do this by keeping track of their **parameter** values (u,v) and update them as we move down the scan lines. This is an important point, namely, that

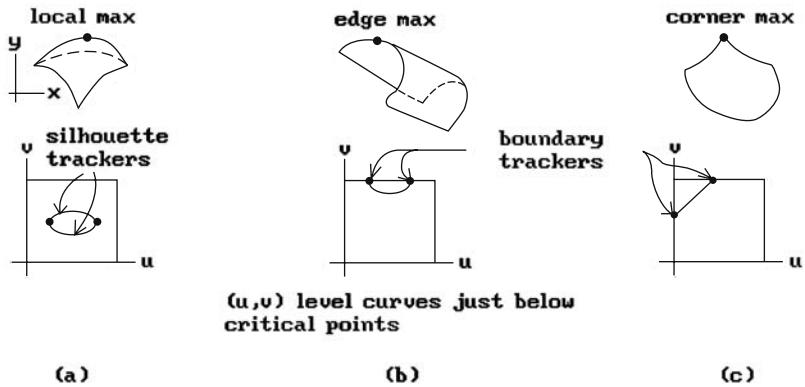


Figure 7.13. Some boundary tracker cases.

one works here in parameter space rather than the corresponding points in 3-space. Blinn refers to this tracking process as maintaining "edge trackers." The questions that need to be asked are: How are "edge trackers" defined? How do they get their edges? How do edges appear/disappear?

To see how edges are created, assume that all the critical points and maxima and minima of f have been found. Figure 7.13 shows some possible cases. The type of edges that are created depends on the type of critical point. In case (a) we have a strict local maximum. From that we get an elliptical intersection and two silhouette trackers. In case (b) we have an edge maximum with a parabolic intersection for two boundary trackers. In case (c) there is a corner maximum that basically gives us a linear intersection for two boundary trackers. In all these cases, initial (u,v) values must be found. These are then updated using a Newton-Raphson method where possible. This works fine in case (c) using the corner as a starting point. In cases (a) and (b) we cannot update with the Newton-Raphson method because the derivative vanishes. Instead, one uses the second derivative of f to locally approximate the level curve of f at the **next** scan line by a parabola or ellipse. This approximation of the level curve is used as an initial guess at (u,v).

Unfortunately, there are all kinds of things that can happen in the curved surface case other than "standard" maxima and minima. Figure 7.14 shows a "folded edge." This is detected by checking the sign of the z- component of the normal, denoted by $\mathbf{n}_z$, of each **boundary** tracker. If it changes from one scan line to the next, then we have this case. What we have to do is to create a new silhouette tracker using the z-coordinate of the boundary edge tracker as an initial guess. Figure 7.15 shows a saddle point. Such points cause the creation of a new silhouette tracker.

When it comes to deleting edge trackers, there are two events for which we have to watch:

(a) when passing a local minimum or
(b) when a silhouette leaves at a boundary.

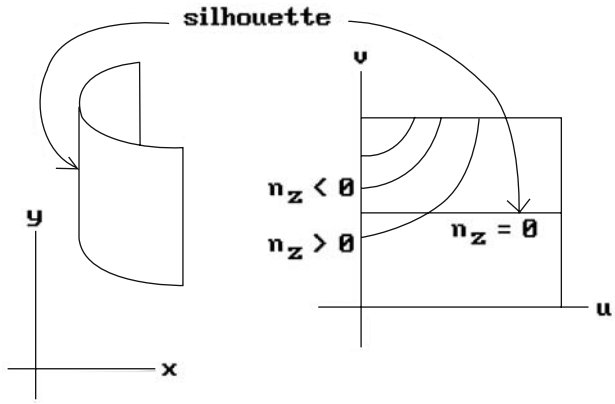


Figure 7.14. Checking for a folded edge.

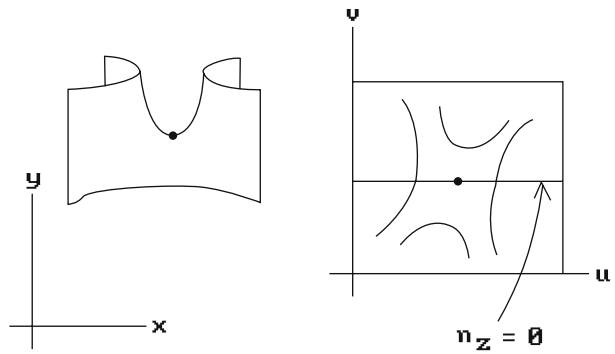


Figure 7.15. Checking for a saddle point.

One deals with (a) by maintaining a list of local minima. Then when one updates the Y-scan, one checks against this list but keep in mind that

(1) The list gives the value in u-v space and we need to find the corresponding point on the edge tracker, that is the scan line.
(2) The numerical operations may fail to converge near such points. Therefore, use a modified Newton-Raphson iteration and discard those points which do not converge.

With regard to (b), we have to keep checking u and v to see if they still lie in [0,1] as we update the silhouette tracker. If either value does not, then delete the tracker.

When moving along a scan line from left to right one needs to maintain a list of active intersection points. These are obtained from the cross-section curves in the x-z plane. The intersections should be kept sorted by their x value. New points enter and leave this list at boundary edges and silhouettes. One updates their value by a two-variable Newton-Raphson method applied to the equations

$$F(u,v) = Y(u,v) - Yscan = 0$$
$$G(u,v) = X(u,v) - Xscan = 0.$$

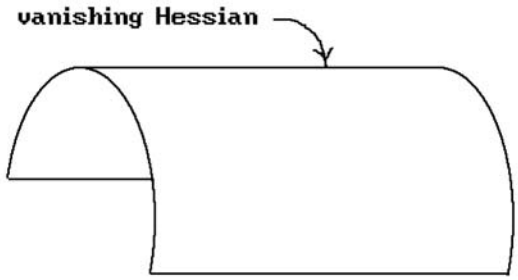


Figure 7.16. A vanishing Hessian case.

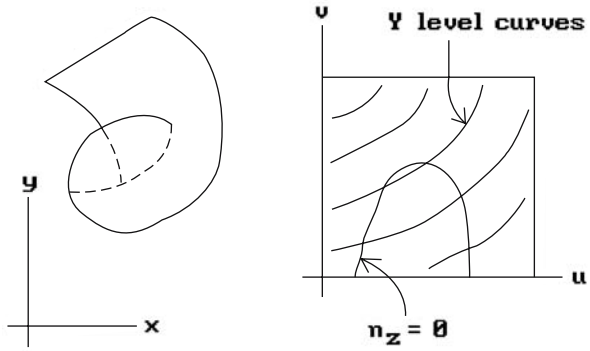


Figure 7.17. A curtain fold example.

We have outlined the main idea behind Blinn's approach, but to implement it one has to overcome a number of special problem cases often caused by denominators vanishing in certain expressions. Some examples of these are:

Vanishing Hessians: See Figure 7.16. Here we need four edge trackers below a maximum, rather than two. The hard part is to decide when such a phenomena occurs.

Curtain folds: See Figure 7.17. Here $\mathbf{n}_z$ becomes tangent to a y-level curve. One would need another pre-scan for such points, but Blinn avoids this via a heuristic.

Singularities intersecting singularities: This is a very general problem. An example of this is where a strict local maximum lies on the boundary of a patch.

With regard to the speed of the algorithm, Blinn found that the average number of iterations needed for the convergence of edge trackers was about 2.5. He was able to improve this substantially by extrapolating in the u-v space: He saved the (u,v) value from the previous scan line and used for an initial guess an extrapolation of the previous two values. This reduced the average number of iterations to less than 1.

To speed the x-scan Blinn found intersections by choosing a sample of points and interpolating. This amounted to approximating level curves via polygonal lines. The sample points were chosen by spacing them equally according to the angle of the normal to the curve. Note that this clusters at places of high curvature. See Figure

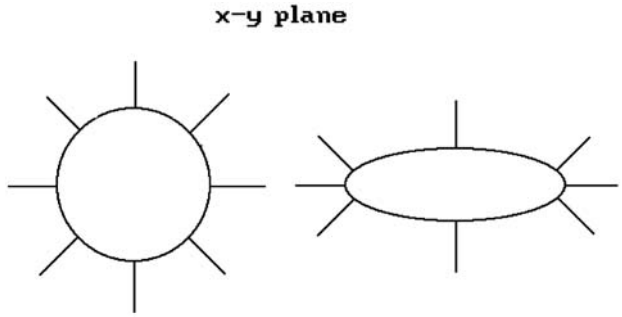


Figure 7.18. Using curve normals for subdivision.

7.18. The computation for finding the points was similar to the way that silhouettes are found, except that now one allows other angles. More precisely, one solves

$$\theta_i = \tan^{-1}\left(\frac{\mathbf{n}_z}{\mathbf{n}_x}\right)$$

or

$$\mathbf{n}_x \sin\theta_i - \mathbf{n}_z \cos\theta_i = 0.$$

We can precompute the values $\sin\theta_i$ and $\cos\theta_i$. This gives a whole new set of edge trackers. All the edge trackers above must be linked together appropriately to get a polygonal approximation to the intersection curve. Links are created as we pass critical points. When we pass saddle points we get *breaks* that must be repaired by "sideways" iteration. One moves along level curves of constant y but changing θ. A new edge tracker is created.

Using this approximation, level curves at a scan line are a chain of edge trackers that can be specified by an index into a table of sines and cosines. The angle rotates and we would expect that the index changes by 1 as we move from one edge tracker to the next (except obviously when we pass from the end of the table to the beginning). There is one **other** case however. The angle θ can have local maxima or minima as we move around. See Figure 7.19. This occurs at inflection points of cross-section curves. Mathematically, if $\mathbf{v}$ is the tangent to the level curve, the directional derivative of θ in the direction $\mathbf{v}$ vanishes: $D_{\mathbf{v}}\theta = 0$. We need to create trackers to follow this function.

The x-sampling described above sometimes made the polygonal nature of the approximation obvious in the output. Nevertheless it seemed to be reasonably good and did not use an excessive amount of time to compute. For smooth shading, most of the time was spent in the y-iteration. Blinn's approach made surfaces look better than if one had used a polygonal approximation to them.

A Summary of the Blinn Algorithm. It is a scan line algorithm that generalizes polygon visible surface algorithms and involves solving equations determined by various geometric features, such as

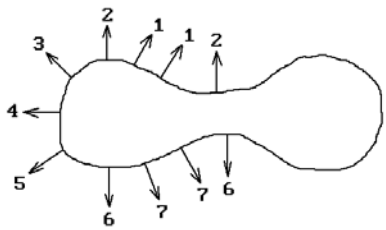


Figure 7.19. An inflection point case with angle trackers.

Feature	Equations	
boundary curves of patches	Y (0,v) = Yscan	Y (u,0) = Yscan
	Y (1,v) = Yscan	Y (u,1) = Yscan
silhouette edges	Y (u,v) = Yscan	$\mathbf{n}_z$ (u,v) = 0
local maxima or minima	∂Y/∂u (u,v) = 0	∂Y/∂v (u,v) = 0
segments of x-scan	Y (u,v) = Yscan	X (u,v) = Xscan

The equations are solved by making initial guesses and then using a Newton-Raphson iteration. One has a current active segment lists with segments being created and deleted and one sorts to find visible points. Blinn used heuristics to handle the many problems that arose.

We mention several other curved surface algorithms:

The Whitted Algorithm ([LCWB80])**.** This algorithm generalized the handling of surface patches bounded by straight lines to the case where boundaries are cubics. It avoided the problem of polygonal silhouettes but had problems with internal silhouettes and numerical techniques.

The Carpenter-Lane ([LCWB80]) and **Clark** ([Clar79]) **Algorithms.** These algorithms, like Catmull's algorithm ([Catm75]), use a subdivision technique that ends up subdividing patches until they are flat enough so that they can be approximated by planar polygons. The polygons are then scan converted using a standard approach. The approximation is as good as the view requires but not a priori. Unlike Catmull's algorithm, these are scan line algorithms. A problem that the algorithms have to worry about is that cracks can appear in the image if adjacent patches are not approximated carefully.

Blinn's algorithm is actually more general than either the Whitted or Carpenter-Lane algorithms.

7.11 Adding Antialiasing

So far in our discussion of visible surface algorithms, we have only touched on the aliasing issue. In fact, these algorithms often have antialiasing techniques built into them. Some general references that have a lot more material on antialiasing methods than we shall mention here are [MagT87], [FVFH90], [WatW92], and [Roge98].

We already mentioned antialiasing lines and polygons in Section 2.6. Antialiasing algorithms are incorporated into image precision algorithms, such as the Warnock algorithm, by a supersampling type approach. One subdivides to below a pixel and averages the subpixel values.

Early antialiasing efforts in scan line algorithms ([Crow77a], [Catm78]) led to the A-buffer algorithm that is used for antialiasing in Z-buffer algorithms. This algorithm originated with [Carp84] and was subsequently improved by [AbWW85]. A pixel is again assumed to have area, but rather than making complicated geometric clipping computations on this area one associates a bitmask to the pixel (Carpenter used a 4×8 bitmask) and one does the clipping via Boolean operations on the bits, which is much faster. The bitmasks are used in conjunction with a z-ordering of the polygon pieces. An edge fill scan conversion algorithm is used to enter the edges of the polygon pieces into the bitmask. The bitmasks are initialized to 0 and in the end have a certain number of 1's. A simple counting of the number of 1's determines the percentage of area of the polygon piece that is used to determine its contribution to the shade associated to the pixel. See [MagT87], [WatW92], or [Roge98].

For antialiasing in ray tracing, see Section 10.2.

7.12 Conclusions

We finish the chapter with some observations on visible surface algorithms. To begin with, note the importance of coherence. Basically, one thinks of a picture as consisting of regions all of whose pixels have the same value. Once we have the value of a pixel for a region, then we keep setting all adjacent pixels to the same value until we run into one of those "few" boundaries between regions where a change takes place. Below is a sampling of some different types of coherence guidelines when dealing with linear polyhedra.

Edge coherence: The visibility of an edge changes only when it crosses another edge. Therefore, we can create a list of edge segments without intersections and need only check one point per segment.

Face coherence: A face is often small compared to the whole scene. Therefore, usually the whole face is visible if one point of it is.

Object coherence: The use of bounding objects is a form of coherence.

Depth coherence: Different surfaces are usually well separated in depth.

Scan line coherence: Segments visible on one line are likely to be visible on the next.

Frame coherence: Images do not change much from frame to frame.

Geometric coherence: The possible visible/invisible configurations at any vertex are limited. For example, in a convex object the outside lines are visible (Figure 7.20(a)) as are all the edges surrounding a vertex (Figure 7.20(b)).

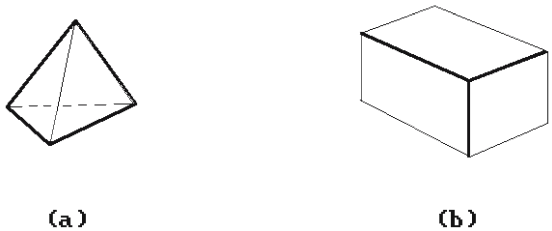


Figure 7.20. Examples of geometric coherence.

Curved objects satisfy similar types of coherence.

Visible surface determination algorithms need to be evaluated in the context of the whole rendering algorithm. There is no "best" such algorithm. Differences in algorithms stem from different requirements: different scenes, different complexity of output. The choice of an algorithm also depends on the shading algorithm that is to be used. For example, although the Watkins algorithm is a nice algorithm, one cannot use it if one wants to use a global reflectance model as described later in Chapters 9 and 10. In general though, image space algorithms seem to be the most popular. In fact, it is fair to say that Z-buffer algorithms, which are extremely simple and versatile, are the de facto standard in high performance graphics system because the price of memory is no longer such a problem. Furthermore, as mentioned earlier, because these systems typically have hardware support for fast display of polygons, all surfaces, including curved ones, are treated as collections of polygons. Many of the mathematical complications in algorithms like Blinn's are thereby avoided. Of course, having a high-speed system is now more important than ever because of the large number of polygons with which one has to deal.

The table below summarizes normalized speed results from Table VII in [SuSS74]:

	Number of facets in the scene		
Algorithm	100	2500	60000
Schumacker	30	179	1215
Newell-Newell-Sancha	1	10	507
Warnock	11	64	307
Z-buffer	54	54	54
Watkins	3	21	457

In the table we have normalized the Newell-Newell-Sancha algorithm to 1. The numbers are estimates only though and should mainly be used for comparison purposes. Small differences are inconclusive and only orders of magnitude are significant.

Depth sorting and scan line algorithms are good if there are only a small number of surfaces (up to several thousand). Scan line algorithms take advantage of coherence between successive scan lines and of span coherence within a scan line. They reduce a three-dimensional problem to a two-dimensional one. One danger though is that very narrow polygons can fall between scan lines. In general, depth sorting is good if the scene is spread out in depth (relative overlap in z-direction) and scan line or area-subdividing algorithms are good if surfaces are separated horizontally or vertically.

For scenes with very large numbers of objects, z-buffer methods or methods designed for objects encoded as octrees are good because their complexity is independent of the number of surfaces.

Sorting is an extremely important part in all the algorithms we discussed. In a sense the real difference between all the algorithms is just in the way they sort. The sorting technique chosen should match the statistical properties of the data being sorted. See [SuSS74] for how sorting affects algorithms.

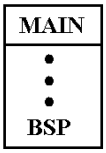
7.13 Programming Projects

Section 7.5

7.5.1 The BSP algorithm

Implement the BSP algorithm to display a world of blocks with solid colored faces. You will have to scan convert the projected polygons using the algorithm from Section 2.9.1 (see programming project 2.9.1.1). If you are using OpenGL, then one can simply use one of the OpenGL functions that fill in polygons.

Add the following item to the main menu



CHAPTER 8

Color

8.1 Introduction

This chapter covers some rudimentary facts about color and its role in computer graphics. Describing what one needs to know for a basic modeling system is not that hard. This is the main goal of this chapter, but we shall also sketch a more complete view of what color is all about. The reader who is unfamiliar with color and the issues surrounding it should be warned, however, that it is very easy to get confused in the terminology and what it really means because this is a big topic and the discussion here is extremely brief. A lot of terms having to do with the psychology of color do not have precise meanings (in fact, there is no universal agreement on some of them) and a real understanding comes only after extensive experience. For the reader who wants to learn more, a good general reference for color is [AgoG87]. Hall's book [Hall89] is a good place to look for computer graphics applications and [Roge98] is another.

The first three sections will cover general topics dealing with color. The last two discuss the main practical facts one needs to know in order to use color in a modeling program.

8.2 What Is Color?

What is color? This may seem like a silly question, but there are some common misconceptions, such as:

(1) Color is a property of materials (as in the case of a red dress).
(2) Color is a property of light (as in the case of a red traffic light).

Color is none of these, that is, there is, for example, no generic entity called "light" to which the property of "red" is added to get "red light." The fact is that under different lighting conditions, colors change. Newton already knew that light rays are not colored. Although the question about color is an old question, real progress has only

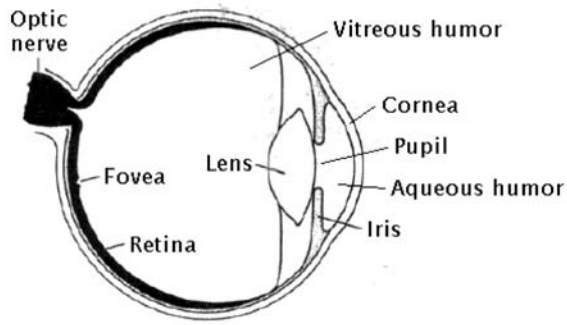


Figure 8.1. A horizontal cross-section of the eye.

been made relatively recently. Even today there is no complete answer. One needs to know more about the eye and the brain.

Some objects like the sun and a burning ember radiate light and are said to be *luminous*. Most objects that we see are *nonluminous*. We see them because light from some other source has reflected off them. Here are the basic events that happen when we look at an object ([AgoG87]). See Figure 8.1.

(1) The object's light enters the eye through the *cornea*.
(2) It passes through a clear liquid (*aqueous humor*), the *pupil*, the *lens*, and some jelly-like material (*vitreous humor*) before it falls on the *retina*.
(3) In the retina it passes through several layers of *nerve cells* before it is absorbed by *receptor cells*. There are two types of receptor cells, called *rods* and *cones*. There are three varieties of cones.
(4) The absorption causes some chemical changes leading to electrical changes that are then transmitted by *optic nerve* fibers from the eye to the opposite half of the brain.
(5) Finally, the brain responds by producing various sensations such as color, size, position, etc.

For an overview of the basic ingredients of the human visual processing system see [Ferw01].

We return to our original question: What is color? There are two possible answers:

(1) It is a sensation *produced* by the brain (the "perceived" color) in response to light received at the retina, so that one would say, for example, "the color produced by some given light is red or whatever."
(2) It is an *arbitrary* definition by specialists in colorimetry, the measure of color.

The next two sections attempt to explain these answers. We rely heavily on [AgoG87].

8.3 Perceived Color

Because a color is often influenced by surrounding colors, how can one judge a color from a nonluminous object in isolation? One way that this can be done is to illumi-

nate the object in a dark room. Another is to view it through a hole in a black panel while focusing on the perimeter of the hole. In the latter case, the perceived colors are called *film* or *aperture colors*.

Perceived colors have a number of characteristics. In the case of film colors, the simplest case, there are only three, namely, hue, saturation, and brightness:

Perceived hue:

Looking at red light we experience a sensation of a red "hue." It is hard to say exactly what hue is. The problem is similar to trying to describe the sensation of bitterness or shrillness in voice. It can also be thought of as the "color" of the color by which the color is designated. Colors are subdivided into

chromatic colors – those perceived colors that have hue
achromatic colors – those perceived colors that do **not** have hue (for example, the colors from a fluorescent lamp)

There are four basic hues (the *unitary* or *unique hues*): unitary red, yellow, green, and blue. All other hues are mixtures.

Perceived saturation:

This is the perception of the relative amount of a hue in a color and can be thought of as a number between 0 and 1. Since light can be thought to have two components, a chromatic and an achromatic one, a working definition of saturation is as the ratio of the chromatic to the sum of the chromatic and achromatic components of a color. For example, pink has a lower chromatic component than red. Saturation measures how much a pure color is diluted by white.

Perceived brightness:

Brightness is an attribute of the illumination in which a nonisolated object is viewed. It is a "perception of the general luminance level." Brightness applies to the color of an object only when the object is isolated and the light comes to the eye from nowhere else. One generally talks about it as ranging from "dim" to "dazzling."

Perceived colors other than film colors have additional characteristics such as:

Perceived lightness:

This is an attribute of a nonisolated color produced in the presence of a second stimulus. One uses terms such as "lighter than" or "darker than" for this.

Perceived brilliance:

This is perceived only when the object is not isolated as, for example, in the case of an area of paint in a painting or a piece of glass among others in a stained glass window.

8.4 Colorimetry

The terms in the last section apply to *color response* and hence are terms of psychology. There is a more scientific approach to analyzing color called colorimetry.

We begin with the question: What is light? Well, light is a form of energy, *electromagnetic energy*. Note that sometimes a distinction is made between *light*, that radiation in proportion to its effectiveness in producing vision, and *visible radiant energy*, which refers to all radiation in the visible range. Only part of all light is visible. Light has wavelengths given in nanometers (nm), where 1 nanometer equals 10^{-9} meters or, equivalently, 10 angstroms. Visible light is the light with wavelengths in the 380 nm to 780 nm range. For example, blue and green correspond to light with wavelengths of 470 nm and 500 nm, respectively.

Here is some more terminology related to light and perceived colors:

Monochromatic light:	This is light consisting of a single wavelength.
Spectral hues:	These are the hues in monochromatic visible radiation (those present in the sun's spectrum). The hues in a rainbow are spectral hues.
Spectral colors:	These are all the colors that are perceived to have a spectral hue.

Monochromatic light has maximum saturation but may have an achromatic component. We experience also *nonspectral* hues, for example, purple or purplish red, which are not present in the sun's spectrum, and *nonspectral* colors.

Because perceived colors vary with illumination, one prefers to make comparisons in daylight. But this must be specified **precisely**. In 1931 the Commission Internationale de l'Éclairage (CIE) defined the *CIE illuminants*. Colors can now be measured by matching against standard samples under standardized viewing conditions using *colorimeters*. Here a photoelectric cell replaces the human eye. One takes a sample color (the color one is trying to identify) and compares it to a mixture of standard colors. The intensities of the latter are varied via three adjustments, the CIE tristimulus values. The *tri-stimulus theory of light* is based on the assumption that there exist three types of color sensitive cones at the center of eye which are most sensitive to red, green, and blue. The eye is most sensitive to green.

Basically, one chooses three beams (a short, medium, and long wave length which are typically the three *additive primary colors* red, green, and blue). Different colors can then be produced by shining the three beams on a spot and varying the intensity of each. The *chromaticity* of a color **C** is defined by a triple (x,y,z) of numbers specifying these three intensities, or weights, for the color. Mathematically, one is representing the color **C** in the barycentric coordinates form

$$\mathbf{C} = \mathrm{x}\,\mathbf{R} + \mathrm{y}\,\mathbf{G} + \mathrm{z}\,\mathbf{B},$$

where **R**, **G**, and **B** represent the colors red, green, and blue, respectively, and x + y + z = 1. The numbers x, y, and z would be called the *chromaticity values* of **C**. Figure 8.2(a) shows this in graphical form. The triangle is called Maxwell triangle chro-

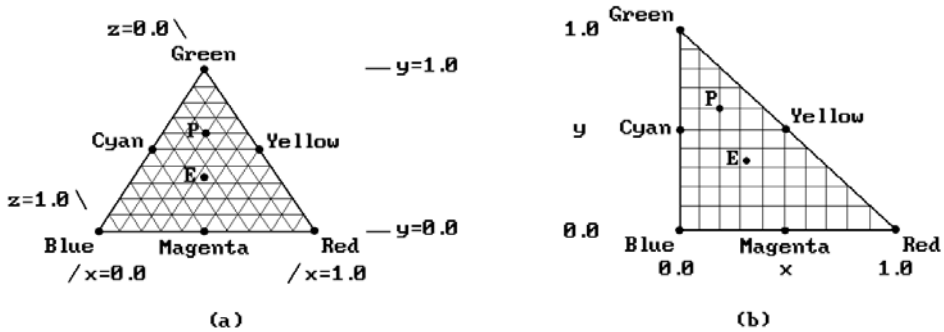


Figure 8.2. Maxwell's triangles.

maticity diagram (the equal-sided case). This was used by Maxwell in his work on color. The condition imposed on the selection of colors to serve as primaries is that in equal amounts they produce white (at **E** in Figure 8.2(a)). The point **P** corresponds to (x,y,z) = (0.2,0.6,0.2). Because the equal-sided triangle is awkward to employ in practice, one prefers a right triangle. (We do not need z anyway.) See Figure 8.2(b).

Now it is known that not all colors can be represented as the sum of three primary colors. What about a color that cannot be so represented? Well, we can add one of the primary beams to this color to bring it into the range of colors we **can** represent. This corresponds to using a negative coefficient. Although one could use negative coefficients for the representation of colors in terms of primary colors, for practical reasons, it was decided by the CIE to use a scheme in which negative numbers do **not** arise. Since this is **not** possible with real primaries, *imaginary primaries* were invented (they are called that because they are not visible) called *imaginary red*, *imaginary green*, and *imaginary blue* and denoted by **X**, **Y**, and **Z**. Then **every** color **C** can be written as

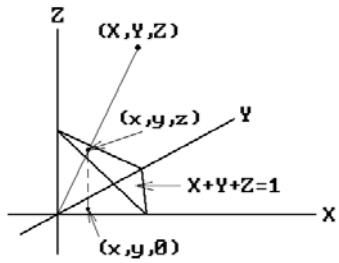
$$\mathbf{C} = \mathrm{X}\,\mathbf{X} + \mathrm{Y}\,\mathbf{Y} + \mathrm{Z}\,\mathbf{Z}, \quad \mathrm{X}, \mathrm{Y}, \mathrm{Z} \geq 0.$$

In this way one can represent colors as triples (X,Y,Z) in a three-dimensional space (actually its first octant). A further step is taken. Rather than using arbitrary triples (X,Y,Z), one normalized the numbers and uses (x,y,z), where

$$\mathrm{x} = \frac{\mathrm{X}}{\mathrm{X}+\mathrm{Y}+\mathrm{Z}} \quad \mathrm{y} = \frac{\mathrm{Y}}{\mathrm{X}+\mathrm{Y}+\mathrm{Z}} \quad \mathrm{z} = \frac{\mathrm{Z}}{\mathrm{X}+\mathrm{Y}+\mathrm{Z}}.$$

This corresponds to using the intersection of the plane X + Y + Z = 1 and the line through the origin and (X,Y,Z). See Figure 8.3. The numbers x, y, and z are called the *chromaticity values* for **C**. Since x + y + z = 1, one only really needs x and y. Note however that we cannot determine the original point (X,Y,Z) from just x and y. We need some additional information and the Y value is used. In the end, therefore, the

Figure 8.3. The chromaticity values of a color.



CIE color specification for the color **C** based on chromaticity is written CIE (x,y,Y) (rather than (X,Y,Z)).

Now the tri-stimulus values of this system really have only **relative** meaning because we do not know what the units are when a color is measured. The CIE specification gives them an absolute meaning by interpreting Y appropriately. What one does is to observe that an important characteristic of imaginary red and blue is that they have zero luminance. Luminance is essentially the intensity. By assigning the luminance, which can be obtained by a separate measurement, to Y (the imaginary green value), the tristimulus values can now be given an **absolute** meaning. For example, for the tuple (X,Y,Z) = (1000,800,1200), if the measurement for the luminance gave Y = 200, then one would adjust the X and Z in proportion and use the tuple (250,200,300) instead. By convention, the tri-stimulus values of a color are specified with respect to Y = 100 and if we had Y = 200, that that value would be reported separately.

The CIE chromaticity diagram shown in Figure 8.4 is based on test data for a narrow 2 degree angle of vision. The diagram is a two-dimensional projection of three-dimensional color space which plots the x and y values of the CIE (x,y,Y) specification of a color. The horseshoe region corresponds to the visible colors. A slightly different diagram generated in 1964 applied to a 10 degree angle of vision, but is considered less useful for computer graphics.

8.5 Color Models

The range of colors produced by an RGB monitor is called its *gamut*. There are a number of models for the gamut of an RGB monitor.

The Color Cube or RGB Model. This is the "natural" gamut model that represents the gamut as the unit cube $[0,1] \times [0,1] \times [0,1]$. See Figure 8.5.

The CMY Model. This model uses the *subtractive primary colors* cyan, magenta, and yellow, which are the complements of red, green, and blue, that is,

$$\begin{pmatrix} \mathbf{C} \\ \mathbf{M} \\ \mathbf{Y} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{pmatrix}.$$

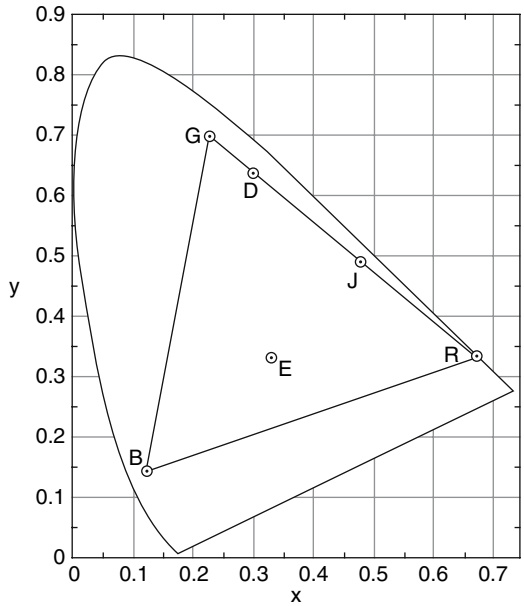


Figure 8.4. The CIE chromaticity diagram.

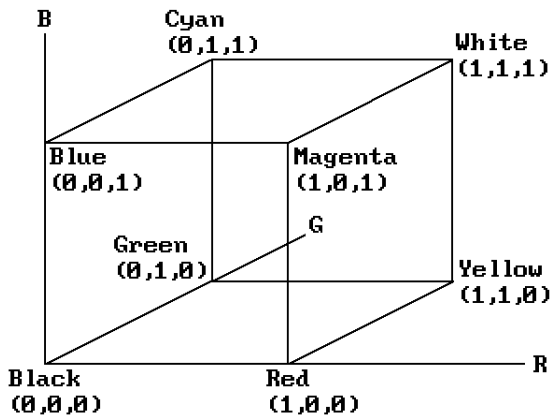


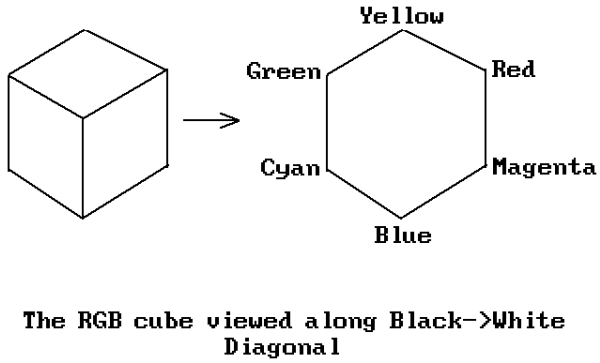
Figure 8.5. The RGB color cube.

The colors are called "subtractive" because they tell one what has to be subtracted from white (rather than what has to be added to black) to get the color. The CMY model is used with reflective sources and for devices like ink-jet plotters.

The YIQ Model. This model is defined by the equation

$$\begin{pmatrix} \mathbf{Y} \\ \mathbf{I} \\ \mathbf{Q} \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix} \begin{pmatrix} \mathbf{R} \\ \mathbf{G} \\ \mathbf{B} \end{pmatrix}.$$

Figure 8.6. The HSV hexcone.



See [Blin93]. We get a new space of primaries called the *transmission* primaries, which were recommended by the National Television System Committee (NTSC) in 1953 as the basis for generating color television broadcast signals in the United States. The value Y is called the *luminance* and measures brightness. It is the only signal received by a black and white TV. This is a good basis with respect to properties of RGB phosphors and the "standard" observer. The triangular region in the chromaticity diagram shown in Figure 8.4 is the possible gamut of the 1953 NTSC recommended RGB monitor, although actual monitors have a smaller triangle.

Although the RGB model is very simple mathematically, getting a desired color by varying the R, G, B guns of a monitor is not so simple. Artists especially would find the RGB model very unintuitive. For that reason other models were introduced.

The HSV (Hue-Saturation-Value) "Hexcone" Model. We get this model by looking down at the color cube along its main diagonal (see Figure 8.6) and recoordinatizing (see Figure 8.7). A color is now specified by three numbers h (hue), s (saturation), and v (value). The hue corresponds to an angle from 0 to 360 degrees, where, for example, red is at 0 degrees and green at 120 degrees. The saturation s, $s \in [0,1]$, measures the departure of a hue from white or gray. The value v, $v \in [0,1]$, measures the departure of a hue from black, the color of zero energy. See Figure 8.8.

The hexcone model tries to mimic how artists work. Their way of working with color has been described as follows in [AgoG87]:

> "Artists choose a pure hue, or pigment, and lighten it to a *tint* of that hue by adding white or darken it to a *shade* of that hue by adding black, or in general obtain a *tone* of that hue by adding a mixture of white and black, that is, gray."

The HSL (Hue-Saturation-Brightness) Triangle Model. See Figure 8.9. The reason for not using the letter "B" is so as not to cause confusion with "blue." The hue is again specified as an angle between 0 and 360 degrees about the vertical axis and the saturation and brightness are values between 0 and 1.

The HSL model is good for color gradations found in nature.

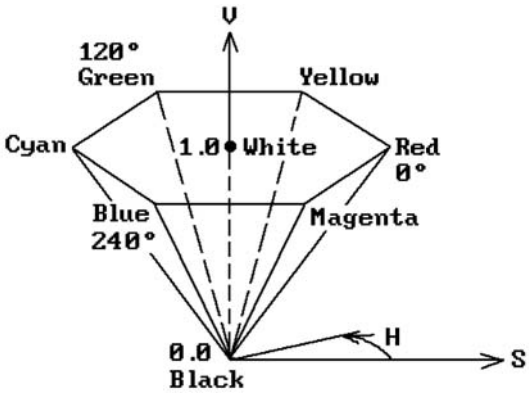


Figure 8.7. The single hexcone HSV color model.

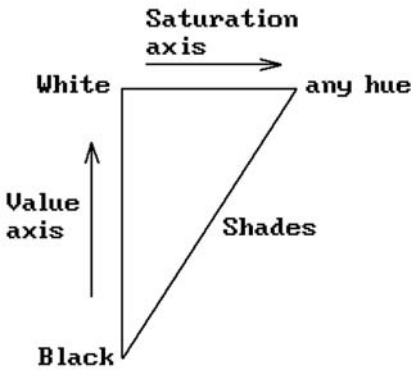


Figure 8.8. The HSV triangle.

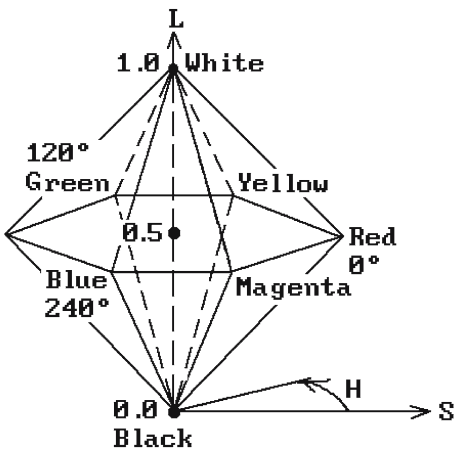


Figure 8.9. The double hexcone HSL color model.

8.6 Transforming Between Color Models

Finally, one needs transformations to convert between the various color models. Algorithms 8.6.1 and 8.6.2 describe the conversions between RGB and HSV. Algorithms 8.6.3 and 8.6.4 convert between RGB and HSL. Algorithm 8.6.4, which converts HSL to RGB, is the one described in [Fish90a]. It basically converts HSL to HSV and then uses the steps in the HSV to RGB conversion algorithm. An undefined hue in these algorithms means that the color is achromatic and corresponds to a shade of gray

```
UNDEFINED is a constant real number outside the interval [0,360].

procedure RGBToHSV (real r, g, b; ref real h, s, v)
{   Input:     r, g, b ∈ [0,1]
    Output:    h ∈ [0,360] , s, v ∈ [0,1] unless s = 0 in which case h = UNDEFINED
begin
    real max, min, d;

    max := Max (r,g,b);
    min := Min (r,g,b);

    v := max;                          { define the value }
    if max = 0                         { define the saturation }
        then s := 0
        else s := (max − min)/max;
    if s = 0
        then h := UNDEFINED
        else                           { the saturation is not zero (the chromatic case) }
            begin
                d := max − min;
                if r = max
                    then h := (g − b)/d;          { color is between yellow and magenta }
                    else if g = max
                        then h := 2 + (b − r)/d;  { color is between cyan and yellow }
                        else if b = max
                            then h := 4 + (r − g)/d; { color is between magenta and cyan }
                h := 60*h;                        { convert to degrees }
                if h < 0 then h := h + 360;       { prevent negative values }
            end
end;
```

Algorithm 8.6.1. Converting RBG to HSV.

UNDEFINED is a constant real number outside the interval [0,360].

```
procedure HSVToRGB (real h, s, v; ref real r, b, g)
{   Input:     h ∈ [0,360] or UNDEFINED,  s , v ∈ [0,1]
    Output:    r, g, b ∈ [0,1]
begin
    integer sextant;
    real    fract, p, q, t;

    if  s = 0
        then
            if  h = UNDEFINED
                then  begin  r := v;  g := v;  b := v;  end
                else  Error ()                          { invalid input }
        else
            begin
                if  h = 360  then  h := 0;
                             else  h := h/60;
                sextant := Floor (h);                   { h is now in [0,6) }
                fract   := h − sextant;                 { the fractional part of h }
                p := v*(1 − s);
                q := v*(1 − (s*fract));
                t := v*(1 − (s*(1 − fract)));
                case  i  of
                    0  :  begin  r := v;  g := t;   b := p;  end;
                    1  :  begin  r := q;  g := v;  b := p;  end;
                    2  :  begin  r := p;  g := v;  b := t;  end;
                    3  :  begin  r := p;  g := q;  b := v;  end;
                    4  :  begin  r := t;  g := p;  b := v;  end;
                    5  :  begin  r := v;  g := p;  b := q;  end;
                end
            end
end;
```

Algorithm 8.6.2. Converting HSV to RGB.

UNDEFINED is a constant real number outside the interval [0,360].

```
procedure RGBToHSL (real r, g, b; ref real h, s, ℓ)
{   Input:     r, g, b ∈ [0,1]
    Output:    h ∈ [0,360] , s, ℓ ∈ [0,1]  unless  s = 0  in which case  h = UNDEFINED
begin
    real  max, min, d;

    max := Max (r,g,b);
    min  := Min (r,g,b);

    ℓ := (max + min)/2;                    { define the brightness }
    if  max = min
        then
            begin  s := 0;  h := UNDEFINED;  end
        else
            begin
                d := max − min;

                { define the saturation }
                if  ℓ ≤ 0.5  then  s := d/(max + min)
                            else   s := d/(2 − max − min);

                { define the hue }
                if  r = max
                    then  h := (g − b)/d;              { color is between yellow and magenta }
                    else if  g = max
                        then  h := 2 + (b − r)/d;      { color is between cyan and yellow }
                        else if  b = max
                            then  h := 4 + (r − g)/d;  { color is between magenta and cyan }
                h := 60*h;                             { convert to degrees }
                if  h < 0  then  h := h + 360;         { prevent negative values }
            end
end;
```

Algorithm 8.6.3. Converting RBG to HSL.

UNDEFINED is a constant real number outside the interval [0,360].

```
procedure HSLToRGB (real h, s, ℓ; ref real r, b, g)
{   Input:     h ∈ [0,360] or UNDEFINED,  s , ℓ ∈ [0,1]
    Output:    r, g, b ∈ [0,1]
begin
    real     v, min, sv, sextant, fract, vsf, mid1, mid2;
    integer  sextant;

    if s = 0 then                                  { the achromatic case }
        begin
            if h = UNDEFINED
                then begin r := ℓ; g := ℓ; b := ℓ; end
                else Error ()                      { invalid input }
        end;

    if ℓ ≤ 0.5 then v := ℓ*(1 + s)
               else v := ℓ + s − ℓ*s;
    if v = 0
        then
            begin r := 0; g := 0; b := 0; end
        else
            begin
                min := 2*ℓ − v;
                sv  := (v − min)/v;
                if h = 360
                    then h := 0
                    else h := h/60;
                sextant := Floor (h);              { h is now in [0,6) }
                fract   := h − sextant;            { the fractional part of h }
                vsf     := v*sv*fract;
                mid1    := min + vsf;
                mid2    := v − vsf;
                case sextant of
                    0 : begin r := v; g := mid1; b := min; end;
                    1 : begin r := mid2; g := v; b := min; end;
                    2 : begin r := min; g := v; b := min1; end;
                    3 : begin r := min; g := mid2; b := v; end;
                    4 : begin r := mid1; g := min; b := v; end;
                    5 : begin r := v; g := min; b := mid2; end;
                end
            end
end;
```

Algorithm 8.6.4. Converting HSL to RGB.

that lies along the central axis of the models. [Roge98] also describes a number of other conversion algorithms such as between RGB and CIE.

8.7 Programming Projects

Section 8.6

8.6.1 Write a program that lets a user define colors graphically by means of three sliders that correspond to the hue, saturation, and value parameters of the HSV hexcone model.

CHAPTER 9

Illumination and Shading

9.1 Introduction

Image synthesis has been defined as the creation of images using an illumination model for the propagation of light, with the goal being *photorealism*. Actually, photography operates under some constraints defined by the camera and film being used, so that a much more general goal would be to produce a visual experience that matches the one that one gets by viewing the real scene.

There are basically five steps to getting a realistic computer generated image of a scene:

(1) Modeling the scene
(2) Projecting to the image plane and transforming to the viewport including clipping
(3) Hidden surface and visibility computations
(4) Light intensity computations (shading) using illumination models
(5) Antialiasing

We have looked at all of these except (4). After the visibility of a surface has been determined, one uses a shading model to establish the light intensity and color at its points. These computations usually dominate the visibility computations unless the number of objects is very large. Several shading models are used in computer graphics. The two that we shall describe in great detail are the Phong model and ray tracing. Good references for additional information about this subject are [Hall89], [FVFH90], and [WatW92].

First of all, one distinguishes between an *illumination* or *reflectance model* and a *shading model*. The latter will be the more comprehensive term. A shading model **uses** illumination models. Two different shading models could use the same illumination model. For example, one shading model could compute the light intensity at **every** point according to a fixed illumination model and another might only compute them at **vertices** of a linear polyhedron (using that same illumination model) and then get the rest of the values with interpolation.

Anyone in geometric modeling will sooner or later have to choose an illumination model and one will have to decide between one of two approaches. On the one hand,

one can try to rigorously simulate the illumination process; on the other, one might be satisfied with achieving the **illusion** of realism. The first approach is an ideal that inevitably takes a lot of CPU cycles. The second allows one to take shortcuts that produce results quicker but hopefully still produce good images. Actual illumination models used in practice differ from theoretical derivations. Hall ([Hall89]) classifies them into three types: empirical, transitional, and analytical. The corresponding shading techniques that evolved from these three models are classified as being incremental, using ray tracing, or using radiosity methods, respectively. There are now also hybrid approaches using the last two techniques.

The first illumination models were empirical in nature. The illumination values were evaluated **after** the geometry was transformed to screen space and standard scan line incremental approaches were used. Transitional models used more optics. They used more object space geometry so that reflections, refractions, and shadows were geometrically correct. Ray tracing was started. Gradually, analytical models were developed. Hall describes the shift in approaches as "a shift in research from the hidden surface problem, to creating realistic appearance, to simulating the behavior that creates the appearance." He points out that as we look at how illumination and shading are dealt with, one finds that the two main approaches can be explained in terms of two questions. One approach starts at the eye, considers the visible surfaces, and asks for each **visible** pixel:

"What information is required to calculate the color for this surface point?"

Getting this information implies that other surfaces must be considered and so the same question is asked recursively. Ray tracing is an example of this approach. The second approach starts at the light sources, traces the light energy, and asks:

"How is this light reflected or transmitted by this surface?"

From this point of view, every illuminated surface becomes an emitter. This is how radiosity works.

The first approach where we start at the eye generates a **view-dependent** map of light as it moves from surfaces to the eye. Every new viewpoint calls for a new map. Initially, rendering programs used only the ambient and diffuse component of light ([Bouk70]), and then the specular component was added ([BuiT75]). This worked fairly well for isolated objects. Eventually, reflections were also dealt with, culminating in ray tracing. The ray-tracing program described by Kajiya ([Kaji86]) started at the eye and sent out 40 rays per pixel.

The second approach to rendering where we start from the light generates a **view-independent** map. This approach may seem extremely wasteful at first glance because one makes computations for surfaces that are not seen. However, starting at the eye gets very complicated if one wants to render diffuse surfaces correctly. Single reflections between n surfaces require n^2 computations. This number goes up very rapidly as one attempts to follow subsequent reflections. Furthermore, having made the computations once, one can then make as many images as one wants without any additional work.

Two important phenomena one has to watch out for when modeling light are:

(1) The interaction of light with **boundaries** between materials, and
(2) the scattering and absorption of light as it passes **through** material.

Therefore, the two basic ingredients to all models are properties of surfaces and properties of light. An important property of a surface is its **reflectance**. Different reflectances for different wavelengths cause color. Another property is **transparency**. Also, in order to model color accurately, one needs to maintain the wavelength or spectrally dependent information throughout the visible range. We shall indicate the dependence of a variable on wavelength by giving it a wavelength parameter λ. In practice we are only interested in the three wavelengths that correspond to the red, green, and blue colors of an RGB monitor, so that each of the intensity equations we shall see in this chapter really represent three equations, one for each of those three colors.

To summarize, several shading models are used in graphics ranging from the simple to the complex. The Phong reflectance model ([BuiT75]), which has been around since the mid-1970s, still seems to be the most popular of the local reflectance models, even though there have been many improvements (see [Glas95]). Phong shading together with texture maps and shadows does a pretty good job for local shading models. Furthermore, one can mix in ray tracing for good result. For really eye-catching images, full-blown ray-tracing methods, radiosity methods, or a combination of the two are the current candidates. Of course, the latter also take correspondingly more computer power.

Finally, although photorealism was the holy grail of computer graphics for many years, now that it has been pretty much achieved, it is no longer the driving force it once was and one is more concerned with finding innovative uses for this new medium.

Section 9.2 starts this chapter's topics with a discussion of local reflectance models. We then move on to shading models in Section 9.3, global reflectance models in Section 9.4, and tie things all together with the rendering equation in Section 9.5. Sections 9.6–9.8 describe ways of enhancing images without increasing geometric complexity. An overview of the whole rendering process is presented in Section 9.9. The chapter ends with a discussion of ways to deal with rendering colors when there are limitations on the size of the color palette.

9.2 Local Reflectance Models

Geometrical optics (or *ray theory*) treats reflected light as having three components, an ambient one and a diffuse and specular one. *Ambient* (or background) light is that light that is uniformly incident from the **environment** and which is reflected equally in all directions by surfaces. *Diffuse* and *specular* light is reflected light from **specific** sources. Diffuse light is light that is scattered equally in all directions and has two sources. It comes either from internal scattering at those places where light penetrates or from multiple surface reflections if the surface is rough. For example, dull or matte material generates diffuse light. Specular light is the part of light that is concentrated in the mirror direction. It gives objects their highlights.

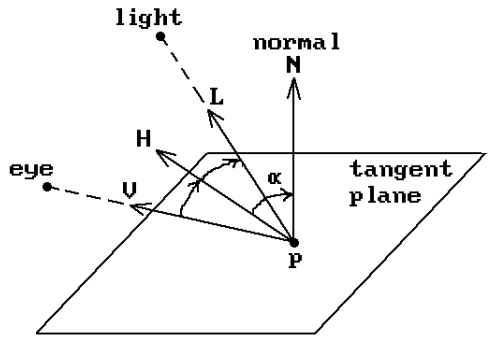


Figure 9.1. Some basic notation.

The following notation will be used throughout this chapter. See Figure 9.1. At each point **p** of a surface the **unit** normal to the tangent plane at that point will be denoted by **N**. The vectors **V** and **L** are the **unit** vectors that point to the eye (or camera) and the light, respectively. It is convenient to define another **unit** vector **H** which is the bisector of **V** and **L**, that is,

$$\mathbf{H} = \frac{\mathbf{V}+\mathbf{L}}{|\mathbf{V}+\mathbf{L}|}.$$

The angle between **H** and **N** will be denoted by α.

The simplest reflectance model ([Bouk70]) takes only ambient and diffuse light into account. The ambient component of the intensity is assumed to have the form

$$I_a(\lambda)k_a(\lambda),$$

where $I_a(\lambda)$ is the ambient light intensity at wavelength λ and $k_a(\lambda) \in [0,1]$ is the *ambient reflection coefficient*, which is the proportion of ambient light that is reflected. The diffuse component is assumed to have the form

$$I_p(\lambda)k_d(\lambda)r_d,$$

where $I_p(\lambda)$ is the intensity of the point light source reaching the point **p**, $k_d(\lambda) \in [0,1]$ is the *diffuse reflection coefficient*, which is a constant that depends on the material, and r_d is the *diffuse reflectance factor*. The factor r_d is computed from Lambert's law for ideal diffuse reflectors which states that they will diffuse incident light **equally in all directions**. Consider Figure 9.2. An area A_1 of light incoming along a direction **L** will shine on an area A_2 in the plane with normal **N**. If θ is the angle between **N** and **L**, then it is easily shown that

$$\frac{A_1}{A_2} = |\cos\theta| = |\mathbf{N}\bullet\mathbf{L}|.$$

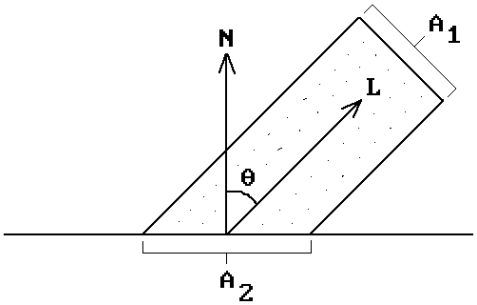


Figure 9.2. Diffuse reflection geometry.

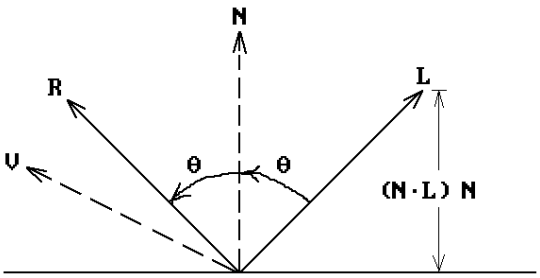


Figure 9.3. Specular reflection geometry.

The ratio A_1/A_2 specifies the part of the light that will diffuse equally in all directions, but there is one other assumption. Any light coming from **behind** the surface will be assumed not to contribute any diffuse light. To put it another way, any surface that faces away from the light, that is, where $\mathbf{N}\bullet\mathbf{L} < 0$, contributes nothing. It follows that $r_d = \max(\mathbf{N}\bullet\mathbf{L},0)$.

Putting this all together gives us

$$\textit{Bouknight's reflectance model}: \quad I(\lambda) = I_a(\lambda)k_a(\lambda) + I_p(\lambda)k_d(\lambda)r_d \tag{9.1}$$

Next, we want to include a specular component into our intensity function. See Figure 9.3. Given a light ray **L** and the normal **N** to a plane, the formula for the "mirror direction" **R** of a light ray hitting the plane is clearly

$$\mathbf{R} = \mathbf{L} - 2(\mathbf{L} - (\mathbf{N}\bullet\mathbf{L})\mathbf{N}). \tag{9.2}$$

One assumes that the specular component has the form

$$I_p(\lambda)\, k_s r_s,$$

where r_s is the specular reflectance factor and the *specular reflection coefficient* k_s is really a function of the *angle of incidence* θ but is usually set to a constant between 0 and 1. In the case of a perfect mirror,

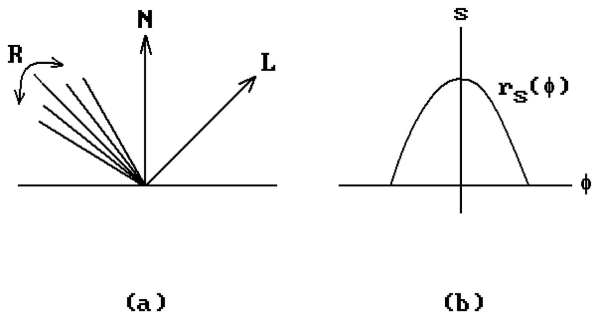


Figure 9.4. More realistic specular drop off.

$$\begin{aligned} r_s &= 1, \quad \text{if } \mathbf{V} = \mathbf{R} \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

This is not realistic. There is a specular contribution also when **V** is close to **R**. See Figure 9.4(a). Phong ([BuiT75]) used the angle ϕ between **R** and **V** to control the decrease in the intensity of reflected light as one moves away from the mirror direction. Specifically, he used a power m of $\cos\phi = \mathbf{R}\bullet\mathbf{V}$ to adjust the sharpness of the highlights and defined r_s by

$$r_s = r_s(\phi) = \max(\cos^m \phi, 0) = \max\left((\mathbf{R}\bullet\mathbf{V})^m, 0\right) \tag{9.3}$$

Phong's specular reflectance model: $I(\lambda) = I_a(\lambda)k_a(\lambda) + I_p(\lambda)(k_d(\lambda)r_d + k_s r_s)$ (9.4)

Although some books say that the constant exponent m measures the "shininess" of a surface, with a large m corresponding to glossy surfaces and a small one to dull surfaces, a better visual description would be that it changes the apparent size of the light by making the highlight smaller or larger, respectively. Typical values of m range from 50 to 60. Note that the specular reflection coefficient k_s is **not** a function of wavelength. This is so that highlights appear to have the color of the source.

In practice, one makes some simplifications to Phong's model. First of all, one approximates the ambient term by a constant. Another simplification is made if the light source and viewer are at infinity. In this case, one replaces the angle ϕ by the angle α between the normal **N** and **H**. The justification for this is that **H** specifies the direction that the surface normal should be for **V** to be the mirror direction. When that happens, then $\phi = 2\alpha$. Therefore, we can compensate for the difference in the dot products by changing the power m. The nice consequence of this is that since **L** and **V** are constant, **H** is also constant, and so we save ourselves some computations. This leads to the following ([Blin77]):

A simplified Phong specular reflectance model:

$$I(\lambda) = \text{ambient constant} + I_p(\lambda)\left(k_d(\lambda)(\mathbf{N}\bullet\mathbf{L}) + k_s(\mathbf{N}\bullet\mathbf{H})^m\right) \tag{9.5}$$

Phong's model was derived empirically. It produced results that were better than those using previous methods. To get better results yet, one needs to go back to theory. Experimentally one has a pretty good match to Phong's model except that the peak

specular direction was incorrect when the angle of incidence for the light was small. Torrance and Sparrow ([TorS67]) worked out a theoretical model that was very accurate. Their model predicted the directional and spectral composition of reflected light. Phong's model often made objects look plastic, and the new model avoided this plastic look for metals. Mathematically, the model differs from Phong's by replacing the factor r_s with another more complicated one.

The Torrance-Sparrow specular reflectance model ([Blin77]):

Hypothesis. The surface being modeled consists of collection of many small mirror-like faces ("micro facets") oriented in a random manner. The specular component is assumed to come from the reflection in those faces oriented in the direction **H**.

Conclusion.

$$r_s = \frac{DGF}{\mathbf{N}\bullet\mathbf{V}}, \quad \text{where} \tag{9.6}$$

D is the distribution function of the directions of the small faces,
G is the amount by which these faces shadow and mask each other, and
F is from the Fresnel reflection law.

Here is a more detailed description of the factors:

D: This term measures the proportionate number of facets oriented at an angle α from the average normal to the surface. Torrance and Sparrow used a simple Gaussian distribution and the formula

$$D = \exp\left(-(\alpha c_2)^2\right),$$

where c_2 is the standard deviation of the distribution and $\alpha = \cos^{-1}(\mathbf{N}\bullet\mathbf{H})$. The c_2 depends on the surface and plays the role of m in Phong's model.

$1/(\mathbf{N}\bullet\mathbf{V})$: Since intensity is proportional to the number of facets pointing in the direction **H**, we see more of the surface if the surface is tilted. The increase is inversely proportional to the cosine of the tilt, leading to this term.

G: This term counteracts the previous one. Some facets shadow others. There are three cases if we assume that facets are V-shaped grooves and angles are equal but opposite about **N**. See Figure 9.5. See [Blin77] for a mathematical justification for the approximation that

$$G = \min(G_{(1)}, G_{(2)}, G_{(3)}), \quad \text{where}$$

$G_{(1)} = 1$ (case of no interference)

$G_{(2)} = 2\dfrac{(\mathbf{N}\bullet\mathbf{H})(\mathbf{N}\bullet\mathbf{V})}{\mathbf{V}\bullet\mathbf{H}}$ (case where the reflected light is blocked)

$G_{(3)} = 2\dfrac{(\mathbf{N}\bullet\mathbf{H})(\mathbf{N}\bullet\mathbf{L})}{\mathbf{V}\bullet\mathbf{H}}$ (case where the incident light is blocked)

Note that $0 \le G_{(i)} \le 1$.

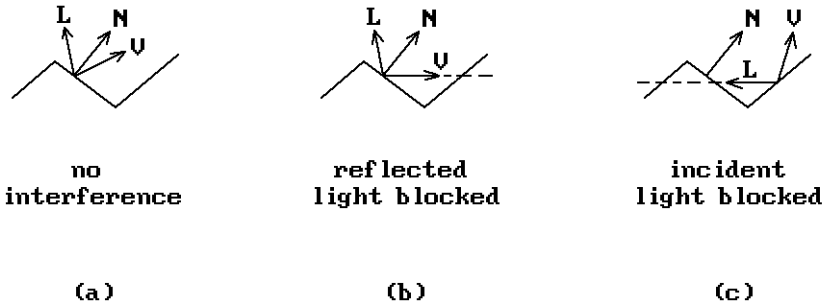


Figure 9.5. Facets shadowing others.

F: This term corresponds to the fraction of light actually reflected rather than absorbed, which, using the Fresnel reflection law, is a function of the *angle of incidence* and *index of refraction* n.

$$F = \frac{(g-c)^2}{(g+c)^2}\left[1 + \frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}\right] \tag{9.7}$$

where $c = (\mathbf{V} \bullet \mathbf{H})$ and $g = n^2 + c^2 - 1$.

Other Ds have been used which are simpler in order to offset the computation for G and F. See [Blin77] for more details.

In the discussion of reflectance models above we assumed a single-point light source. For multiple light sources one adds the diffuse and specular contribution of each but uses only one ambient term.

Finally, one fact that has not been touched on so far is how the distance between an object and a light source affects its intensity. If one does not take distance into account, then two equally sized spheres, one of which is much further away from the light than the other, would be visually indistinguishable in a picture, which is not what one would experience in real life. Laws of physics imply that light intensity is inversely proportional to the square of the distance from the source. It turns out, however, that if one models this effect, a lot of the time the pictures do not come out right and so this factor is often ignored. Sometimes a factor inversely proportional to the distance (rather than its square) is used. There are times when one wants to assume that the light is "infinitely" far away (the rays are all parallel) and, in that case, clearly distance has to be ignored otherwise one would have zero intensity. The sun shining on a scene is an example of this. If one wants to add a distance factor f, then one should use an intensity equation of the form

$$I = I_a k_a + f\, I_p (k_d r_d + k_s r_s), \tag{9.8}$$

where f could be $1/D^2$, D being the distance.

9.3 Simple Approaches to Shading

The last section described some simple illumination models and how one can use them to compute the illumination at each point of an object. In this section we show how this information is used to implement shading in a modeling program. The details depend on

(1) the visible surface determination algorithm that is used,
(2) the type of objects that are being modeled (linear or curved), and
(3) the amount of work one is willing to do.

Constant Shading. The simplest way to shade is to draw all objects in a constant color. The visible surface algorithms in Chapter 7 would then suffice to accomplish that. A more sophisticated model would draw each polygon in the scene with a constant shade determined by its normal and one of the described illumination models.

Constant shading that used facet normals would work fine for objects that really were linear polyhedra, but if what we were doing was approximating curved objects by linear polyhedra, then the scene would look very faceted and not smooth. To achieve more realistic shading, we first consider the case of a world of linear polyhedra.

Gouraud Shading. Gouraud's scan line algorithm ([Gour71]) computes the illumination at each vertex of an object using the desired illumination model and then computes the illumination at all other points by interpolation. As an example, consider Figure 9.6. Assuming that the illumination is known at the vertices **A**, **B**, **C**, and **D**, one computes it at the point **X** as follows: Let **P** and **Q** be the points where the edges **AC** and **BD** intersect the scan line containing **X**, respectively. Compute the illumination at **P** and **Q** by linearly interpolating the illumination values at **A**, **C**, and **B**, **D**, respectively. The value at **X** is then the linear interpolation of those two values.

To get the exact illumination values at vertices, normals are needed. These normals are typically computed by taking the average of adjacent face normals. One needs to realize though that doing this will have the effect of smoothing out an object. This is what one wants if our objects are faceted representations of curved objects. On the other hand, sometimes one wants to show edges, as in the case of a cube, and then we need to avoid this smoothing operation. When we shade such a face, the normals of the vertices for that face should be the same as the face normal.

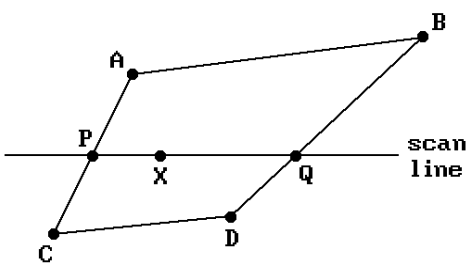


Figure 9.6. Gouraud shading example.

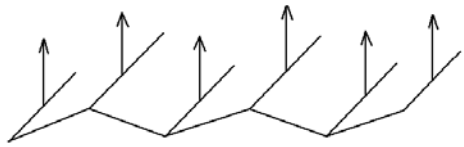


Figure 9.7. Gouraud shading anomaly.

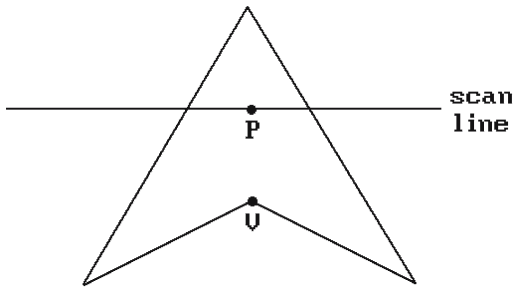


Figure 9.8. Phong shading anomaly.

Some anomalies can occur with Gouraud shading. Figure 9.7 shows a faceted surface which might have been an approximation to a wavy surface, but whose Gouraud shading would make it look flat because of the averaging of the normals. The simplest way to get around that is to subdivide the given facets more. In the case of Figure 9.7, we could divide each rectangular face into four subfaces.

There are other problems. If one uses Phong's reflectance model, then small variations in the normal can produce **large** variations in the specular reflection. This causes Gouraud's method to produce peculiar highlights.

Phong Shading. To remedy some of the problems with Gouraud shading, Phong ([BuiT75]) interpolated the surface normals (rather than the intensities) at vertices across faces and then computed the illumination at each point directly from the normals themselves. This clearly takes more work, however. In particular, generating unit-length vectors means taking square roots, which is costly in time. To lessen this cost, one can use a lookup table and linear interpolation ([Duff79]). Alternatively, one can use a Taylor expansion to get a good approximation ([BisW86]). The latter approach produces pictures indistinguishable from real Phong shading at a cost that is not much greater than that of Gouraud shading.

Phong shading produces better results than Gouraud shading, but it also has problems. Consider the concave polygon in Figure 9.8. The difference between the interpolated normal at the point **P** and the normal at the vertex **V** could cause a big change in the illumination value as one moves from **P** to **V**. This again shows the importance of sampling properly.

Gouraud and Phong shading basically assumed a scan line visible surface algorithm approach. In that context, one can speed up the process by computing the shading equations incrementally.

9.4 Global Illumination Models

The problem with the simple models discussed in Section 9.2 is that they are only **local** illumination models in the sense that they totally ignore the contribution of other objects in the world. We have assumed that the light comes to a point directly from a single source (with **no** shadows) and have dealt only with reflections from a **single** surface when in reality light often reflects from **several** surfaces before reaching the eye. Transparency properties have also been ignored. This section looks at some global aspects of illumination and how to deal with them.

9.4.1 Shadows

In trying to produce realistic pictures, one will quickly discover that they do not look that great if we omit shadows. A good discussion of shadow algorithms can be found in [WoPF90] and [WatW92]. The algorithms are classified into the following approaches:

(1) as part of early scan line visible surface algorithms,
(2) as part of the Weiler-Atherton visible surface algorithm ([AtWG78]),
(3) using "shadow volumes" ([Crow77b]),
(4) using "shadow z-buffers" ([Will78]).
(5) as part of ray tracing, and
(6) as part of radiosity methods.

We shall not go into the first two of these approaches. Approach (2) amounted to running the basic Weiler-Atherton algorithm twice and involved lots of clipping of polygons.

In the shadow volumes approach one extends each edge of an object that is an outline. See Figure 9.9. The volume between the tetrahedron **ABCD** and its projection **A′B′C′D′** from the light source on some fixed distant plane is called the *shadow volume generated by the object* **ABCD**. The faces obtained in this way bound a volume in which light has been obscured. For several light sources we get several such which are tagged by the light source. Real polygons **along** with these shadow ones are passed to the visible surface determination algorithm. Potentially many *shadow polygons*, that is, faces of shadow volumes, will lie between the viewpoint and a surface. One uses a

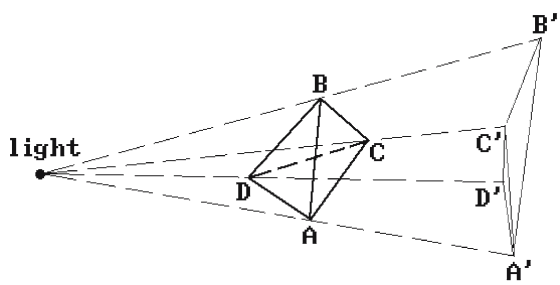


Figure 9.9. Shadow volumes.

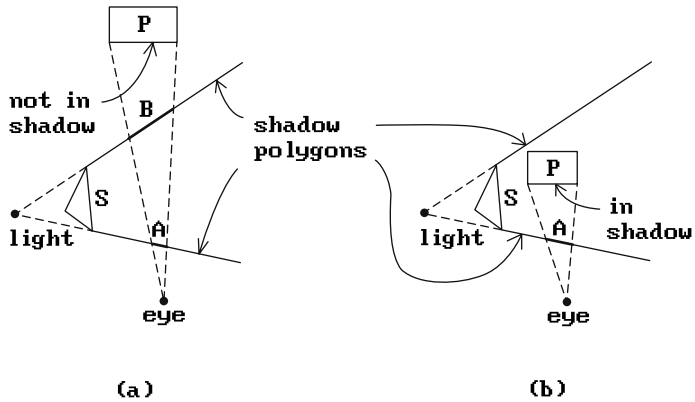


Figure 9.10. Using parity with shadow polygons.

parity algorithm to determine if a point on a surface is in a shadow. If an even number of polygons with the same tag are encountered, then the point is **not** shadowed by that light source, otherwise it is. Figure 9.10 shows how the parity works for spans associated to a polygon **P**. In Figure 9.10(a) object **P** is **not** in the shadow of **S** because there are two shadow polygons determined by **A** and **B** that lie between it and the eye. In Figure 9.10(b) object **P is** in the shadow of **S** because only a single shadow polygon, the one containing **A**, lies between it and the eye. Although we have only a single pass through the visibility algorithm when using shadow volumes, there are more objects and they are more complex.

The shadow z-buffer approach to shadows first determines the visibility of objects with respect to the light source and then uses this information in a second pass that determines the visibility with respect to the viewer. This amounts to running the visible surface determination algorithm twice. The approach has the advantage of being easy to integrate into a standard z-buffer algorithm. One uses an extra buffer, called the *shadow z-buffer*, for each light source. (One can get away with only one buffer for multiple light sources by running the basic algorithm once for each light source.) In the case of a single light source, here are the two steps in this algorithm:

Step 1. One runs a z-buffer type algorithm from the point of view of the light source to "render" the scene into the shadow z-buffer. In this pass only the depths of points are recorded and no light computations are made.

Step 2. One runs the standard z-buffer algorithm from the viewer, but with the following change: Before storing any light information into the frame buffer for a visible point, the coordinates of that point are first transformed into the light source coordinate system and its distance from the light is compared with the value stored for that position in the shadow z-buffer. If its value is larger than the stored value, then the point is in a shadow and the frame buffer value is not updated.

Ray-tracing and radiosity methods will be described shortly and in the next chapter. Computing shadows in the context of ray tracing is very easy. Basically, once

we have determined a visible point, all we have to is to send out a ray toward the light source and see if it hits an object on the way. Radiosity methods handle shadows by in large automatically, except some extra processing is required along the outlines of shadows.

Finally, we need to point out that one needs to distinguish between *hard* and *soft shadow algorithms*. We have discussed the former that simply determine in a "true" or "false" manner whether a point is in a shadow. One associates either 0 or the normal light intensity value to the point. Because of this, all that is involved is a visibility determination of points in the *umbra* region of a shadow (the points that receive no light). A soft shadow algorithm also includes a *penumbra* region in the computation (the points that receive partial light). We refer the reader to the above-listed references to see how algorithms can be modified to accomplish this. The references also touch on the much more difficult problem of shadows when there are transparent objects present.

9.4.2 Transparency

There are two ways to model transparency. The simple way is to ignore refraction and pretend that light will pass straight through an object without any bending. The other is to take refraction into account.

Using the simple model with no refraction, transparency can be implemented by letting surfaces only partially obscure surfaces behind them. If a painter's algorithm is used in the visible surface determination algorithm, then one can overwrite (in the back-to-front manner) each face on the **current** background with a blending function of the type

$$I = (l - k_t)I_f + k_t I_b, \tag{9.9}$$

where I_f and I_b are the illumination associated to the face and **current** background, respectively, and k_t is a transparency factor. As usual, there is a separate equation (9.9) for each wavelength. One problem with this approach is that one loses any specular reflection. To avoid this one can restrict the interpolation to the ambient and diffuse component of the light and then add the specular component from the front object. Adding transparency in this way greatly increases the cost of the rendering algorithm because every surface is painted whether visible or not.

Using refraction in the transparency model increases the cost even further because light no longer travels in a straight line and one has to compute how it is bent. Consider a light ray passing from one medium to another at a point **p** on their common boundary. See Figure 9.11. Let **L** and **T** be the unit vectors associated to the light paths in the first and second medium as shown in the figure. Let θ_1 (the *angle of incidence*) be the angle between **L** and the unit normal vector **N** to the boundary surface at a point and θ_2 (the *angle of refraction*), the angle between **T** and **N**. Snell's law states that

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_1}{n_2}, \tag{9.10}$$

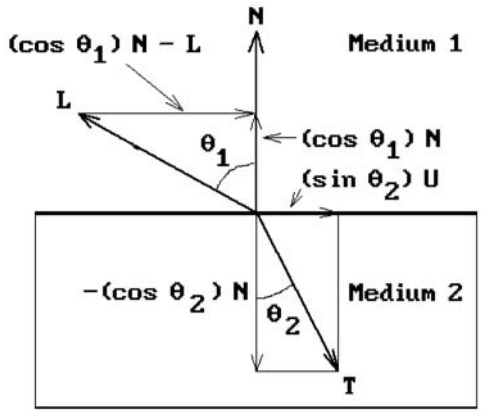


Figure 9.11. The basic refraction model.

where n_1 and n_2 are the *index of refraction* of the first and second medium, respectively. A medium's index of refraction is the ratio of the speed of light in a vacuum to the speed of light in the medium. The index of refraction varies with wavelength. For example, it is 1.0 if the medium is a vacuum and close to 1.0 in the case of air. For other material it is larger than 1.0. Water has an index of refraction of 1.33; crown glass, 1.52; and dense flint glass, 1.66.

To compute the refracted ray **T**, let **U** be the unit vector obtained by normalizing the orthogonal projection of **–L** onto the plane orthogonal to **N**. It is easy to see that

$$\mathbf{U} = \frac{(\cos\theta_1)\mathbf{N} - \mathbf{L}}{|(\cos\theta_1)\mathbf{N} - \mathbf{L}|} = \frac{1}{\sin\theta_1}((\cos\theta_1)\mathbf{N} - \mathbf{L}).$$

Since

$$\mathbf{T} = (\sin\theta_2)\mathbf{U} - (\cos\theta_2)\mathbf{N},$$

substituting the formula for **U** into this expression and rearranging terms gives

$$\mathbf{T} = (n_{12}\cos\theta_1 - \cos\theta_2)\mathbf{N} - n_{12}\mathbf{L},$$

where $n_{12} = n_1/n_2$. But

$$\cos\theta_2 = \sqrt{1-\sin^2\theta_2} = \sqrt{1-n_{12}^2\sin^2\theta_1} = \sqrt{1-n_{12}^2\left(1-(\mathrm{N}\bullet\mathrm{L})^2\right)},$$

so that

$$\mathbf{T} = \left[n_{12}(\mathbf{N}\bullet\mathbf{L}) - \sqrt{1-n_{12}^2\left(1-(\mathbf{N}\bullet\mathbf{L})^2\right)}\right]\mathbf{N} - n_{12}\mathbf{L}. \tag{9.11}$$

Note that formula (9.11) has a square root in it. This can be imaginary in certain circumstances. For example, if the index of refraction of the second medium is lower

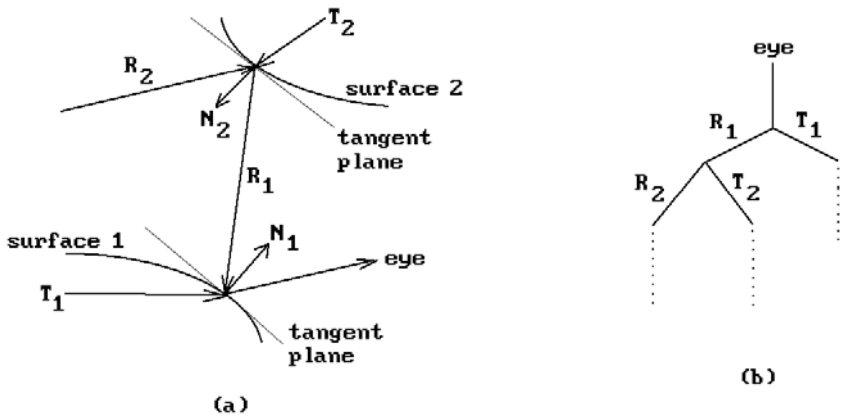


Figure 9.12. Basic ray tracing.

than that of the first medium, then θ_2 is larger than θ_1. It is therefore possible for θ_2 to become larger than 90°. In that case, the light is reflected and no light enters the second medium. This phenomenon is referred to as *total internal reflection* and the smallest angle θ_1 at which this occurs is called the *critical angle*. Since $\sin\theta_2$ has value 1 there, Snell's law shows that the critical angle is $\sin^{-1}(n_2/n_1)$. Usually one is not interested in this angle and one only wants to know if total reflection has occurred, so that one simply checks if the quantity under the square root sign in the formula for **T** is negative.

9.4.3 Ray Tracing

Dealing with shadows and transparency handles the nonlocal property of illumination in only a very narrow way. There is a lot more interaction between objects. For example, two glossy objects near one another reflect off each other. The models above do not deal with this. Whitted ([Whit80]) is usually credited with implementing the first comprehensive global illumination model. He suggested a recursive ray-tracing algorithm, which can be represented as a tree. See Figure 9.12. One follows each component ray and finds its intersection with all surfaces. Whitted used a formula of the type

$$I = r_a I_a + r_d I_d + r_s I_s + r_t I_t,$$

where the new $r_t I_t$ term models the transparency effects due to refraction. This worked quite well. The next chapter will study ray tracing in detail and so we say no more about it here. Ray tracing is **very** expensive computationally. Whitted used bounding volumes (spheres) to eliminate objects with respect to whether a given ray intersects it. He also incorporated antialiasing.

A lot of work has been done on ray tracing. Its best results can be seen in pictures where there are lots of reflections between objects. The pictures are not totally real

however because these reflections come out too sharp. This does not happen in real life where things are fuzzier. Ray tracing captures the specular interaction between objects very well but not the diffuse interactions. Radiosity methods deal with the latter.

9.4.4 Radiosity Methods

First implemented in 1984 at Cornell University ([GoTG84]), radiosity methods are view-independent solutions that are based on the conservation of light in a closed world. The term *radiosity*, derived from the literature on heat transfer, refers to the rate at which energy leaves a surface and is the sum of the rates at which energy is emitted, reflected, or transmitted from that surface to others. We shall give more details in the next chapter. Suffice it to say that the method is more complex than ray tracing but it produces more realistic pictures even though it does not handle specular light correctly. It is possible to combine ray tracing and the radiosity approach.

9.5 The Rendering Equation

Looking back over what has been covered with regard to illumination in this chapter, we see lots of different formulas and approaches. Kajiya ([Kaji86]) attempted to unify the general illumination problem by expressing it in terms of finding a solution to a single equation that he called *the rendering equation*:

$$I(\mathbf{p},\mathbf{p}') = g(\mathbf{p},\mathbf{p}')\left[\varepsilon(\mathbf{p},\mathbf{p}') + \int_{\text{surfaces}} \rho(\mathbf{p},\mathbf{p}',\mathbf{p}'')I(\mathbf{p}',\mathbf{p}'')d\mathbf{p}''\right], \tag{9.12}$$

where

- $\mathbf{p}$ and $\mathbf{p}'$ are any two surface points,
- $I(\mathbf{p},\mathbf{p}')$ is the intensity of light passing from $\mathbf{p}$ to $\mathbf{p}'$,
- $g(\mathbf{p},\mathbf{p}')$ is a visibility term (which is 0 if $\mathbf{p}$ and $\mathbf{p}'$ cannot see each other and inversely proportional to the square of the distance between the points otherwise),
- $\varepsilon(\mathbf{p},\mathbf{p}')$ is the intensity of light emitted from $\mathbf{p}'$ to $\mathbf{p}$,
- $\rho(\mathbf{p},\mathbf{p}',\mathbf{p}'')$ is related to the intensity of all light reflected towards $\mathbf{p}$ from a point $\mathbf{p}'$ having arrived at $\mathbf{p}'$ from the direction to $\mathbf{p}''$, and
- the integration is over all surfaces in the scene.

Notice that (9.12) is a recursive equation because the function I appears on both sides of the equation. Also, each wavelength has its own equation (9.12). It can be shown ([WatW92]) that most of the illumination models discussed in this chapter are approximations to the rendering equation. The rendering equation does not model everything however. For example, it ignores diffraction and transparency.

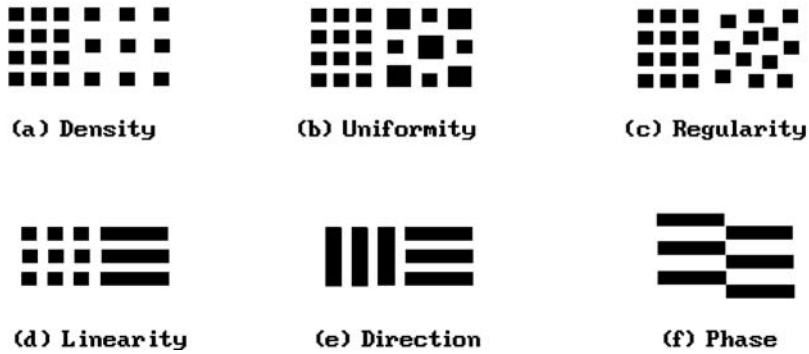


Figure 9.13. Synthetic textures.

9.6 Texture and Texture Mappings

Surfaces are usually **not** homogeneous with respect to any particular property such as color, intensity, etc., but they usually have a more or less uniform pattern that is called the *visual texture*. This pattern could be generated by *physical texture*, such as a rough wall surface, or *markings*, as on wallpaper. Sometimes a collection of objects is viewed as one, as in the case of a brick wall, and then the pattern in each determines the texture of the whole. Texture is a useful concept in understanding scenes. Without texture pictures do not look right.

What exactly is meant by texture? The characteristics of synthetic textures are easiest to explain. Examples of these are shown in Figure 9.13. It is much harder in case of natural phenomena such as sand, straw, wood, etc., but there is some uniformity. One studies texture as a property of a pattern that is uniform in a *statistical* or *structural* sense. There is a nice discussion of texture in [Neva82]. We summarize a few of the main points.

Statistical Texture Measures. Such measures are motivated by a lack of a simple pattern. One looks for average properties that are invariant over a region. An example of this is the probability distribution of single pixel attributes, such as, the mean and variance of its intensity function. Another is the use of histograms of individual pixel attributes. Better yet, one can try to detect the presence of certain features, such as the density of edges, and then compute the mean, variance, etc., of these to distinguish between "course" and "fine" textures. The Fourier transform has also been used to look for peaks since textures are patterns. Using such measures one can generate symbolic descriptions like "bloblike," "homogeneous," "monodirectional," or "bidirectional." An example of a fancier measure is a function of the type

$$p(i,j,d,\theta) = \text{the probability of a pair of pixels separated by a distance } d \text{ in direction } \theta \text{ with intensities } i \text{ and } j$$

Such measures have been used successfully for wood, grass, corn, water, etc.

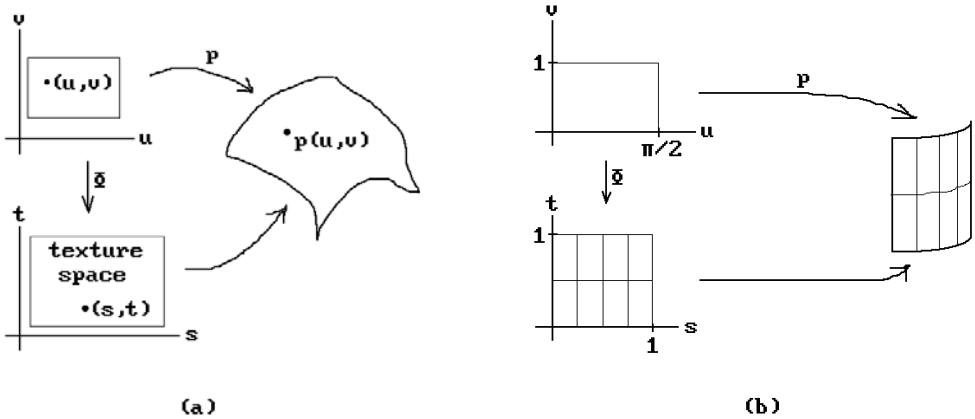


Figure 9.14. Texture mappings.

Structural Texture Measures. The idea here is to find primitives from which the totality is built. This clearly is hard to compute for natural textures. There may be a hierarchy: one pattern may repeat to form a larger pattern. One can think of structural texture in a syntactical way. The primitives are a little bit like sentences of a language with a specified grammar.

Texture is introduced into graphical images by means of *texture mappings*. It is a way to attach detail to surfaces without a geometric model for the detail so that one can produce much more complex images without more complexity in geometric descriptions. This idea was first used by Catmull and then extended by Blinn and Newell ([BliN76]). Heckbert ([Heck86]) presents a good survey of texture maps. See also [WeiD97]. In general, texture maps can be functions of one or more variables. We concentrate on the two variable case here.

Assume that we are given a surface patch parameterized by a function p(u,v). In addition to each point **p** on the surface having (u,v)-coordinates, we now associate *texture coordinates* (s,t) and a predefined texture map T(s,t) defined on this texture coordinate space, which defines the light intensity at the point **p**. If **p** projects to screen coordinates (x,y), then the value T(s,t) is copied to frame buffer location (x,y). Basically, we have a map Φ which sends (u,v) to (s,t). See the commutative diagram in Figure 9.14(a). Usually the map Φ is a linear map and T is represented by a two-dimensional array. Figure 9.14(b) shows how one can map a grid of lines onto a cylinder. The parameterization is assumed to be the map p given by

$$p(u,v) = (\cos(u - \pi/2), \sin(u - \pi/2), v)$$

With domain $[0,\pi/2] \times [0,1]$. The map Φ is given by

$$\Phi(u,v) = (2u/\pi, v).$$

If T is represented by a two-dimensional array, then the intensity T(Round(2u/π),Round(v)) would be associated to the pixel at (x,y). Another way to

deal with repeated patterns like this is to predefine only a primitive part of the pattern and then get the repetition using the **mod** function as in

$$\Phi(u,v) = (2uk/\pi \bmod 1, vk \bmod 1).$$

For example, if k is 10, then we get a 10×10 grid on the patch.

These examples show the essential idea behind texture mappings but assume a perfect mathematical model with all computations carried out to infinite precision. Implementing such an approach involves a lot of work and care must be taken to avoid aliasing. If the rendering algorithm is a scan line algorithm, then for each screen coordinate (x,y) one needs to find the (u,v) so that p(u,v) projects to (x,y), which is time consuming. Catmull ([Catm74]) subdivided the surface until each patch projected into a single pixel. One could then map the center of each of the corresponding rectangles in (u,v) space to texture space. Unfortunately, this straightforward approach leads to aliasing. In Figure 9.14 we might miss the grid lines. Aliasing is a serious problem for texture mappings. One solution is to sample at higher resolutions and the other is to use filters before sampling. Catmull used the latter and a convolution filter. He also subdivided texture space at the same time until each of its patches mapped onto a single pixel and used the average of the texture values in that final texture patch. A better solution is the one found in [BliN76].

Another problem with the above is distortion of the texture if the parameterization is not chosen correctly. For example, if we parameterize the cylinder by

$$p(u,v) = \left(u, -\sqrt{1-u^2}, v\right),$$

then the grid pattern is no longer uniformly spaced on the cylinder. The problem is that the parameterization is not a similarity map. Few are. One simple approach that seems to be quite successful for spline surfaces is to use a chord length approximation to the original parameterization. See [WooA98]. Bier and Sloan ([BieS86]) suggested another approach to alleviate the distortion problem. The idea is to define the texture for an intermediate surface **I** and then use a map μ from that surface to the target surface **O**. Four methods have been used to define the map $\mu : \mathbf{I} \to \mathbf{O}$. Let $\mathbf{q} = \mu(\mathbf{p})$.

Method 1. This method computes μ^{-1}. If **R** is the ray starting at **q** that is the reflection of the ray from the eye to **q**, then **p** is the intersection of **R** with the intermediate surface. See Figure 9.15(a).

Method 2. This method also computes μ^{-1}. If **R** is the ray starting at **q** in the direction of the normal to the target surface at **q**, then **p** is the intersection of **R** with the intermediate surface. See Figure 9.15(b).

Method 3. This is yet another method which computes μ^{-1}. If **R** is the ray from the centroid of the target surface to **q**, then **p** is the intersection of **R** with the intermediate surface. See Figure 9.15(c).

Method 4. If **R** is the ray from **p** in the direction of the normal to the intermediate surface at **p**, then **q** is the intersection of **R** with the target surface. See Figure 9.15(d).

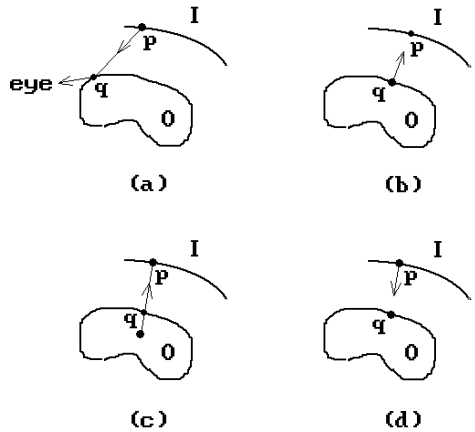


Figure 9.15. Texture mappings with intermediate surfaces.

Some intermediate surfaces that have been used are planes, the surface of boxes, spheres, and cylinders. Using intermediate surfaces that reflect the shape of the target surface rather than always using a square or the sphere is what avoids some of the distortion. Bier and Sloan refer to this approach as "shrink wrapping" a pre-distorted surface onto an object. One could of course eliminate all distortion by letting the intermediate surface be the target surface; however, the latter was presumably too complicated to have defined the texture on it directly. One has to walk a fine line between having relatively simple intermediate surfaces and minimizing distortion. Furthermore, the map μ or μ^{-1} should not be too complicated.

One way to avoid the problems associated with texture maps that we mentioned above is to use three-dimensional texture maps. Specifically, we assign a texture T(x,y,z) to each world point (x,y,z). Then for each point **p** of an object in world coordinates we would simply use the texture T(**p**). In this way textures can be mapped in a nice continuous way onto objects and we have solved one of the main problems of two-dimensional texture maps. Of course, we need to be able to define such a map T(**p**). A table of values would now take up a prohibitive amount of space so that a procedural definition would be called for, but that is not always easy to find.

Aliasing can be problem with texture. The most common solution is to use mipmaps. *Mip-mapping* was developed by Williams ([Will83]) specifically for textures. Instead of a single texture, one precomputes a sequence, each successor being half the resolution of the previous one. One selects the texture for a region of an object based on its distance from the viewer to get the level of detail correct. For a more thorough description the reader can also see [WatW92] or [WatP98].

9.7 Environment Mappings

An *environment mapping* starts with a predefined picture on some closed surface that surrounds the entire world of objects and then maps this picture onto the objects. The difference between this and texture mappings is that the picture is mapped in a view-

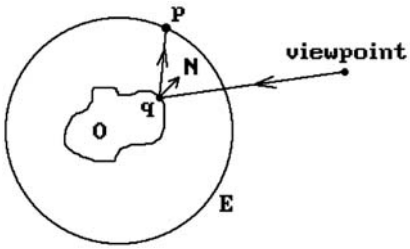


Figure 9.16. A spherical environment mapping.

point dependent way. As an example, consider Figure 9.16. The picture is assumed to be painted on a spherical environment surface **E**. We map it onto the object **O** as follows: To each visible point **q** on **O** we map that point **p** on **E** to which the ray from the viewpoint reflects. Nice effects can be achieved by either moving the object **O** or changing the viewpoint. The environment surface does not have to be a sphere. In fact, it turns out that a sphere is not a good choice because trying to paint a picture on it can easily cause distortion. A more common choice is to use a cube. One could, for example, take six pictures of a room and map these to the six sides of the cube.

Environment mappings were originally developed in [BliN76] where they were called *reflection mappings*. [Gree86] suggested using cubes. The whole idea of environment mappings is basically a cheap way to get the kind of reflection effects that one gets with ray tracing, but they have become popular. Large flat surfaces on objects cause problems however because the reflection angle changes too slowly.

9.8 Bump Mappings

A problem related to giving texture to objects is to make them look rough or smooth. Simply painting a "rough" texture on a surface would not work. It would not make the surface look rough but only look like roughness painted on a smooth surface. The reason for this is that the predefined texture image is using a light source direction that does not match the one in the actual scene. One needs to change the normals (from which shading is defined if one uses the Phong model) if one wants an effect on the shading. This was done by Blinn ([Blin78]), who coined the term "bump mapping." Again, assume that we have a surface patch **X** parameterized by a function p(u,v). A normal vector n(u,v) at a point on the surface is obtained by taking the cross-product of the partial derivatives of p(u,v) with respect to u and v, that is,

$$n(u,v) = p_u(u,v) \times p_v(u,v).$$

If we perturb the surface slightly along its normals, we get a new surface **Y** with parameterization function P(u,v) of the form

$$P(u,v) = p(u,v) + b(u,v)\frac{n(u,v)}{|n(u,v)|}, \tag{9.13}$$

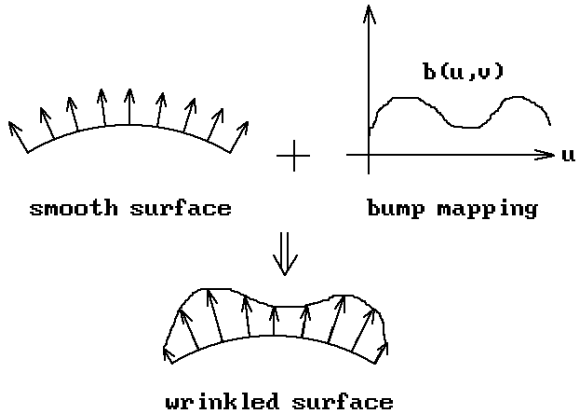


Figure 9.17. Texturing with bump maps.

where b(u,v) is the *bump map* or perturbation. See Figure 9.17. The vectors

$$N(u,v) = P_u(u,v) \times P_v(u,v)$$

are normal vectors to **Y** at P(u,v). But, suppressing references to the parameters u and v,

$$P_u = p_u + b_u \frac{n}{|n|} + b \frac{\partial}{\partial u}\left(\frac{n}{|n|}\right)$$

and

$$P_v = p_v + b_v \frac{n}{|n|} + b \frac{\partial}{\partial v}\left(\frac{n}{|n|}\right).$$

If we assume a small perturbation b(u,v), then it is reasonable to neglect the last terms. Therefore, N is approximated by

$$\begin{aligned} N' &= p_u \times p_v + b_u \frac{n \times p_v}{|n|} + b_v \frac{n \times p_u}{|n|} \\ &= n + b_u \frac{n \times p_v}{|n|} + b_v \frac{n \times p_u}{|n|}. \end{aligned} \tag{9.14}$$

Note that in order to compute the approximate normals for **Y** we do not need to know the perturbation function b(u,v) itself, but only its partial derivatives. Any function can be used as a bump function. To speed up the computation one typically uses a lookup table and interpolation. Standard approximations to the partials are

$$b_u(u,v) = \frac{1}{2\varepsilon}(b(u+\varepsilon,v) - b(u-\varepsilon,v)) \tag{9.15a}$$

$$b_v(u,v) = \frac{1}{2\varepsilon}(b(u,v+\varepsilon) - b(u,v-\varepsilon)), \tag{9.15b}$$

for suitable small value ε. Thus it suffices to use a table b(i,j) and to compute b(u,v) at all other values via a simple linear interpolation. The values of the partials b_u and b_v are computed with formulas (9.15).

To reduce aliasing, Blinn suggested that one sample intensities at a higher resolution and then average the values.

9.9 The Rendering Pipeline

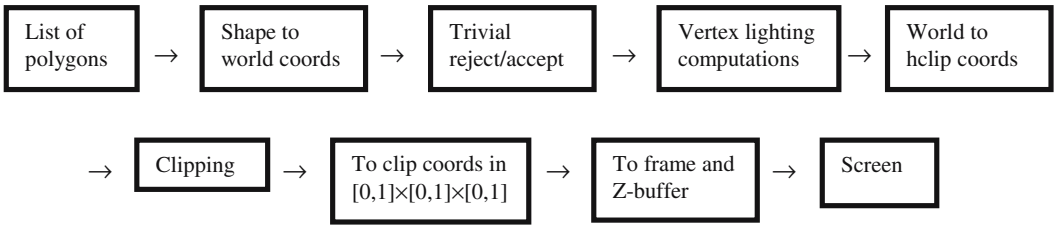
In this and previous chapters we have covered many different topics that deal with getting a real picture onto a computer's monitor screen. We started be describing the basic graphics (projection) pipeline and then moved on to visible surface algorithms and illumination and shading models. Everything might have made sense in isolation, but the reader may have been left wondering how one puts all this together in practice. What is the order in which various steps should be performed? This section will try to clarify that. The answer depends on how objects are represented, whether we are using a local or global illumination model, and which visible surface algorithm we are using.

The most popular way to render scenes has become a case of first polygonizing the objects in the scene (if they are not already a collection of polygons) and then rendering the resulting polygons. With this approach, a polygon becomes the basic unit in the rendering process and the typical graphics hardware supports this nowadays. Such support is important for real-time rendering because a complex scene with curved objects might need millions of polygons for a good representation. Although rendering smooth surfaces by means of approximating polygons is the most efficient approach with today's hardware, there are problems, such as aliasing, and algorithms exist for rendering them directly. See, for example, [ElbC96].

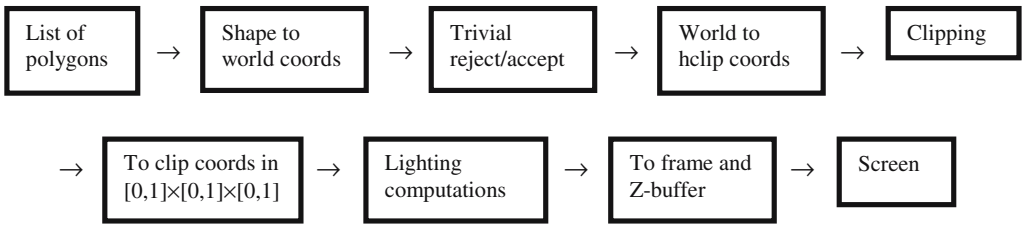
Local Illumination Pipelines. There are two basic ways to render the list of polygons. One can render each polygon individually and independent of each other (*by-polygon rendering*) or one can render all the polygons in a scan-line order (*by-scan-line rendering*), meaning that all the segments in which polygons meet a scan line are rendered before one moves on to the next scan line. One can use different visible surface algorithms, but the use of a Z-buffer or some sort of depth ordering has become commonplace.

Rendering approach	Compatible visible surface algorithm type
by-polygon	Z-buffer, list priority type algorithms
by-scan-line	Z-buffer, scan line Z-buffer, scan line algorithm

The advantage of by-polygon rendering is that it is easy to implement with little active data to maintain, whereas the by-scan-line rendering has to maintain a potentially large amount of data for each scan line. On the other hand, by-scan-line rendering is



(a) Z-buffer algorithm and Gouraud shading



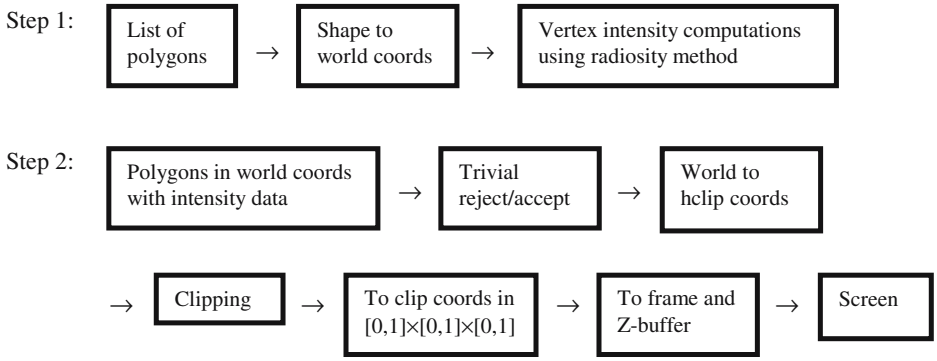
(b) Z-buffer algorithm and Phong shading

Figure 9.18. Local illumination rendering pipelines.

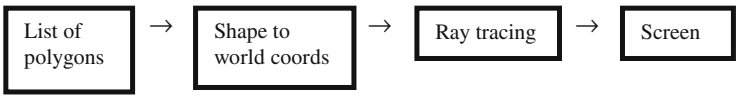
more efficient since it can take advantage of coherence. Furthermore, since the image is generated in scan line order, there are opportunities for hardware optimization along with making anti-aliasing easier.

As an example, Figure 9.18(a) shows the rendering pipeline for a Z-buffer algorithm and Gouraud shading. A polygon is first transformed into world coordinates. Simple tests, such as back face elimination or bounding box tests, are performed to eliminate polygons that are obviously not visible. The illumination is computed at each vertex. This computation cannot be postponed because the perspective transformation distorts the needed normal and light vectors. (Camera coordinates would also work.) Next the object is mapped to the homogeneous clipping coordinates and clipped. If new vertices are introduced at this stage, then we must compute illumination values for them. To get these new values we have to map the points back to the corresponding world coordinate points and do the illumination computations for the latter. Finally, the vertices of the clipped polygon are mapped back down to the normalized clip coordinates in the unit cube and sent to the Z-buffer along with the illumination information. The Gouraud algorithm will now interpolate the vertex values over the entire object.

A second example of the rendering pipeline where we use the Z-buffer algorithm and Phong shading is shown in Figure 9.18(b). The difference between this pipeline and the one in Figure 9.18(a) is that, since we need normals, we cannot do any lighting computations until we are finished clipping. It is at that point that we need to map all vertices back to world or camera coordinates. The normals, their interpolated



(a) Radiosity and Gouraud shading



(b) Ray tracing

Figure 9.19. Global illumination rendering pipelines.

values, the light direction vectors, etc., on which illumination computations are based, must all be in world or camera coordinates.

As a third example of the rendering pipeline, suppose that one were to use a list priority algorithm such as the BSP algorithm and Phong shading. The pipeline will be similar to the one shown in Figure 9.18(b), but we do not need a Z-buffer here because polygons are ordered by depth and we can use a painter's algorithm.

Global Illumination Pipelines. In the case of radiosity and, say, Gouraud shading we first convert objects to world coordinates and precompute the view-independent vertex intensities. This new data is then fed into a standard rendering pipeline, which no longer needs to do any lighting computations. See Figure 9.19(a).

The ray-tracing rendering pipeline is the easiest of them all. It is shown in Figure 9.19(b). We simply convert all objects into world coordinates and then do the ray tracing for each pixel on the screen.

9.10 Selecting a Color Palette

After the pixel data has been generated for an image, one is sometimes faced with one last decision, namely, choosing a *palette* (an indexed collection of colors). The problem is that there may be a limit to how many colors a graphics system can display at any one time and the image may contain more colors than that. This is no longer such a

big issue today because graphics systems typically have no problem displaying 24 bits of color; however, in earlier days it was only possible to display 256 colors (or less). We sketch some solutions to the color quantization problem here. More details can be found in [Lind92]. See also [Paet90].

Fixed Palette Quantization. Here one basically ignores the problem and simply uses a fixed palette of, say, 256 colors for **all** pictures and "rounds" all image colors to one of these. This is obviously the simplest approach but would probably not be very satisfying.

Uniform Quantization. Here we divide the RGB color cube into uniformly spaced subcubes and round any number to its appropriate subcube representative.

Popularity Quantization ([Heck82])**.** In this approach we count the number of pixels associated to each color in the image and then use the colors with the largest counts. All other colors are mapped to one of these using some sort of minimum distance criterion. The algorithm works fairly well if there are only a small number of colors in the image. It has problem with highlights that usually account for only a few pixels. One way that this problem is diminished is by always including the eight corners of the RGB cube.

Median Cut Quantization ([Heck82])**.** This improves on the popularity quantization and is probably the most popular method. The idea here is to choose colors so that each represents roughly the same number of pixels in the image. The set of colors in the RGB color cube from the image is enclosed in a box and the longest side of the box is divided in half at the median point so that half of the image colors are in each half. This division process is repeated recursively on the smaller boxes until we have as many boxes as the desired number of colors in our palette. All image colors are then rounded to the colors at the centroid of the boxes.

Octree Quantization ([GerP90])**.** This approach mimics the octree approach to representing solids. We start with the color cube and divide it into eight subcubes. These subcubes are then further subdivided recursively depending on whether or not they contain a color from the image. The tree is maintained at the desired size with an averaging process. Any reductions in size are carried out as the tree is being built. There is no need to use up a lot of space and build the complete tree first. The main advantages of octree quantization are that it is faster, uses less memory, and produces similar results to the other methods.

9.11 Programming Notes

This section provides the reader with a few helpful hints for using shading in a modeling program. Most are in reference to the Bouknight or Phong reflectance models.

First of all, use ambient light because, without ambient light, back faces would be black. Set the ambient light to the color of the background, even though this is not what we actually experience because in real life light reaches that face from diffuse

surfaces in many ways. In any case, the ambient term lets us model this light in a simple way without doing lots of complicated computations (such as radiosity). Too little ambient light creates very sharp shadows, too much washes out the entire picture.

Picking good reflectance coefficients k_a, k_d, and k_s is not easy and they are usually found by experimentation. The task is basically an art because it relies on one's experience. This should not be that surprising given the empirical foundation of the Bouknight and Phong models. Simple models set the entire ambient term to a constant. Other constants to try are

$$k_a = 0.3, k_d = 0.7, k_s = 0.9, m = 100, \text{and } k_t = 0.65.$$

For several light sources one uses separate diffuse and specular light components. There is one problem that arises whether one is using a single or several light sources. One wants the total computed light intensities at a pixel to fall inside the interval [0,1] because they are considered to be based on the color cube model. This may not happen though. The simplest solution would be to clip the value to [0,1]. One can also scale all pixel values appropriately, although this would involve more work. Unfortunately, either of these solutions may cause color artifacts. For a good analysis of this problem see [Hall89]. In general, Hall's book is a good all-around reference for practical suggestions dealing with illumination and regarding the use of the Phong model. Note that once intensities are in [0,1] a program would scale them to the correct bit values. For example, if our graphics system supports 8 bits each for red, green, and blue, then we would scale to an integer value between 0 and 255. The typical graphics API requires that colors are represented by integer values (or composites thereof, since one combines the red, green, and blue parts into one word usually).

Here are some sample RGB color values ([Lind92]), but be aware that these are not universally agreed on values:

aquamarine	= (0.498,1.0,0.8314)	ivory	= (1.0,1.0,0.9412)
beige	= (0.64,0.58,0.5)	maroon	= (0.6902,0.1882,0.3765)
brown	= (0.5,0.1647,0.1647)	mint	= (0.74,0.99,0.79)
carrot	= (0.93,0.57,0.13)	orange	= (1.0,0.5,0.0)
chartreuse	= (0.498,1.0,0.0)	orchid	= (0.8549,0.4392,0.8392)
chocolate	= (0.8235,0.4118,0.1176)	pink	= (1.0,0.7529,0.7961)
cobalt	= (0.24,0.35,0.67)	plum	= (0.8667,0.6275,0.8667)
copper	= (0.84,0.04,0.15)	purple	= (0.6275,0.1255,0.9412)
coral	= (1.0,0.498,0.3137)	silver	= (0.8,0.8,0.8)
flesh	= (1.0,0.49,0.25)	turquoise	= (0.251,0.8784,0.8157)
gold	= (1.0,0.8431,0.0)	violet	= (0.56,0.37,0.6)
indigo	= (0.03,0.18,0.33)	wheat	= (0.9608,0.8706,0.7020)

On another matter, to make a light appear in a scene associate to it a sphere which is fully transparent.

Finally, there is a fact about luminance (measured brightness) that should be mentioned. RGB color values and the luminance that they define are not related in a linear way on most CRT monitors or our visual system. If one were to switch from an intensity value of 0.5 to one of 1.0, one might expect that the image would be twice as

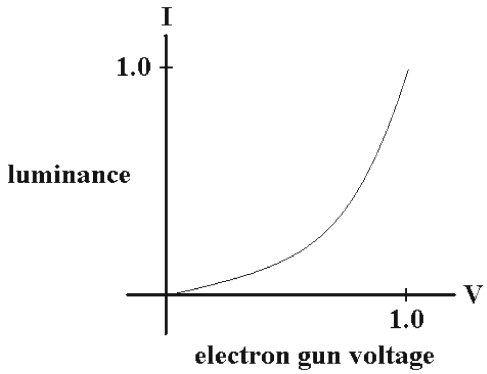


Figure 9.20. The relationship between CRT gun voltage and luminance.

bright, but that is not what happens. Figure 9.20 shows the relationship between electron gun voltage V and luminance I. The formula relating the two has the form

$$I = c\,V^{\gamma},$$

where c and γ are constants that are different for each monitor. The typical range for γ is a value between 2.0 and 3.0 with $\gamma = 2.2$ being the NTSC signal standard. The compensation for this nonlinearity is called the *gamma correction* and is accomplished by using a lookup table. For example, if intensities are translated into 8-bit integer values, then one would have a table of size 256 and location k in the table would hold the value $k^{1.0/\gamma}$. The same lookup table would be used for the red, green, and blue components of a color.

9.12 Programming Projects

Section 9.3

9.3.1 Extension of programming project 7.5.1.

Now include shading. Compare the results you get from using the Bouknight and Phong reflectance model.

Section 9.4

9.4.1 A simple shaded world.

(a) Display a shaded world of spheres and a "floor." Also show shadows. Represent the floor as a large rectangle in the plane with the spheres above it. Use a simple ray-tracing program. Send out two rays: one from the camera (through each pixel in the view plane) to find the point on the nearest object and then one from that point to the light to see if it is in the shadow. For testing purposes, add the following submenu to your main menu

SPHERES
ViewPos LightPos

Clicking on the ViewPos and LightPos items should let the user to change the position of the viewer and the light, respectively.

Again compare the results from using the Bouknight and Phong reflectance model. See how adding a distance factor changes the image.

(b) Add blocks to the world in (a).

Section 9.6

9.6.1 Implement some textures on boxes and spheres.

CHAPTER 10

Rendering Techniques

10.1 Introduction

As computers get faster and faster the goal of producing very realistic images in real time is becoming attainable even on a PC. This chapter describes ray tracing and the radiosity method, methods that generate the most impressive images. We also briefly discuss volume rendering.

We already touched on ray tracing in Chapters 7 and 9. What we called an "image precision algorithm" in Chapter 7 is basically a very simple generic ray-tracing algorithm. A more complete ray-tracing algorithm is the Whitted ([Whit80]) algorithm mentioned in Chapter 9. In this chapter, we describe ray tracing in more detail. The basic idea is really very simple. The problem is that to carry it out involves finding a large number of intersections of rays with objects making it very slow. In the early days of ray tracing it was common to have individual ray traced pictures take many hours of computer time. We start off in Section 10.2 with ray tracing in a continuous world. Ray tracing in the context of volume rendering is considered in Section 10.4. Section 10.2.1 presents code for a basic ray tracer. Section 10.2.2 presents the mathematical solution to some more ray-object intersection problems. The special problem of ray tracing CSG objects is discussed in Section 10.2.3. For more about implicit objects see [Bloo97].

Section 10.3 is on the radiosity method. Because this approach is much more involved we only give an overview of the main points and refer the reader to other more comprehensive treatments of the subject. Section 10.4 gives a brief overview of some topics in volume rendering.

Finally, although computer graphics has progressed to a stage where the pictures it produces are really impressive, Greenberg ([Gree99]) questions their "realism." He raises the point that, by in large, the pictures are not yet accurate from a visual point of view (as opposed to geometric accuracy, which is being handled adequately). They may look good but do not represent a real physical scene most of the time. We still have a ways to go if we want to use graphics to simulate the world and make predictions with regard to visual properties (in the same way that we can accurately predict geometric properties). Although we have accurate light reflectance and transport (how the light propagates through the environment) models, one is often forced

to make simplifications because of the computational complexity. A sometimes-overlooked important third stage (and one we do not consider at all in this book) is mapping these values correctly to the display device to produce the correct image for the user. The goal is to make a user's perceptual reaction to a real scene and to its picture the same. These comments lead us to say a few words about another topic, image-based rendering.

Image-based rendering has become a very active area of research. In very broad terms, what one is trying to do here is to reconstruct a three-dimensional world from two-dimensional data that may have been collected from a variety of sensors, such as cameras. In the case of pictures from a camera, one is starting with something that is as realistic as it can get since it involves an image of an actual scene, not an artificially created one. In one form of image-based rendering one takes one or more pictures of a scene and then attempts to show how the scene would look from different view points. No three-dimensional models are built and no lighting model is involved because one is simply transforming given pixel values. In other types of image rendering the goal is to reconstruct a three-dimensional world that can then be displayed from different views. Depending on the generality of the scenes that one wants to handle, this might involve using tools from the field of computer vision. *Computer vision* deals with taking some raw two-dimensional pixel data and trying to make sense of it by, for example, finding the edges in the picture and then collecting them together into higher-level objects such as curves and rectangles. Knowing the rectangles in a picture one can then try to figure out the three-dimensional object that projected on them.

By in large, image-based rendering is only peripherally related to the subject of this book. A good place for a vast amount of information about the field are some of the courses at the yearly ACM SIGGRAPH conference that have been dedicated to this subject, for example, [Debe99]. As one can see from the title of that particular course, the field can be considered as divided into three parts: image-based modeling, image-based rendering, and image-based lighting. In *image-based modeling* all or part of the goal is to reconstruct a three-dimensional world. Although one uses methods of geometric modeling, this is not really a part of the field of geometric modeling per se. Image-based rendering, as we described above, deals with generating different views of a picture by transformations of the original. *Image-based lighting* refers to the process of reconstructing the correct lighting for the different views generated via image-based modeling or rendering.

10.2 Ray Tracing

10.2.1 A Ray-Tracing Program

A top-level abstract ray-tracing program looks like the following:

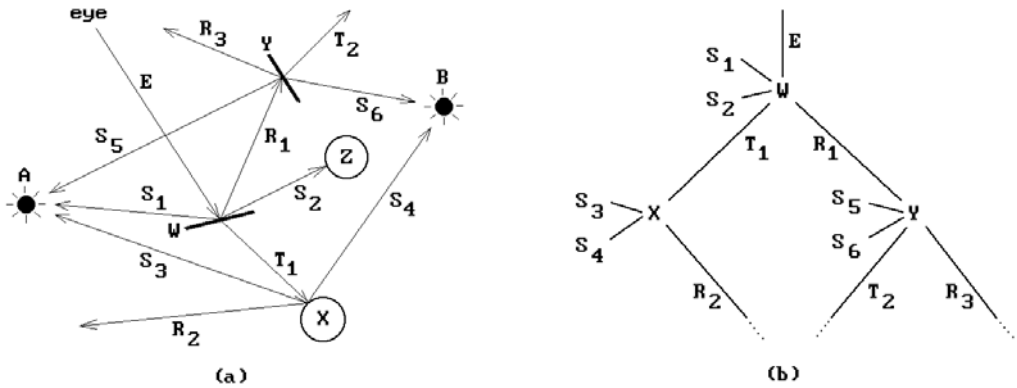


Figure 10.1. Ray-tracing rays.

```
for all pixels p do
    begin
        Define the ray R(p) which starts at the eye and goes in the
           direction determined by p in world coordinates;
        Find the intersections of R(p) with all objects in the world;
        Determine which intersection is closest to the eye;
        Determine the color of the light returning to the eye from that
           point;
        Set the value of p to that color;
    end;
```

To determine the color at the pixel, one needs to determine all possible light rays that arrive at the corresponding point in the world. This reduces to recursively sending out a number of secondary rays.

The ray-tracing program described in this section will keep track of four different rays. See Figure 10.1(a). A *pixel ray* is the ray "from" the eye to the point in the view plane determined by a pixel. A *reflection ray* is a ray that carries reflected light to a point on an object. A *transparency ray* is a ray that carries light (the *transmitted light*) to a point on an object from "within" it. Finally, a *shadow ray* is the ray from a point on an object to a light source. Shadow rays are used to determine if a point is in the shade of another object. As mentioned in the last chapter, shadow information is important for making pictures look realistic. In Figure 10.1(a) we assume that objects **W** and **Y** are transparent whereas objects **X** and **Z** are not, that is, the latter two objects do not allow any light to pass through them. In the figure, E is a pixel ray. R_1 is the reflection of E at a point on object **W** and T_1 is the transparency ray. S_1 and S_2 are the shadow rays to the two light sources A and B. The fact that S_2 intersects **Z** means that **Z** casts a shadow on **W**. The ray T_1 hits object **X** and is reflected along R_2. Two shadow rays S_3 and S_4 are spawned. The ray R_1 hits object **Y** and generates a reflected ray R_3 and a transparency ray T_2. The two shadow rays here are S_5 and S_6. The ray tree is shown in Figure 10.1(b).

Assume that the viewport is the rectangle [XMIN,XMAX] × [YMIN,YMAX] and that I[i,j] specifies the intensity value of the (i-j)th pixel in the frame buffer. Algorithm

```
integer          i, j;
ray              r;                  { defines starting point and direction of ray }
point            eyePt, hitPt;
shape pointer    hitObjP;
real array       I [XMIN .. XMAX, YMIN .. YMAX];

for  i:=YMIN to YMAX  do
     for  j:=XMIN to XMAX  do
          begin
               ComputeRay (i,j,r);
               FirstIntersection1 (r,hitPt,hitN,hitObjP);
               if  hitObjP ≠ nil
                    then  I[i,j]  :=  Shade (eyePt,nil,hitPt,hitN,hitObjP,1)
                    else   I[i,j]  :=  background
          end;

procedure ComputeRay (integer i, j;  ref ray  r)
{ Computes the ray r (in world coordinates) from the eye defined by the (i,j)th pixel }

procedure FirstIntersection1 (ray r; ref point hitPt, hitN; ref shape pointer hitObjP)
{ Returns a pointer hitObjP to the closest object intersected by the ray r, the point hitPt
  on that object where the ray intersects it, and the surface normal hitN at that point.
  Use bounding boxes for projected objects in view plane to speed up computation. }
```

Algorithm 10.2.1. A ray-tracing program.

10.2.1 shows a more detailed ray-tracing program. The Shade function is shown in Algorithm 10.2.2. There we further assume that each object has associated with it two fields – **shininess** and **transparency** – which indicate if it is shiny or transparent enough to warrant sending out new reflection and transparency rays, respectively. See [Hill90]. For simplicity we treat intensity as just a single real number in both algorithms even though it has three components, for red, green, and blue. A real algorithm would have to make three separate computations. For a more sophisticated ray-tracing program using OpenGL, see [Hill01].

To speed up a ray-tracing program one needs to reduce the number of ray-object intersection computations. One thing that is usually done is to use bounding boxes or other bounding objects. More generally, one uses a hierarchical structure for the bounding objects. For example, in the case of a table one might have a bounding box for the whole table, and, if a ray intersected this box, then one would next check for intersections with bounding boxes for the legs and top. Kay and Kajiya ([KayK86]) discuss the properties such a hierarchy should have and how one can construct it. A natural hierarchy of bounding objects can be gotten by mimicking the hierarchy or tree structure of the parts of an object.

```
{ Constants }
maxDepth = 10;              { the maximum number of reflections or refractions we allow }
minShinyness     = 0.1;
minTransparency = 0.1;

real refl, refr;            { Global reflection and refraction coefficients }

real function  Shade  ( point atPt;  shape pointer atObjP; point fromPt, fromN;
                        shape pointer fromObjP;  integer depth )
{ This returns the shade at point atPt on the object to which atObjP points determined by the
  light coming from fromPt on object to which fromObjP points. fromN is normal at fromPt. }
begin
    boolean     rayInObject;
    real        light;        { the intensity }
    point       viewV, lightV, lightPt, newPt, newN;
    ray         reflectedRay, refractedRay;
    shape pointer  newObjP;

    if  fromObjP = nil  then  return (background intensity);

    viewV         := atPt – fromPt;
    rayInObject  :=  (atObjP = fromObjP);

    { First we compute the local intensity }
    if  rayInObject
        then  light  :=  0.0
        else
            begin
                light  :=  AmbientLight (atPt);
                for all   light source location lightPt   do
                  begin
                    lightV := lightPt – fromPt;
                    if  not (InShadow (fromPt,lightPt)  then
                        light  :=  light  +  DiffuseLight(viewV,lightV,fromN)
                                          +  SpecularLight(viewV,lightV,fromN);
                  end
            end;

    if  depth < maxDepth  then
        begin
            { Now shoot the reflected ray }
            if  ShinynessOf (fromObjP) > minShinyness  then
                begin
                    reflectedRay := MkReflectedRay (viewV,fromPt,fromN);
                    FirstIntersection2 (reflectedRay,newPt,newN,newObjP);
                    if  newObjP ≠ nil  then
                      light  :=  light  +  refl*Shade (fromPt,fromObj,newPt,newN,newObjP);
                end;
```

Algorithm 10.2.2. A shade function.

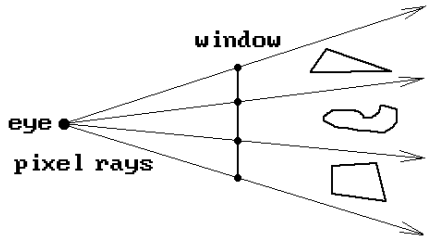
```
            { Next, shoot the refracted ray }
            if TransparencyOf (fromObjP) > minTransparency then
              begin
                if not(TotalInternalReflection (fromN,viewV)) then
                  begin
                    refractedRay := MkRefractedRay (viewV,fromPt,fromN,fromObj);
                    FirstIntersection2 (refractedRay,newPt,newN,newObjP);
                    if newObjP ≠ nil then
                      light := light + refr*Shade (fromPt,fromObj,newPt,newN,newObjP);
                  end
              end
        end;

    return (light);
end; { Shade }

{ The functions MkReflectedRay and MkRefractedRay return the appropriate ray }

procedure FirstIntersection2 (ray r; ref pnt3d hitPt, hitN; ref shape pointer hitObjP)
{ The only difference between this procedure and FirstIntersection1 is that we use
   three-dimensional bounding boxes rather than two-dimensional boxes }
```

Algorithm 10.2.2. *Continued*

Bounding boxes help in determining the intersection of a ray with an object, but perhaps there was no need to consider that object in the first place. In other words, we should try to mark those parts of space that contain no objects at all and take advantage of **spatial coherence**. For example, using a *median cut scheme* that is generated automatically by the program we can create a top-down binary tree obtained by sorting objects by the x-coordinate of some different coordinate axes at each level and using the median value to partition the objects. If one were to sort on **all** three coordinates x, y, and z **simultaneously**, one would get a hierarchy similar to that of an octree. See [Glas84] and [FuTI86]. Alternatively, Sung and Shirley ([SunS92]) describe how a BSP tree can be used and compare this method to others. Subsequently, Havran et al. ([HKBZ97]) showed that one can speed up the traversal substantially if one uses an orthogonal BSP tree and ensures that statistically more common cases are handled more efficiently. Another way to use coherence is to alternate scanning from left-to-right and right-to-left. In that way one can use coherency at the ends of scan lines to make intersection computations more efficient. For a more detailed discussion of ray tracing speed-ups see [WatW92].

The simple ray-tracing program described in Algorithm 10.2.1 runs into the old problem of aliasing, that is, the problem of not sampling enough. For example, we only sent out as many primary rays as there are pixels. This means that some small objects may not show up because they "fell through the crack." See Figure 10.2. Two

Figure 10.2. Losing small objects in ray tracing.



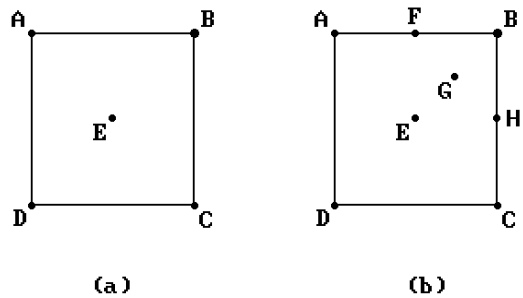


Figure 10.3. Shooting rays with adaptive supersampling.

ways to deal with this aliasing problem are (uniform) supersampling and stochastic sampling. More thorough discussions can be found in [WatW92] and [Roge98].

Supersampling. With this approach one simply sends out more rays and then averages the results. For example, one could send out k^2 rays for each pixel. More specifically, if the resolution is $m \times n$, assume a virtual resolution of $km \times kn$ and send out a ray for each virtual pixel and then average the values of successive $k \times k$ grids. Although this approach does not solve the aliasing problem, it helps. According to [Whit85], $k = 4$ seems to give adequate results. Two methods related to supersampling that avoid the problem in simple ray tracing caused by sending out an infinitesimally thin ray are *cone tracing* ([Aman84]), where single rays are replaced by thin cones, and *beam tracing* ([HecH84]), where rays are replaced by bundles of parallel rays.

Adaptive Supersampling ([Whit80])**.** This is a variant of supersampling. The idea here is that rather than blindly shooting off all the rays that supersampling requires, one should concentrate where it counts. See Figure 10.3. One starts by tracing rays through the four corners and center of a pixel, marked A–E in the figure. Next, one compares the colors of the pairs of rays (A,E), (B,E), (C,E), and (D,E). One starts subdividing and sending out more rays only where differences are detected. For example, suppose that the values at the pairs (A,E) and (D,E) are similar, but the values at the other pairs are substantially different. In this case, one should look at the quadrant defined by E and B more closely. To do this, one sends out new rays through F, G, and H shown in the figure. Again one compares the values in this quadrant and, if there are any differences, then the subdivision process is continued until no substantial differences between adjacent points are detected. Next, one would notice that the values at C and E are different and apply the same recursive process to the E-C quadrant.

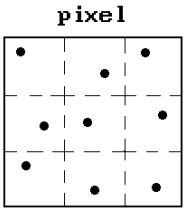


Figure 10.4. Shooting nine rays for distributed ray tracing.

In the end we have values for a collection of rays and the last step is to determine an appropriate weighted average of all these ray values, which is then assigned to the pixel. A simple scheme for doing this is to recursively average the four subquadrant values for every quadrant.

Stochastic Sampling. Here, rather than sending out extra rays in a uniform pattern as is done in supersampling, we sample in a nonuniform manner. One pattern, suggested by [Cook86], is referred to as a minimum-distance Poisson distribution, where one rejects points from a Poisson pattern that are closer than some minimum distance. This method is very expensive because it requires a very large lookup table. Another method, which turns out to be an approximation to the one just mentioned, is called *jittering*, where we start with a uniformly spaced pattern of points and then displace them by a small amount. This is much easier to implement. From a theoretical point of view, what we are doing is replacing aliasing with noise, something that the human eye finds less bothersome.

Distributed Ray Tracing ([CoPL84], [Cook89])**.** This is a special case of the jittering approach. The area of a pixel is uniformly subdivided and a ray sent out through a randomly chosen point in each subdivided area. Cook found that 16 rays per pixel was adequate. See Figure 10.4 for the case of nine rays. The method was not used just for visibility computations, but also for other properties, providing a uniform approach to achieving

(1) blurry reflections,
(2) blurry transparency,
(3) soft shadows,
(4) depth of field, and
(5) motion blur.

Only the initial ray was sampled stochastically. All subsequent reflected rays used a precomputed Monte Carlo method via lookup tables.

Other stochastic sampling methods were developed to cut down on unnecessary rays by establishing criteria for "important" areas.

10.2.2 Ray Intersection Formulas

In this section we shall derive formulas for the intersection of a ray in $\mathbf{R}^n$ with various standard surfaces. Before we begin we should again point out the obvious. No matter

how efficient ray intersection tests are it is better not to have to perform them at all. One should do whatever is possible to avoid the tests altogether. To that end we mention one useful step ([Ritt90]). If an object has a bounding box, then find a good bounding rectangle in the screen of the projection of the corners of this box and check if the pixel that defines the current ray lies in this rectangle. These computations will typically be simpler than any ray intersection test for the three-dimensional object.

We shall use the following notation throughout this section: **X** will denote a ray from a point **p** in a direction **v** and **L** will denote the **line** through **p** with direction vector **v**.

10.2.2.1 Problem. To find the intersection **q**, if any, of the ray **X** and the (n − 1)-dimensional sphere **S** with center **a** and radius r.

Solution. This problem is solved in the same as that for the ray–circle intersection problem already handled in Section 6.5.

The next problems assume that we are dealing with rays in $\mathbf{R}^3$.

10.2.2.2 Problem. To find the intersection **q**, if any, of the ray **X** and the cylinder **Y** with radius r and height h centered at the origin with axis the z-axis.

Solution. First of all, note that the cylinder **Y** is defined by

$$\mathbf{Y} = \{(x,y,z) \mid x^2 + y^2 = r^2 \text{ and } 0 \le z \le h\}.$$

Let $\mathbf{p} = (p_1,p_2,p_3)$, $\mathbf{v} = (v_1,v_2,v_3)$, and $\mathbf{q} = \mathbf{p} + t\mathbf{v}$. Then we must have

$$(p_1 + t\,v_1)^2 + (p_2 + t\,v_2)^2 = r^2,$$

which reduces to

$$At^2 + 2Bt + C = 0, \tag{10.1}$$

where

$$A = v_1^2 + v_2^2, \quad B = p_1v_1 + p_2v_2, \quad \text{and} \quad C = p_1^2 + p_2^2 - r^2.$$

Case A = 0: In this case the ray is parallel to the axis of the cylinder. One must check for the special case where the ray lies in the cylinder, that is, $p_1^2 + p_2^2 = r^2$.

Case A ≠ 0: In this case one has to analyze the solutions to the quadratic equation (10.1) as we did in the case of spheres. If a solution is found, then one must check that $0 \le p_3 + t\,v_3 \le h$. Only then will the ray actually intersect the cylinder.

If the cylinder has a top and/or bottom and the ray did not intersect the sides of the cylinder, then we need to find the intersection of the ray with the planes z = 0 and z = h. Given such an intersection $\mathbf{q} = (q_1,q_2,q_3)$, the ray intersects the bottom or top if $q_1^2 + q_2^2 \le r^2$.

10.2.2.3 Problem. To find the intersection **q**, if any, of the ray **X** and the cone **Y** with radius r and height h centered at the origin with axis the z-axis.

Solution. The cone **Y** is defined by

$$\mathbf{Y} = \left\{(x,y,z) \middle| x^2 + y^2 = (r/h)^2(h-z)^2 \text{ and } 0 \le z \le h\right\}.$$

Let $\mathbf{p} = (p_1,p_2,p_3)$, $\mathbf{v} = (v_1,v_2,v_3)$, and $\mathbf{q} = \mathbf{p} + t\mathbf{v}$. Then we must have

$$(p_1 + t\,v_1)^2 + (p_2 + t\,v_2)^2 = (r/h)^2(h - p_3 - t\,v_3)^2,$$

which reduces to

$$At^2 + 2Bt + C = 0, \tag{10.2}$$

where

$$A = v_1^2 + v_2^2 - (r/h)^2 v_3^2,$$

$$B = p_1v_1 + p_2v_2 + (r/h)^2(h - p_3)v_3, \quad \text{and}$$

$$C = p_1^2 + p_2^2 - (r/h)^2(h - p_3)^2.$$

Case $A = 0$: In this case the direction vector of the ray lies in the cone, that is, it is parallel to a generating line for the cone.

- $B = 0$: The ray intersects the cone if and only if **p** lies on **Y** and satisfies its equation.
- $B \neq 0$: In this case use equation (10.2) to solve for t and check whether $0 \le p_3 + t\,v_3 \le h$.

Case $A \neq 0$: In this case one has to analyze the solutions to the quadratic equation (10.2) similar to what we did in the case of spheres. If a solution is found, then one must check that $0 \le p_3 + t\,v_3 \le h$. Only then will the ray actually intersect the cone.

If the ray did not intersect the sides of the cone and the cone has a bottom, then one must still check if the ray intersects the bottom. This is done in the same way that one checks whether a ray intersects the bottom of a cylinder.

A generalization of Problems 10.2.2.2 and 10.2.2.3 is finding the intersection of a ray with an arbitrarily positioned cylinder or cone. The best way to deal with this problem is to transform the problem to a normalized coordinate system to which the above solutions apply.

10.2.2.4 Problem. To find the intersection, if any, of the ray **X** and a polygon.

Solution. One first checks whether the ray is parallel to the plane containing the polygon. If it is, then there will be no intersection if **p** does not belong to the plane. If **p** does belong to the plane, then the problem reduces to finding the intersection of a ray in $\mathbf{R}^2$ with a polygon in $\mathbf{R}^2$. We now have a two-dimensional clipping problem of the type considered in Chapter 3. If the polygon is convex, then one can use a Cyrus-Beck approach.

Next, assume that the ray was not parallel to the plane containing the polygon. One needs to find the intersection of **X** with the plane containing the polygon. If such an intersection exists, one will then have to check if it lies in the polygon. This is a point-in-polygon type problem that we have already considered in Section 6.3, except that now we have an arbitrary plane rather than just $\mathbf{R}^2$. To reduce the problem to one of two coordinates rather than three, we project the point and polygon orthogonally to the xy-, xz-, or yz-plane depending on whether $|n_3|$, $|n_2|$, or $|n_1|$ is the largest, respectively, where $\mathbf{n} = (n_1,n_2,n_3)$ is a normal vector to the plane. Of course, the projection involves no work. It is simply a case of using the corresponding two coordinates of the 3-coordinate points. The motivation behind making our choice of projection depend on the largest component of **n** is numerical accuracy. We want to have the projected polygon as large as possible and not have its vertices projected too closely together.

10.2.2.5 Problem. To find the intersection, if any, of the ray **X** and a convex quadrilateral.

Solution. This problem could be considered to be a special case of Problem 10.2.2.4 but because it is often encountered it is worth considering separately. For example, parametric surfaces are often represented by a grid of points gotten by evaluating their parameterization at the vertices of the subrectangles from a subdivision of their rectangular domains. Although such surface quadrilaterals may not actually be planar or convex, they may be assumed to be so since they will be very small. There are also adaptive tessellation schemes that can be used to guarantee that the quadrilaterals are within a given tolerance of being convex and planar.

The first part of the solution is the same as that in Problem 10.2.2.4. It is the test as to whether a point in the plane belongs to a planar polygon that can be improved in the case of a quadrilateral. We describe the solution presented in [SchS95]. It is a generalization of Badouel's ([Bado90]) point-in-triangle solution. Assume that we are in $\mathbf{R}^2$. Consider a point **Q** and a nondegenerate convex quadrilateral with vertices **A**, **B**, **C**, and **D**. Any point of **ABCD** can be expressed in the form

$$\mathbf{G} + s\,\mathbf{GH}, \text{ where } \mathbf{G} = \mathbf{A} + r\,\mathbf{AB} \text{ and } \mathbf{H} = \mathbf{D} + r\,\mathbf{DC} \text{ and } 0 \le r, s \le 1.$$

See Figure 10.5. Therefore, the point **Q** will belong to **ABCD** if and only if

$$\mathbf{AQ} = r\,\mathbf{AB} + s\,\mathbf{AD} + rs\,\mathbf{AE},$$

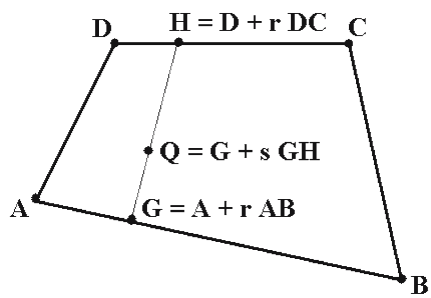


Figure 10.5. Surrounding test for quadrilateral.

where

$$\mathbf{AE} = \mathbf{DC} + \mathbf{BA} = \mathbf{DA} + \mathbf{BC}$$

and $0 \le r, s \le 1$. The two quadratic equations in two unknowns r and s are easily solved and one simply has to check if r and s lie in [0,1]. The solutions become even easier if the quadrilateral is a trapezoid with two parallel sides.

10.2.2.6 Problem. To find the intersection, if any, of the ray **X** and an oriented faceted surface.

Solution. This problem reduces to checking for an intersection of the ray with each of the facets of the surface using the solution to Problem 10.2.2.4. In fact, we only have to look at front-facing (front) facets, that is, facets whose normals (induced from the orientation) have a nonpositive dot product with **v**.

10.2.2.7 Problem. To find the intersection, if any, of the ray **X** and a generalized bounding box $\mathbf{B} = B(\mathbf{X},\mathbf{n}_1,\mathbf{n}_2, \ldots ,\mathbf{n}_k)$ (as defined in Section 6.2).

Solution. First of all, finding the intersection of the line **L** with a slab is easy. One simply substitutes the equation for the line into the equation of the far and near planes for the slab. This will produce an interval $[a_i,b_i]$ of parameters t so that $\mathbf{p} + t\mathbf{v}$ lies in the ith slab. Let **I** be the intersection of these intervals. Equivalently, if

$$a = \max\{t \mid \mathbf{p} + t\,\mathbf{v} \text{ lies on one of the near planes}\}$$

and

$$b = \min\{t \mid \mathbf{p} + t\,\mathbf{v} \text{ lies on one of the far planes}\},$$

then $\mathbf{I} = [a,b]$. Finally, if $\mathbf{I} \neq \phi$ and $a \ge 0$, then $\mathbf{p} + a\mathbf{v}$ is the nearest intersection of **X** with **B**.

10.2.3 Ray Tracing CSG Objects

Ray tracing CSG objects is much simpler than constructing them. The basic idea, described in [Roth82], only involves computing ray intersections with the primitives used in the CSG tree that defines the object and then combining the resulting one-dimensional segments with the set operations from that CSG tree.

Let T be a CSG tree that defines an object **X**. Let **r** be a ray with starting point $\mathbf{p}_0$ and unit direction vector **v**. Let LT and RT be the left and right subtree of the root of T and assume that they correspond to objects **A** and **B**, respectively. The ray **r** will intersect **A** and **B** in a collection of segments S_A and S_B.

Given the operation **op** at the root of T, the ray **r** will intersect **X** in the segments S_A **op** S_B. See Figure 10.6. This leads to Algorithm 10.2.3.1. In the algorithm we have made use of the fact that if the ray does not intersect the object corresponding to the

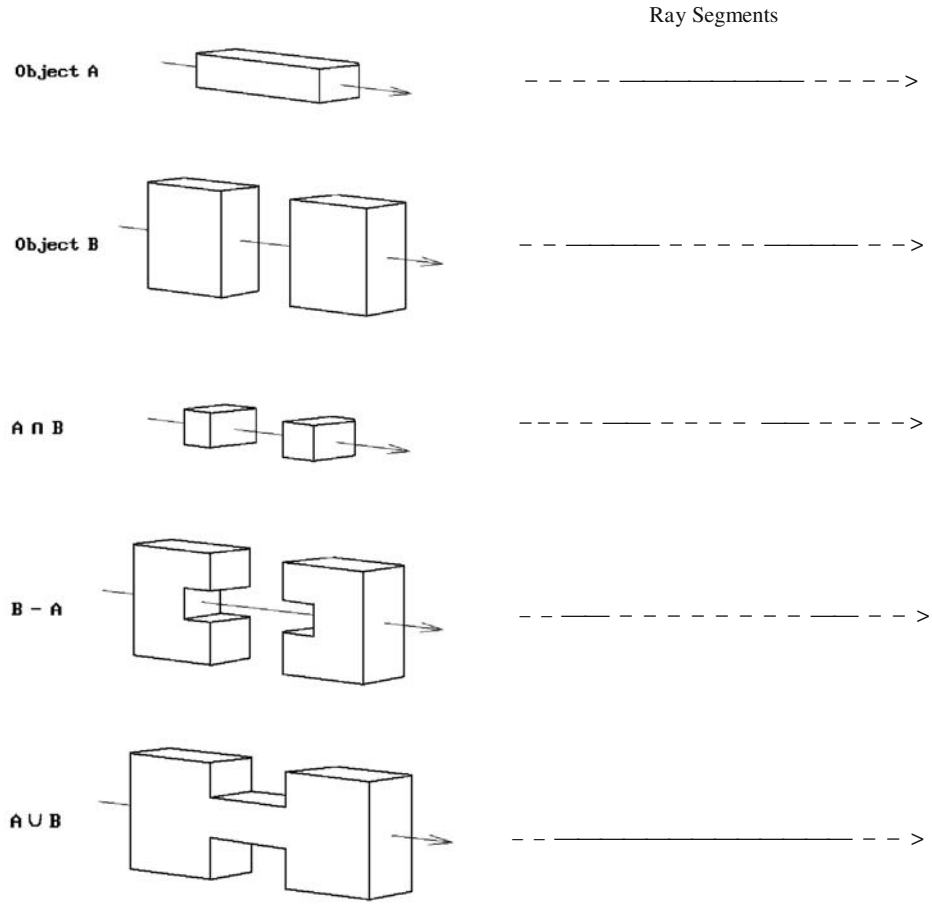


Figure 10.6. CSG ray tracing.

left subtree of the CSG tree for an object **X**, then it will also miss **X** except possibly in the case where the operation at the root is the union operation.

The CombineSegs function in Algorithm 10.2.3.1 described in [Roth82] consists of three steps:

(1) One merges the sorted endpoints of the segments in LL and RL to produce a new segmentation of the ray.
(2) The resulting segments are classified as "in" or "out" depending on the set operation being applied to the left and right subtree of the CSG tree and the classification of the segments in LL and RL.
(3) Lastly, contiguous segments with the same classification are merged to simplify the result.

Step (2) uses the following lookup table:

```
segList function CSG_RayIntersect (CSGtree T, ray r)
begin
    segList LL, RL;

    if IsPrimitive (T)
        then return RayIntersection (T, r)
                {RayIntersection computes the intersection of the ray with a primitive
                 object directly on a case by case basis using its special properties.}
        else
            begin
                LL := CSG_RayIntersect (LeftSubtree (T), r);
                if (Empty (LL) and (Op (T) ≠ ∪))
                    then return nil
                    else
                        begin
                            RL := CSG_RayIntersect (RightSubtree (T), r);
                            return CombineSegs (LL, RL, Op (T));
                        end
            end
end;
```

Algorithm 10.2.3.1. CSG ray intersection algorithm.

Left	Right	$\cup$	$\cap$	$-$
in	in	in	in	out
in	out	in	out	in
out	in	in	out	out
out	out	out	out	out

Finally, if all one wants is to determine whether or not a ray intersects an object and one is not interested in the actual intersection, then one can use a simplified version of Algorithm 10.2.3.1. There is no need to check for an intersection of the right subtree if one determines that the left subtree was intersected.

To speed things up one should use bounding boxes at each node of the CSG tree. Bounding boxes work best when they are lined up with the viewing coordinate system. Maintaining this would mean that one has to recompute them whenever the view changes. If this happens a lot, one could use bounding spheres, but boxes typically fit objects better than spheres.

10.3 The Radiosity Method

As indicated in Section 9.4.4, radiosity methods were another step to model illumination more accurately. They correct some defects in the ray-tracing approach. In par-

ticular, they solve the problem of interactions of diffuse light in a closed environment in a theoretical way. The theoretical foundation of the radiosity method lies in the theory of heat transfer or exchange between surfaces and the conservation of energy. It involves radiometry and transport theory. Because the radiosity method is technically rather complicated, all we shall do in the next two sections is sketch its overall strategy and implementation. The interested reader is directed to [CohW93], [FVFH90], [WatW92], [CohG85], or [CCWG88] for more details.

If we assume perfect diffuse (Lambertian) surfaces, then the rate of energy leaving a surface (the radiosity) is equal to the sum of the self-emitted energy and the energies that came as reflections from other surfaces. This leads to an equation for the radiosity function $B(\mathbf{p})$ at a point $\mathbf{p}$ of the form

$$B(\mathbf{p}) = E(\mathbf{p}) + \rho(\mathbf{p})\int_S B(\mathbf{q})G(\mathbf{p},\mathbf{q})dA, \tag{10.3}$$

where

$E(\mathbf{p})$ is an emitted light function,
$\rho(\mathbf{p})$ is a diffuse reflectivity function,
$G(\mathbf{p},\mathbf{q})$ is a function of the geometric relationship between $\mathbf{p}$ and $\mathbf{q}$, and the integration is over all surfaces in the environment.

Equation (10.3) is called the *Radiosity equation*. It is a special case of the rendering equation (9.12).

If we now subdivide all surfaces into patches A_i over which the radiosity and emitted energy are essentially constant with constant value B_i and E_i, respectively, then equation (10.3) leads to the following equation relating the B_i:

$$B_i A_i = E_i A_i + \rho_i \sum_{j=1}^{n} B_j F_{ji} A_j, \tag{10.4}$$

where

E_i is the light emitted from the ith patch,
ρ_i is the reflectivity of the patch, namely, the fraction of the light that arrives at the patch that is reflected back into the environment, and
F_{ji} is the fraction of the energy leaving the jth patch which reaches the ith patch.

The factors F_{ji} are called *form factors* and are assumed to depend only on the geometry of the surfaces in the environment. In fact, using the identity

$$F_{ij} A_i = F_{ji} A_j \tag{10.5}$$

equation (10.4) reduces to

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij} \tag{10.6}$$

Solving for the B_i reduces to solving a system of linear equations that looks like

$$\begin{pmatrix} 1-\rho_1F_{11} & -\rho_1F_{12} & . & . & -\rho_1F_{1n} \\ -\rho_2F_{21} & 1-\rho_2F_{22} & . & . & -\rho_2F_{2n} \\ . & . & . & . & . \\ . & . & . & . & . \\ -\rho_nF_{n1} & -\rho_nF_{n2} & . & . & 1-\rho_nF_{nn} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ . \\ . \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ . \\ . \\ E_n \end{pmatrix} \tag{10.7}$$

in matrix form. If we define a matrix $\mathbf{K} = (K_{ij})$, where $K_{ij} = \delta_{ij} - \rho_i F_{ij}$, then equation (10.7) can be written more compactly as

$$\mathbf{KB} = \mathbf{E}. \tag{10.8}$$

When the B_i have been solved for, we will have a single radiosity value for each of our surface patches. We can now use a standard Gouraud shading model renderer to display the world. The renderer will actually need illumination values at vertices of the surfaces, but these can be obtained by averaging the illumination values of the patches surrounding a vertex. In general terms, the overall steps for a radiosity renderer are then

generate the surfaces for a scene	→	subdivide them into sufficiently small patches	→	compute the form factors	→	solve the matrix equation (10.8)	→	display world using a standard renderer

The first four of these steps are view independent, which means that the most time-consuming computations have to be made only once.

Because the matrix equation (10.8) involves a large matrix, direct methods for solving it, like Gaussian elimination, are unsuitable and one is led to use iterative methods. The idea is to make an initial guess $\mathbf{B}^{(0)}$ and then by means of a sequence of corrections produce a sequence of new guesses that hopefully converge to a solution. Let $\mathbf{B}^{(k)}$ denote the kth guess and define the *residual* $\mathbf{r}^{(k)}$ by

$$\mathbf{r}^{(k)} = \mathbf{KB}^{(k)} - \mathbf{E}.$$

The size of a residual tells us how close we are to an actual solution. An iterative algorithm of this type is called a *relaxation method* if at each step of the iteration a guess is updated in such a way as to set one of the residuals to zero. Although this may change the other residuals, one hopes that an overall improvement has occurred. Algorithm 10.3.1 is an $O(n^2)$ algorithm, which computes an approximation to the radiosity vector $\mathbf{B}$. It is based on a relaxation method called the Gauss-Seidel algorithm. The justification for the formula in the algorithm is derived from the fact that to make the ith component of the residual for the kth solution $\mathbf{B}^{(k)}$ zero, we must replace the ith component $B_i^{(k)}$ of $\mathbf{B}^{(k)}$ by

$$B_i^{(k)} + \frac{1}{K_{ii}}\left(E_i - \sum_{j=1}^{n} K_{ij}B_j^{(k)}\right).$$

Make an initial guess **B** for the radiosity vector.

while **B** has not yet converged **do**
  for i:=1 **to** n **do**

$$B_i := E_i - \sum_{j=1, j\neq i}^{n} \frac{K_{ij} B_j}{K_{ii}}$$

Algorithm 10.3.1. A Gauss-Seidel algorithm for computing the radiosity vector.

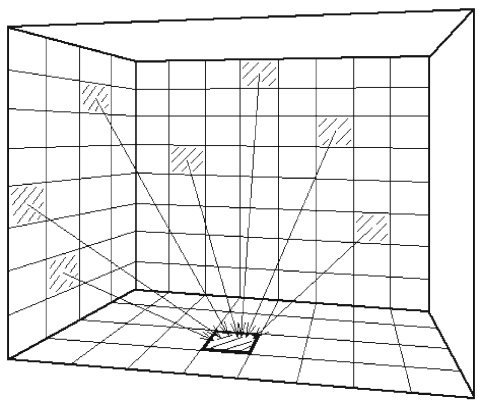


Figure 10.7. Gathering contributions in Gauss-Seidel radiosity algorithm.

If there is no information to aid in making an initial guess, then set $\mathbf{B}^{(0)}$ equal to the light source radiosities **E**. The convergence criterium is usually to require that the largest of the $|\mathbf{r}_i^k|$, i = 1,2, . . . ,n is sufficiently small. One can show that the matrices in (10.8) are of a form that guarantees the convergence of the solutions that the algorithm generates.

In the terminology of [CCWG88], the Gauss-Seidel algorithm corresponds to updating the ith patch at each iteration by "gathering" together the contribution of the radiosities from all other patches as suggested in Figure 10.7. A more efficient way to compute the radiosity is what is called a "progressive refinement" method. Here one updates the whole picture during each iteration rather than just a single patch. One basically reverses the procedure and asks what contribution the ith patch makes to the radiosity of all the other patches. Equation (10.6) shows that

$$\rho_i B_j F_{ij}$$

determines the radiosity contribution of patch j to patch i. Using equation (10.5), one sees that

$$\rho_i B_j F_{ij} \frac{A_i}{A_j}$$

```
B  := E;
ΔB := E;
while B has not yet converged do
    begin
        Pick an i so that A_i ΔB_i is the largest (to speed convergence);
        Calculate the row of form factors F_ij using a hemicube centered on patch i ;
        for each j ≠ i do
            begin
                Δrad := ΔB_i ρ_j F_ij A_i / A_j;
                ΔB_j  := ΔB_j + Δrad;
                B_j   := B_j + Δrad;
            end;

        ΔB_i := 0;
    end;
```

Algorithm 10.3.2. A progressive refinement algorithm for computing the radiosity vector.

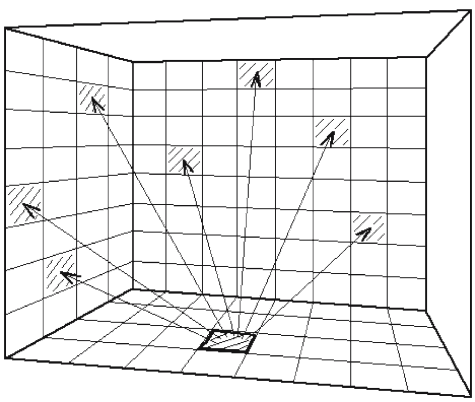


Figure 10.8. A progressive refinement radiosity algorithm.

determines the radiosity contribution of patch i to patch j. The new algorithm, Algorithm 10.3.2, will compute this contribution for all patches j. We can think of this as "shooting" light out from patch i into the environment. See Figure 10.8. As the algorithm iterates over various ith patches, the same i may be repeated several times. At a subsequent use of i, since the environment has already used the previous B_i, we only need to consider the difference, $\Delta\mathbf{B}_i$, between the previous and current value of B_i.

One of the nice features of Algorithm 10.3.2 is that, rather than having to wait until the final picture is computed, one can watch intermediate renderings. The algorithm as it stands will start with a dark picture that gradually turns light. To get a

brighter picture sooner, [CCWG88] describes a way of introducing an initial ambient term.

10.3.1 Form Factors: The Hemicube Method

The lengthiest computations of the radiosity method are evaluating the form factors. These factors arise from an integral of the form

$$F_{ij} = \frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{\cos\phi_i\cos\phi_j}{\pi r^2}dA_j dA_i \tag{10.9}$$

These factors are not easy to compute. Some analytical formulas are known dealing with areas such as polygon to polygon (without occlusion), differential area to polygon, and some special cases such as rectangle to rectangle and point to disk. They are, however, purely geometric and do not depend on the aspects of light, such as reflectivity or emissivity. One mostly has to use numerical approaches.

The integral in (10.9) can be approximated in different ways. One can use the trapezoidal rule or Simpson's rule, for example. The standard approach there is to sample at evenly spaced points. On the other hand, if one knows something about the integrand, then one might be able to sample less by sampling at the most significant places. One of the early and efficient methods of computing form factors is what is called the *hemicube method*. This is the only method we shall describe here.

To begin with, let us make the assumption that patches are far apart relative to their size. Two simplifications follow from this assumption. First, the inner integral is essentially constant, so that the outer integral with the $1/A_i$ term essentially disappears and we may assume an approximation of the form

$$F_{ij} \approx F_{dij} = \int_{A_j}\frac{\cos\phi_i\cos\phi_j}{\pi r^2}dA_j \tag{10.10}$$

Second, two patches that have the same projection onto the surface of a hemisphere centered on the patch A_i have the same form factor. See Figure 10.9. Rather than dealing with a hemisphere, it is more convenient to use a hemicube centered on A_i. See Figure 10.10. Subdivide this hemicube surface into small rectangles Q_{st}. The form factor for each Q_{st} has an approximation

$$\Delta F_{Q_{st}} = \frac{\cos\phi_i\cos\phi_j}{\pi r^2}\Delta A_{st},$$

where ΔA_{st} is the area of the rectangle Q_{st}. One precomputes these form factors and stores them in a lookup table. To compute the form factor F_{ij} one simply adds up the form factors for all the rectangles Q_{st} onto which the patch A_j projects, that is,

$$F_{ij} = \sum_Q \Delta F_Q, \tag{10.11}$$

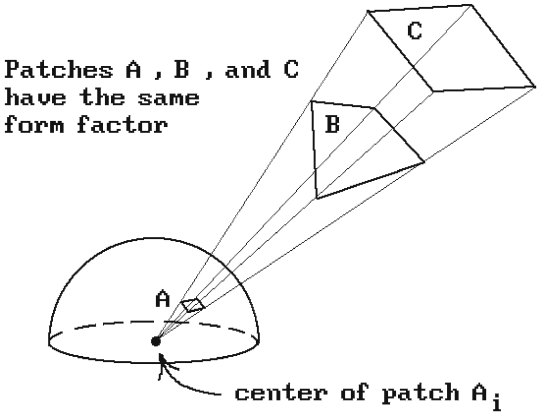


Figure 10.9. Form factors using hemispheres.

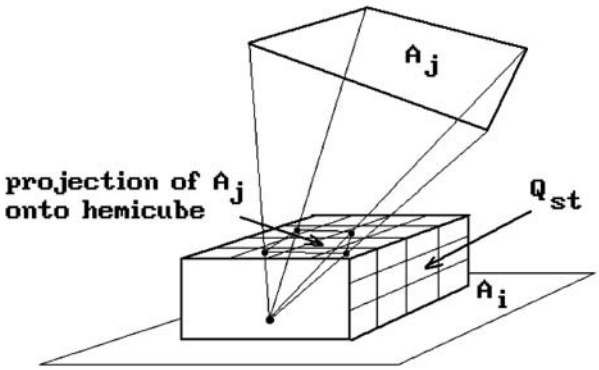


Figure 10.10. Form factors using rectangles.

where Q ranges over all rectangles Q_{st} that lie in the projection of A_j onto the hemicube. A three-dimensional version of the Cohen-Sutherland clipping algorithm can be used to compute the projection of A_j onto the hemicube. One can also handle occlusions of patches. This is done on the R_{ij} level, that is, for each rectangle Q_{st} in the projection, use the form factor of that patch A_j that is closest. There is a similarity with the z-buffer algorithm, except that here we maintain labels for the nearest patch rather than a z-value. Figure 10.11 shows a cross-section of an example.

The formulas for the form factors ΔF_Q in equation (10.11) are easy to compute. For rectangles Q in the top of the hemicube we have

$$\Delta F_Q = \frac{1}{\pi\left(x^2 + y^2 + 1\right)^2}\Delta A.$$

This follows from the geometry shown in Figure 10.12. There are similar formulas for Q in the side of the hemicube. For example, for Q in the plane x = 1, we have

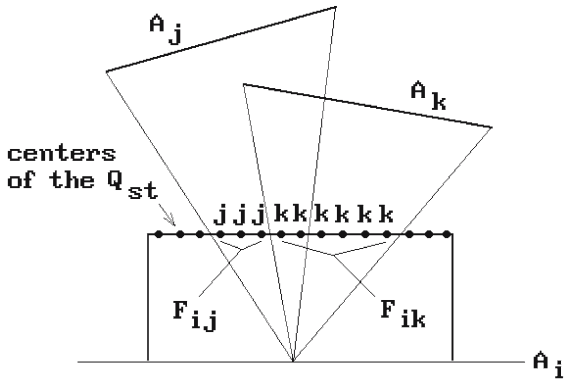


Figure 10.11. Handling occlusions with form factors.

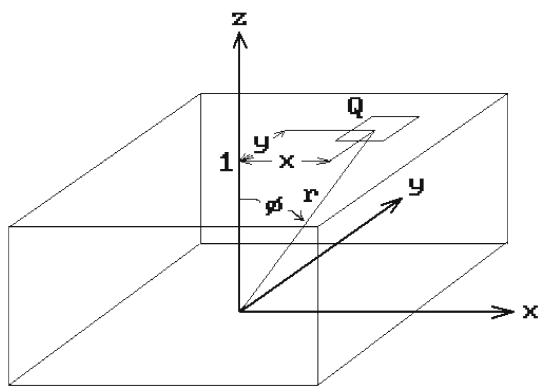


Figure 10.12. The geometry for rectangular form factors.

$$\Delta F_Q = \frac{z}{\pi\left(y^2 + z^2 + 1\right)^2}\Delta A.$$

Finally, as mentioned earlier, the biggest cost in the radiosity method is in computing the form factors. Precomputing them helps, but there is a storage problem. The form factor matrix is basically an n^2 matrix. Since many patches cannot see each other, many entries of this matrix are zero, but many nonzero entries remain. An estimate from [CCWG88] indicates that one would need on the order of a gigabyte of memory to store the form factors with 50,000 patches. Using the progressive refinement algorithm this n^2 storage requirement is eliminated.

Aliasing also rears its ugly head with radiosity methods. The biggest culprit is the uniform subdivision used in the hemicube method. One approach ([WaCG87]) is to change the orientation of the hemicube with respect to the surface normal in a random way as one moves from patch to patch.

Being view independent, radiosity methods separate the shading problem from the visible surface determination problem. Because the basic radiosity method does not handle the specular part of light very well, something that ray tracing does, there

are now solutions to the global illumination problem that combine these two approaches. See, for example, [NeuN95].

10.4 Volume Rendering

Up to now, the type of rendering of three-dimensional objects we have been discussing is sometimes called *surface rendering*, because it assumed that objects were represented by their boundary and so that was what we had to display. The "interior" of an object was never considered. This approach works fine for objects that have well-defined surfaces, but does not work so well when modeling natural phenomena such as clouds, fog, smoke, etc. Here the light partially penetrates the objects and their interior becomes important and volume rendering comes to the rescue. This and the next two sections give an overview of volume rendering and some of the algorithms it uses. Two general references are [Elvi92] and [LiCN98]. Elvins describes several additional algorithms.

The complete volume-rendering pipeline really consists of three parts: data acquisition, data classification, and the actual rendering algorithms. The data acquisition part will not be considered here. We assume that we have been given some volumetric data consisting of a cubical collection of voxels. Data classification is actually quite a tricky part of the overall process and refers to deciding on criteria for what part of the raw data to use. After one has decided on a classification, the last step is to display the geometry it represents.

In the past (see [KaCY93]), *volume rendering* was sometimes defined as a technique for visualizing volumes **directly** from the volumetric data without the use of surfaces or other explicit intermediate representations of the geometry. We shall use the term in a more general sense, so that it includes any technique that is used to render volumetric data. Some techniques in fact do involve constructing surfaces, such as the marching cube algorithm described in Section 10.4.2.

We look at direct volume rendering first and *ray casting* approaches. (The terms "ray tracing" and "ray casting" are usually used interchangeably, but some prefer to use the term "ray casting" in the volume rendering context because they give it the more restricted meaning that the rays are sent in only a single direction, in contrast to "ray tracing," which for them suggests that the rays bounce around in all directions in the scene.) Similar to visible surface algorithms, they can be classified as image precision or object precision type algorithms.

Image Precision Volume Rendering. Here, we send out a three-dimensional ray for each pixel on screen. For a parallel projection these rays would be perpendicular to the view plane. See Figure 10.13. The rays can be parameterized and density values associated to points evaluated at uniform intervals or they could be discrete rays generated by a Bresenham-type algorithm (see Section 10.4.1). In either case, we would have to interpolate the density values at their points from the adjacent voxel data that we were given. One simplification in the case where we use discrete rays in a parallel projection is that we can use "templated" discrete rays. What this means is that we only need to compute one ray starting at one pixel and then the relative movement from one voxel to another would be the same for the rays at all the other pixels. One

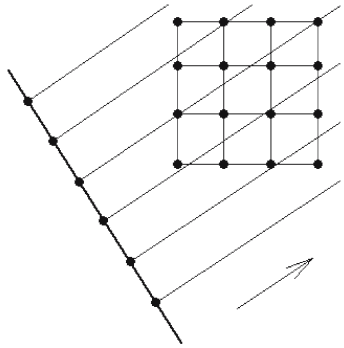


Figure 10.13. Image precision rays for volume rendering.

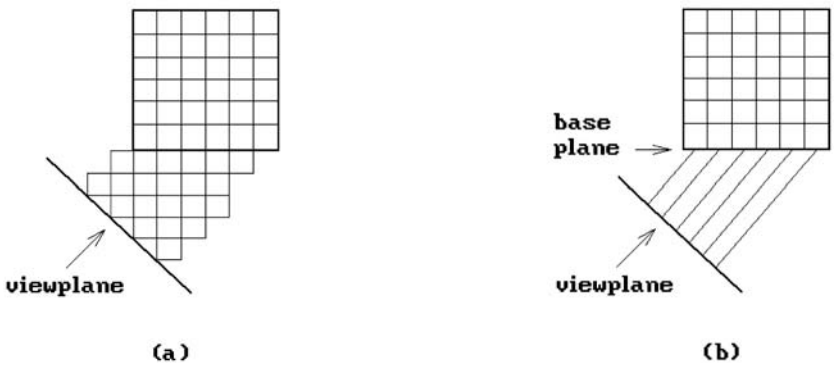


Figure 10.14. Image plane versus base plane ray casting.

has the option of starting these rays at the view plane or at a base plane of the voxel volume. See Figure 10.14. In the former case one may miss some voxels and in the latter case we do not, but we may get some warping, which requires some resampling to map the result back to the view plane. Given density values along the ray we then have to decide how to use these values to get a value for the pixel to which they project. This is the compositing problem that we consider shortly.

Object Precision Volume Rendering. In this approach we project to the view plane along rays starting at voxels in the view volume. See Figure 10.15. Since voxels do not necessarily project to pixels in the view plane, we can see that there are some potential problems that have to be handled. As far as assigning a density value to a pixel, we can do that based on a weighting of the values of voxels that map to a neighborhood of the pixel. To overcome holes we can use a technique called *splatting* (see [West90]), which distributes values of a voxel over a region around the point in the view plane to which the voxel projects. Another problem one has to deal with is gaps caused by getting too close to the voxels.

In each of the ray-casting methods above we run into a compositing problem. The problem is that potentially many voxels map onto the same pixel and combining the

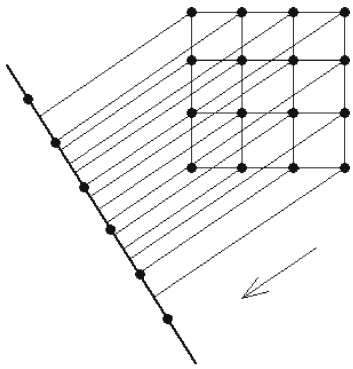


Figure 10.15. Object precision rays for volume rendering.

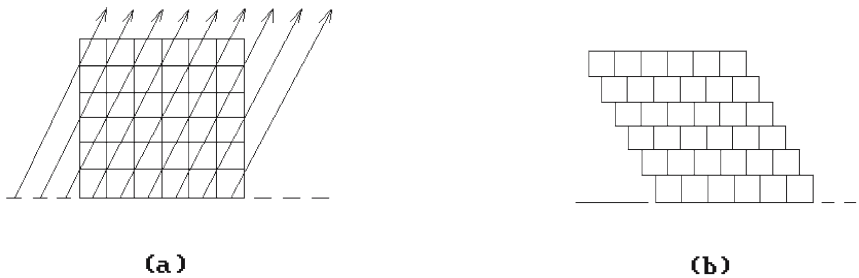


Figure 10.16. Shear-warp volume rendering.

density values at these voxels to come up with a single value at the pixel is called *compositing*. From a theoretical point of view, one should integrate the intensities along rays, but different compositing functions can be chosen. The simplest is to project the maximum density along the ray to the pixel. This is often useful when trying to isolate well-defined structures in the view volume. We can do the compositing in a front-to-back or back-to-front manner. One has the option of computing with the transparency t associated to a voxel or its *opacity* o = 1 − t.

[LacL94] describes an efficient variant of ray casting called *shear-warp volume rendering*. Rather than sending out rays that are skew to the voxel volume (Figure 10.16(a)), they apply a shear transformation to the voxels to achieve an equivalent situation where rays are cast that are perpendicular to the view plane (Figure 10.16(b)). In this way one can define an efficient ray traversal. Two other methods used in the algorithm to speed things up is run-length encoding of the voxel scan lines (a definition of run-length encoding can be found in [Salo99], for example) and early ray termination, where one stops the compositing process when a pixel has reached full opacity.

Data classification in direct volume rendering amounts to defining the opacity or transparency of voxels. This is accomplished by defining what is called a *transfer function* that maps densities, or densities together with gradient information, to opacities. Typically, the transfer functions map ranges of densities and gradient values to the

same opacity. Opacity values make it possible to extract and make visible different parts inside an object. For example, if we want to see the bone structure, then we would set the other material to be transparent. Transfer functions can also be used to map to colors.

Once transfer functions have been defined, the rendering is automatic. Unfortunately, it is not possible to generate all possible such functions systematically because the features one is trying to extract may be hard to specify with a program. Defining suitable transfer functions remains one the difficult tasks in volume rendering. See [RhyT01]. Extracting features from data is called *segmentation* and sometimes requires user input and, in the worst case, may have to be done entirely by hand. In volume rendering it is basically a labeling procedure applied to voxels to indicate their material type. It is a preprocessing procedure applied to the data before it is rendered and the segmentation information is stored along with the other voxel data that can then be used by transfer functions.

Now, in many applications of volume rendering, the issue is not photorealism but making data meaningful. For that reason parallel projection is typically used in volume rendering. For typical medical applications nothing is gained by perspective views. Furthermore, perspective views have problems in that, since the rays diverge, they may miss voxels and create aliasing problems.

To get shading one needs normals. To get these one typically uses the gradient

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

of the density function f. The central difference operators

$$f(i+1, j, k) - f(i-1, j, k),$$

$$f(i, j+1, k) - f(i, j-1, k),$$

$$f(i, j, k+1) - f(i, j, k-1),$$

are the most common approximations to the partial derivatives $\partial f/\partial x$, $\partial f/\partial y$, and $\partial f/\partial z$, respectively, but there are others.

Finally, sometimes one knows that there are surfaces present in the volumetric data. Two well-known approaches used in volume rendering try to construct a conventional surface model **S** from the data that is then rendered in the ordinary way.

Approach 1. Here one proceeds in two stages: first one determines the curves that make up the contour of **S** in each slice and then one tries to connect these contours with surface patches, a process called skinning. Figuring out how to connect the contours from one slice to the next is an especially tricky problem, because a contour may consist of several curves. Chapter 14 will have more to say about finding contours and skinning.

Approach 2. In Approach 1 separate and independent algorithms are used in the two stages. There are more general approaches to finding the surface **S** that work on

the three-dimensional data directly. Because **S** corresponds to the voxels having some specified density values it is also sometimes called a contour or isosurface and the algorithms that find it, contour algorithms. One is called the *marching cube algorithm*. It is more complicated though. We give an overview of the algorithm in Section 10.4.2. Another algorithm that finds **S** using octrees is described in [WilV90a]. A third algorithm due to Ehud Artzy is described in [Herm98]. Here we assume that the voxels have been assigned a value of 0 or 1. In practice, the voxels with value 1 correspond to the tissue of interest in a CT scan. The algorithm then determines all the faces of the voxels that are in the boundary of a connected component of the set of voxels marked with 1. Although Artzy's algorithm is not hard to describe, proving that it works relies on a theorem in discrete topology (one proved in [Herm98]).

When using surfaces in volume rendering one problem is the fact that defining a surface is a binary decision process (a surface divides a region into two parts). Since that decision is made on sampled functions, one can easily end up with spurious or missing surface parts. With regard to the aliasing problem, we run into an added complication in volume rendering. Since we only have the sampled data that was originally presented to us, we cannot readily generate new samples on our own. Basically, the responsibility lies with the original data having been sampled adequately.

Finally, there are many nice aspects to volume rendering. Some tasks that are hard with other rendering methods become easy here. Consider again the example of how easy it is to modify the geometry and create the effect of cutting a hole in an object to look inside. Levoy ([Levo90]) describes a hybrid ray tracer for both volume and polygon data.

10.4.1 Discrete Three-Dimensional Lines

Section 2.2 already introduced the basic discrete topology concepts. Specifically, we defined what is meant by a curve in $\mathbf{Z}^3$ that is 6-, 18-, or 26-connected. Our goal now is to describe some three-dimensional analogs of the two-dimensional Bresenham line-drawing algorithm. These are the algorithms used in volume rendering to define voxelized rays. Because of the higher dimension, things get somewhat more complicated. Furthermore, since we are interested in the application of this to volume rendering, we have to address some issues that were not considered in Chapter 2.

We start again with the two-dimensional case because examples are easier to draw and therefore clearer. When boundaries of sets are represented in a discrete way, we have to worry about "holes" or "tunnels" through which a ray can pass. Figure 10.17(a) shows an example of an 8-connected ray passing through an 8-connected boundary without intersecting it. Figure 10.17(b) shows the same thing in three dimensions, namely, an 18-connected ray passing through a 6-connected object. We do not want to allow this, otherwise, the rendering of our voxelized objects will be flawed. To avoid the problems demonstrated by Figure 10.17, we must make sure that objects and rays are suitable connected.

The algorithms for generating discrete rays in 3-space are motivated by the midpoint line-drawing algorithm we used in the plane. See Algorithm 2.5.3.1 in Section 2.5.3. Suppose that we want to draw a discrete version of a line **L** from $\mathbf{p}_0 = (x_0,y_0,z_0)$ to $\mathbf{p}_1 = (x_1,y_1,z_1)$. Let

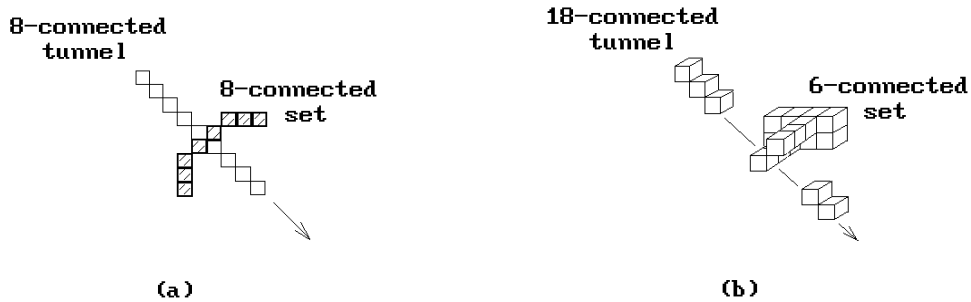


Figure 10.17. Rays tunneling through objects.

$$dx = x_1 - x_0, \quad dy = y_1 - y_0, \quad \text{and } dz = z_1 - z_0.$$

We shall describe an algorithm for generating 26-connected lines first.

Assume that $dx \geq dy \geq dz \geq 0$. Let $\mathbf{L}_{xy}$ and $\mathbf{L}_{xz}$ be the projections of $\mathbf{L}$ on the xy- and xz-plane, respectively. We shall build our discrete line from the midpoint line drawing algorithm applied to the lines $\mathbf{L}_{xy}$ and $\mathbf{L}_{xz}$. Now the line $\mathbf{L}_{xy}$ goes from $(x_0,y_0,0)$ to $(x_1,y_1,0)$ and $\mathbf{L}_{xz}$ goes from $(x_0,0,z_0)$ to $(x_1,0,z_1)$. Using the notation of Algorithm 2.5.3.1 in Section 2.5.3, let d_{xy} be the decision variable for the line $\mathbf{L}_{xy}$ and $posInc_{xy}$ and $negInc_{xy}$ the increments of this variable. Similarly, let d_{xz}, $posInc_{xz}$, and $negInc_{xz}$ be the corresponding variables for the line $\mathbf{L}_{xz}$. Since the metric associated to 26-connected curves is the max metric, our assumptions on the relative sizes of dx, dy, and dz mean that the x-coordinate of the discrete points we generate will increase by one each time, so that the curve will have length dx. Furthermore, we can determine how the y-coordinate and z-coordinate change by looking at the midpoint algorithm applied to the lines $\mathbf{L}_{xy}$ and $\mathbf{L}_{xz}$, respectively. If we fill in the details we get Algorithm 10.4.1.1. Finally, for a complete algorithm that generates 26-connected discrete lines, one that handles lines between any two points, we need two more versions of Algorithm 10.4.1.1 to handle the case of nonnegative dx, dy, and dz, namely, one where dy is the largest increment and one where dz is the largest. The rest of the cases where one or more of the dx, dy, or dz are negative are handled by symmetry.

Next, we look at the case of 6-connected lines. We shall describe the algorithm presented in [CohK97]. This time we only need to assume that $dx, dy, dz \geq 0$. In addition to the projections $\mathbf{L}_{xy}$ and $\mathbf{L}_{xz}$ we also consider the projection $\mathbf{L}_{yz}$ of the line $\mathbf{L}$ onto the yz-plane. Define functions $f_i(x,y)$ by

$$f_1(x,y) = (dy)x - (dx)y + c_1$$

$$f_2(x,z) = (dz)x - (dx)z + c_2$$

$$f_3(y,z) = (dz)y - (dy)z + c_3.$$

Then the lines $\mathbf{L}_{xy}$, $\mathbf{L}_{xz}$, and $\mathbf{L}_{yz}$ are defined by $f_1(x,y) = 0$, $f_2(x,z) = 0$, and $f_3(y,z) = 0$, respectively. Assume that pixels are centered on points with integer coordinates. If the line crosses a pixel centered at (x,y,z), then it will leave the pixel via one of the three faces adjacent to the point $\mathbf{O} = (x + 0.5, y + 0.5, z + 0.5)$. See Figure 10.18(a). We shall

```
procedure DrawLine (integer x0, y0, z0, x1, y1, z1)
{ We assume that (x1 – x0) ≥ (y1 – y0) ≥ (z1 – z0) .
  The procedure Draw (x,y,z) is assumed to draw a voxel at location (x,y,z) on the
  three-dimensional raster. }
begin
    integer dx, dy, dz, dxy, posIncxy, negIncxy, dxz, posIncxz, negIncxz, x, y, z;

    dx := x1 – x0;    dy := y1 – y0;    dz := z1 – z0;
    dxy := 2*dy – dx;    posIncxy := 2*dy;    negIncxy := 2*(dy – dx);
    dxz := 2*dz – dx;    posIncxz := 2*dz;    negIncxz := 2*(dz – dx);
    x  := x0;    y := y0;    z := z0;
    Draw (x, y, z);
    while x < x1 do
        begin
            if dxy ≤ 0
                then
                    begin
                        dxy := dxy + posIncxy;
                        if dxz ≤ 0
                            then dxz := dxz + posIncxz;
                            else
                                begin
                                    dxz := dxz + negIncxz;
                                    z    := z + 1;
                                end
                    end
                else
                    begin
                        dxy := dxy + negIncxy;
                        if dxz ≤ 0
                            then dxz := dxz+ posIncxz;
                            else
                                begin
                                    dxz := dxz + negIncxz;
                                    z    := z + 1;
                                end
                        y := y + 1;
                    end;
            x := x + 1;
            Draw (x, y, z);
        end
end;
```

Algorithm 10.4.1.1. A 26-connected line drawing.

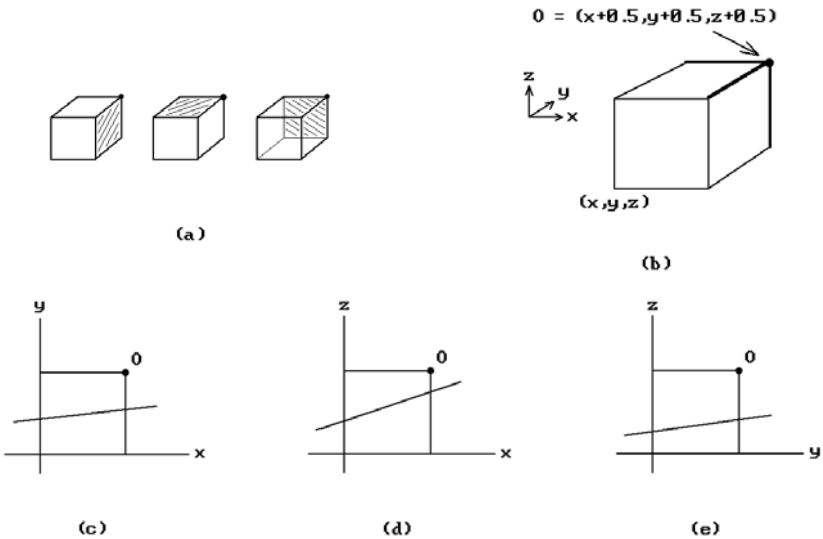


Figure 10.18. Tripod 6-connected line-drawing algorithm cases.

determine the actual face by which it leaves by analyzing the projections $\mathbf{L}_{xy}$, $\mathbf{L}_{xz}$, and $\mathbf{L}_{yz}$ with respect to the projections of the three edges of the pixel that meet in the point **O**. See Figure 10.18(b–e). Note that, like in the midpoint line-drawing algorithm, the sign of $f_i(0.5,0.5)$ tells us the relative position of the corresponding line with respect to the point (0.5,0.5). If the sign is negative, the line is below the point. Since it is the taxicab metric that determines the length of lines in the case of 6-connected lines, our line will have dx + dy + dz + 1 points. Algorithm 10.4.1.2 implements the algorithm. The reason for the name is that the vertex **O** and the three edges emanating from it in Figure 10.18(b) have the appearance of a tripod.

Efficient C versions of Algorithms 10.4.1.1 and 10.4.1.2 can be found in [CohK97]. Rather than indexing into a voxel array the idea is to use a pointer to the beginning of the array and then, using pointer arithmetic, to add the appropriate offsets instead of updating indexes. [CohK97] also contains a discussion of and references for other algorithms for drawing three-dimensional discrete lines. Another algorithm and more references can be found in [Roge98].

[YaCK92] describes how rays generated using the algorithms of this section can be used to ray trace voxelized objects. Note that 26-connected rays contain a lot fewer pixels than 6-connected rays, but one has to worry about tunnels that would cause artifacts in the image. One can mix 26-connected and 6-connected rays, using the latter only when we are near objects.

10.4.2 The Marching Cubes Algorithm

The marching cubes algorithm was developed independently by [WyMW86] and [LorC87]. We follow the outline of the algorithm presented in [LorC87]. For an implementation of the algorithm we refer the interested reader to [WatW92].

```
procedure DrawLine (integer x0, y0, z0, x1, y1, z1)
{ We assume that  x1 − x0 , y1 − y0 , z1 − z0 ≥ 0 . The procedure Draw (x,y,z) is
  assumed to draw a voxel at location (x,y,z) on the three-dimensional raster. }
begin
    integer dx, dy, dz, dxy, dxz, dzy, num, i, x, y, z;

    dx := x1 − x0;    dy := y1 − y0;   dz := z1 − z0;
    dxy := dy − dx;    dxz := dz − dx;   dyz := dz − dy;
    num  := dx + dy + dz;
    i := 0;    x := x0;    y := y0;    z := z0;
    Draw (x, y, z);
    while  i < num  do
        begin
            if  dxy < 0
                then
                    begin
                        if  dxz < 0
                            then
                                begin
                                    x := x + 1;
                                    dxy := dxy + 2*dy;    dxz := dxz + 2*dz;
                                end
                            else
                                begin
                                    z := z + 1;
                                    dxz := dxz − 2*dx;    dyz := dyz − 2*dy;
                                end
                    end
                else
                    begin
                        if  dyz < 0
                            then
                                begin
                                    y := y + 1;
                                    dxy := dxy − 2*dx;    dyz := dyz + 2*dy;
                                end
                            else
                                begin
                                    z := z + 1;
                                    dxz := dxz − 2*dx;    dyz := dyz + 2*dz;
                                end
                    end;
            Draw (x, y, z);
        end
end;
```

Algorithm 10.4.1.2. The 6-connected tripod line drawing.

The setting for the algorithm is the following: It is assumed that the data has been acquired and possibly processed with some image processing techniques and is available as a three-dimensional array of voxels. If we think of the data as corresponding to some density values, then the object is to use this data to define a surface **S** specified in terms of a given density value σ. Data classification then amounts to using σ as a threshold value.

The algorithm proceeds as follows: For each voxel or cube we determine how its vertices are situated with respect to our surface **S** and then "march" on to the next cube. Each vertex is assigned a value of 1 if its density is greater than or equal to σ, that is, they are inside the surface, and 0 otherwise, if they are outside. This gives rise to a classification of the possible types of intersections of the surface with the cube, where the classification is based on which edges of the cube the surface intersects. We then simply use a table to look up what our intersection looks like for any particular case. The numbering of vertices allows, in principle, up to $2^8 = 256$ possible configurations of 1s and 0s for a given cube. Each case corresponds to an intersection type. Fortunately, one can use symmetry to reduce this number. One type of symmetry is based on the fact that the intersection type of the surface is unchanged if we swap what is considered as the "inside" and "outside" of the surface, that is, if we exchange 1s and 0s. This means that we only need to consider the cases where only zero to four vertices have a value of 1. A second type of symmetry is rotational symmetry. Making use of these symmetries, we can reduce our table of intersections to fourteen. These are shown in Figure 10.19. Solid disks indicate the vertices with value 1. Figure 10.19 also shows the triangulations that are used to approximate the intersection. The triangles are defined using linear interpolation of the vertex values. Since what is important is the intersection of the surface with the edges of the cube, we record the edge intersection information in our table. As it happens, two of the cases in Figure 10.19, cases 5 and 10, are ambiguous and these ambiguities have to be resolved. See [ScML98] or [WilV90b].

For rendering, one also needs normals for each triangle. These are also obtained via linear interpolation of the gradient of density values at the vertices of the cube. It is assumed that this gradient is nonzero along our surface. Now, it is a well-known fact (see Section 8.4 in [AgoM05]) that if a surface is defined by an equation $f(x,y,z) = c$ where f is some function and c is a constant, then the gradient of f, ∇f, is normal to the surface. To approximate the gradient at the vertices, one uses central differences:

$$\frac{\partial f}{\partial x} \sim \frac{f(i+1,j,k) - f(i-1,j,k)}{\Delta x}$$

$$\frac{\partial f}{\partial y} \sim \frac{f(i,j+1,k) - f(i,j-1,k)}{\Delta y}$$

$$\frac{\partial f}{\partial z} \sim \frac{f(i,j,k+1) - f(i,j,k-1)}{\Delta z}$$

where Δx, Δy, and Δz are the lengths of the sides of the cubes.

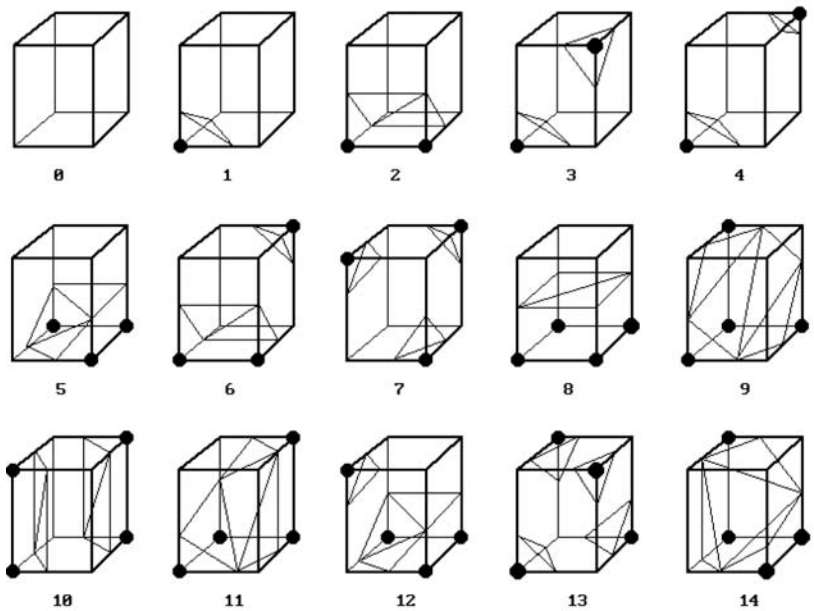


Figure 10.19. The marching cube algorithm cases.

[LorC87] applied the marching cube algorithm to three-dimensional medical data that provided two-dimensional slices of data. They used the following steps to create a surface from this data:

(1) Four slices of data were read into memory at a time.
(2) Two slices were scanned to create a cube from four neighbors on one slice and four neighbors on the next slice.
(3) By comparing the eight density values at the cube vertices with the surface constant an eight bit (one bit per vertex) index was calculated.
(4) This index was used to look up in a precalculated table the edges of the cube that are intersected by the surface.
(5) The actual surface-edge intersections are then computed by linear interpolation of the density values at the cube vertices. The intersection is divided into triangles.
(6) Central differences are used to compute a unit normal at each cube vertex which are then interpolated to each triangle vertex.
(7) Finally, the triangle vertices and normals are output.

[LorC87] points out that linear interpolation is adequate and that using higher-degree interpolants does not provide any significant improvement. Although one could determine normals from the surface triangles themselves, it turns out that the gradient method used above gives substantially better results. In order to avoid aliasing problems, the initial data must have been sampled enough to produce sufficiently small triangles in the surface.

After the marching cube algorithm finds the surface and its normals one can render the surface with a rendering technique such as ray casting and Phong shading. According to [THBP90], the algorithm has problems with thin surfaces and sharp edges and is expensive compared with other surface methods.

A 2d version of the marching cubes algorithm, called the *marching squares algorithm*, is described in [ScML98].

10.5 Exercises

Section 10.3

10.3.1 Consider the cylinder **C** with base in the x-y plane and height 7 defined by

$$(x-3)^2 + (y-5)^2 = 4.$$

Let **X** be the ray that starts at $\mathbf{p} = (-1,8,15)$ and has direction vector $\mathbf{v} = (1,-1,-2)$. Find the first intersection of **X** with **C** in the following two cases:

(1) **C** has neither a top or bottom.
(2) **C** has both a top and bottom.

10.6 Programming Projects

Section 10.2

10.2.1 A ray-tracing program

Build on programming project 9.4.1(a) and implement a **complete** ray-tracing program for a world of spheres that includes shadows, reflections, and transparencies using the Phong illumination model.

PART II

GEOMETRIC MODELING TOPICS

CHAPTER 11

Curves in Computer Graphics

Prerequisites: Basic calculus and vectors,
Section 5.2 in [AgoM05] (metric),
Sections 8.1–5 in [AgoM05] (parameterization, curve, tangent vector, manifold),
Sections 9.1–4 in [AgoM05] (arc length, curvature)

Terminology. The terms "curve" and "surface" get used in different ways. Sometimes people use them to refer to sets, other times, to functions. Although it is usually clear from the context which is meant, this ambiguity can lead to confusion because there are times when that distinction plays a role. For that reason we want to use terms as consistently as possible from the outset. An awareness of what is already known to mathematicians is also important in graphics. A lack of such and perhaps a confusion in the meaning of terms has given rise to some nonoptimal terminology in CAGD, as, for example, in the case of "G^k" curves and surfaces. Although our comments are made for curves, similar comments hold for surfaces that are discussed in the next chapter.

In this book the term "curve" by itself always means a **set**. The potential ambiguity in terminology arises because such sets are usually described via functions and one is tempted to use the same word "curve" for them. We shall allow that in specific situations that we now describe, in order to be compatible with the way one actually talks. The term "curve" preceded by appropriate adjectives will mean a **function**. In particular, the term "parametric curve" encompasses all such functions and means a continuous vector-valued **function** $p : [a,b] \to \mathbf{R}^m$ defined on some interval $[a,b]$. In this context, the expression "look at the curve" means "look at the set $p([a,b])$ that is its range." Other examples of expressions that refer to **functions** (parametric curves) and **not** sets are "cubic curve," "Bézier curve," and "B-spline curve." In addition, an expression of the form "the curve $p(u)$" is shorthand for "the parametric curve $p(u)$."

Finally, the reader will recall from calculus that there is a difference between continuity and differentiability. Graphs of real-valued functions that are not differentiable have "corners." In geometric modeling there are times when we do not want our curved objects to have corners. In analogy with the real-valued function case, a natural reaction would be to assume that if we restrict our parameterizations to be differentiable, then everything will be fine. Unfortunately, this is not so. There exist curves,

Figure 11.1. A curve with a differentiable parameterization.

such as the one shown in Figure 11.1, which have differentiable parameterizations but also have a cusp. Section 11.10 discusses how one can detect that condition. The same thing can happen in the case of surfaces. Smoothness of the function p does not imply smoothness of its range **X** (and vice versa). At the heart of this is again the distinction between a C^k **function** and a C^k manifold, which is a **set**. We shall take a closer look at this issue in Section 11.9.

11.1 Introduction to Curves and Surfaces

The first 10 chapters of this book have described the basic ideas and algorithms currently used to render geometric objects. The main topics covered were the mathematics for the graphics pipeline, clipping, drawing discrete lines, visible surface determination, and shading. In short, we know pretty much all that we need to know to render any linear polyhedra. Linear polyhedra are a too-limited domain, however, even if we were to include the conics. The time has come to talk about general "curved" objects. After all, most interesting objects are curved.

There are many aspects to the study of curves and surfaces. We shall touch on a lot of them in the next two chapters because curves and surfaces are clearly central to geometric modeling. However, the subject and the literature dealing with it are especially large and we do not want to raise overly high expectations with respect to the coverage. Readers who become interested in a really in-depth discussion of certain topics may not find enough here, especially with regard to all the various choices and some practical details related to the most efficient algorithms and implementations. They should not be surprised to find themselves reaching for one of the references for this depth. Given the breadth of our overview of geometric modeling, it was just not possible to do more here. It is hoped, however, that we shall have at least conveyed the essence of the forest of fundamental ideas if not of the trees. Specifically, at the end of the day, we expect the reader to have learned the following:

(1) the ability to describe the main curves and surfaces that one encounters in the real world of manufacturing and other areas of computer graphics,
(2) a basic understanding of what makes these objects tick,
(3) efficient and robust algorithms to compute some of their most important properties,
(4) ways to make it easier to manipulate these objects, because it is not just mathematicians that use the objects, and
(5) an appreciation of the richness of the subject.

We begin with some general comments. The first question that needs to be answered is how curved objects should be represented. A polygon could be repre-

sented by a sequence of points. This is not going to work here, although we will want finite descriptions since everything is going to be represented in a computer. The two standard representations are by parameterizations or implicitly via equations. Relatively few objects have a convenient implicit representation, with the conics being the main exception. Our emphasis will be on parametric curves and surfaces.

How are curves and surfaces defined in practice? This depends on what one is trying to do. There are three aspects to this problem though—the definition, the implementation, and the user interface. The underlying mathematical theory is obviously important but often the actual formulas have been known for a long time and the real problem is to make it easy for a user to specify them. In other words, it is often the user interface that is the driving force. Users would like to be able to create and manipulate objects easily in terms of properties relevant to their task and ones they can understand. One way to think of this is in terms of a black box that has some dials, one for each property that the user wants to adjust. A large percentage of the papers on curves and surfaces have to do with finding more intuitive and convenient ways to define and manipulate the **same** underlying mathematical curve. For example, one may want to define a piecewise cubic curve by simply specifying some points that control its shape, or by some tangent vectors, or by conditions on its curvature. It is the mathematician's task to make this possible **and** efficient.

As one looks over how curves and surfaces are used in geometric modeling one finds that the subject develops in two directions. Are we trying to model a very precise object, so that accuracy is paramount, or are we designing shapes in a more rough outline manner? For example, in the design of the wing of an airplane or the blades of a turbine one is dealing with analytical models that must be reproduced faithfully within strict tolerance limits. On the other hand, when designing an automobile body, this is more intuitive and involves aesthetics. Here the tolerances are not so strict. Many definitions of curves and surfaces are derived from data-fitting-type problems and in one sense their study deals with special cases of the following:

The general approximation problem: Given a fixed collection of functions $\varphi_1, \varphi_2, \ldots, \varphi_k$, find coefficients c_i such that

$$g(\mathbf{x}) = \sum_{i=1}^{k} c_i \varphi_i(\mathbf{x}) \tag{11.1}$$

is an approximation to some "theoretical" function $f(\mathbf{x})$. The functions φ_i are often called *primitive* or *basis* functions.

For example, if $f(x)$ is a real-valued function of a real variable, the function $\varphi_i(x)$ could be the monomial x^i, in which case we are simply looking for the polynomial $g(x)$ that best approximates $f(x)$.

Figure 11.2 depicts the environment in which we are operating. The function f is thought of as a given function in some large function space $\mathbf{X}$ over some domain $\mathbf{D}$. We are looking for a function g that comes from a certain special linear subspace $\mathbf{A}$ of $\mathbf{X}$ defined by the φ_i. The domain $\mathbf{D}$ could be quite general, so that the variable $\mathbf{x}$ is not necessarily a real number. Often $\mathbf{D}$ consists of points in $\mathbf{R}^n$. For example, the functions could be defined on a surface and then the $\mathbf{x}$s would be elements of $\mathbf{R}^2$. The function g should also be a "good" approximation. Desirable properties are:

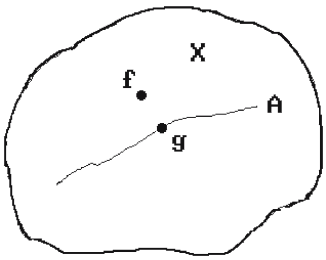


Figure 11.2. The general approximation problem.

(1) The function g should be "close" to f with respect to some appropriate metric.
(2) The coefficients c_i should be unique.

What does it mean for functions to be close? Well, that depends on the metric that is chosen for the function spaces in question (see Section 5.2 in [AgoM04]). Many metrics are possible and one needs to choose the one that is appropriate for the problem at hand. A common metric is the so-called "max" metric where d(a,b), the distance between two functions a($\mathbf{x}$) and b($\mathbf{x}$), is defined by

$$d(a,b) = \int_{\mathbf{D}} |a(\mathbf{x}) - b(\mathbf{x})| d\mathbf{x}.$$

On the other hand, the function f($\mathbf{x}$) may only be known at a finite number of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_s$, so that the best we can do is to minimize the error at the points $\mathbf{x}_j$. Before we define the well-known approach to the approximation problem in this important special case, we make two observations. First, the function g defined by equation (11.1) depends on the values c_i and so we shall use the notation $g(\mathbf{x};c_1, c_2, \ldots, c_k)$ to indicate this dependence explicitly. Second, since distances involve a square root, one simplifies things without changing the minimization problem by using the square of the distance.

Definition. The function $g(\mathbf{x};c_1,c_2, \ldots, c_k)$ that minimizes

$$E(c_1,c_2,\ldots,c_k) = \sum_{j=1}^{s} (f(\mathbf{x}_j) - g(\mathbf{x}_j;c_1,c_2,\ldots,c_k))^2$$

is called the *least squares approximation* to the function f($\mathbf{x}$).

Finding the least squares approximation involves setting the partials $\partial E/\partial c_i$ to zero and solving the resulting system of linear equations for the c_i.

When dealing with approximation problems one usually encounters other constraints. Some of these are:

(1) Interpolatory constraints:

$$g(\mathbf{x}_j) = f(\mathbf{x}_j), \quad \text{for some fixed points } \mathbf{x}_j.$$

(2) A mixture of (1) and smoothness conditions, such as conditions that the derivatives of g and f agree at the $\mathbf{x}_j$.
(3) Orthogonality constraints:

$$(f - g) \bullet \varphi_i = 0 \quad \text{for all } i.$$

(4) Variational constraints:

$$|f - g| = \min_{h \in A} |f - h|$$

(5) Intuitive shape constraints involving, for example, the curvature of the curve or surface.

A common thread, alluded to earlier, that underlies much of the discussion of parametric curves and surfaces in CAGD, is:

Although it may seem like we are discussing different parameterizations, we are often talking about **one** single function throughout and it is **not** the case that we are describing **different** functions. **The only thing that changes is how we represent the parameterization**—which control points we choose, what knots there are, if any, etc. Terms such as "Bézier curve" or "B-spline curve" simply refer to different ways of looking at the same function.

The reader will find it helpful to keep this in mind. The study of curves and surfaces in the context of CAGD largely revolves around coming up with techniques for letting the user control their shape in the manner that is most natural for achieving one's current ends. Furthermore, it involves finding ways to switch between various representations. We shall see that polynomials are the most popular functions used for parameterizations. The reason is that they are relatively simple to compute. Furthermore, they play the same role for functions as integers play for real numbers.

We are almost ready to start our study of curves, but first some terminology. Consider a parametric curve

$$p : [a, b] \to \mathbf{R}^m, \; p(u) = (p_1(u), p_2(u), \ldots, p_m(u)).$$

Note that the component functions p_i of p are just ordinary real-valued functions of a real variable.

Definition. If **all** the p_i have a certain property, then we shall say that p has that property. For example, if all the p_i are polynomials or splines (a term that will be defined shortly), then we say that p is a *polynomial* or *spline curve*, respectively. If all the p_i are linear, quadratic, or cubic polynomials, then we say that p is a *linear*, *quadratic*, or *cubic curve*, respectively.

Our plan for this chapter is to start off with some simple examples of curves and their properties that require no new knowledge past calculus. In particular, Sections

11.2 looks at some old interpolation problems beginning with two classical approaches to curve fitting. Section 11.3 translates the results on Hermite interpolation into matrices. We then discuss the popular Bézier and B-spline curves from their "classical" point of view in Sections 11.4 and 11.5.1, respectively. The material up to this point is really intended as a warm up. It is in Section 11.5.2 that we describe the modern treatment of the curves defined in the earlier sections. We introduce the easy but fundamentally new idea of multiaffine maps that is the elegant basis for most of the curves and surfaces that are used in CAGD. From a high-level standpoint, we really should have started with Section 11.5.2, but doing so without the background of the curves described in the earlier sections might have left a reader somewhat overwhelmed by the simple but yet technical nature of multiaffine maps. Furthermore, the facts themselves are worthwhile knowing. The only thing missing was a uniform framework. After defining rational B-spline and NURBS curves in Section 11.5.3, we present some algorithms in Section 11.5.4 that compute B-spline and NURBS curves efficiently. Section 11.5.5 describes some aspects of B-spline interpolation. We finish the discussion of splines in Section 11.6 with nonlinear splines. Section 11.7 defines superellipses, an interesting special class of curves. Section 11.8 discusses the subdivision problem. The problem of piecing together curves in such a way that one gets a globally smooth curve in the end is discussed briefly in Section 11.9. Section 11.10 looks at some issues related to the shape of curves. Section 11.11 defines hodographs and Section 11.12 explains the fairing of curves along with some comments on interpolation with fair curves. Section 11.13 introduces parallel transport frames on curves, which are the sometimes useful alternative to Frenet frames. Finally, switching from smooth curves to polygonal curves, Section 11.14 discusses an algorithm that can be thought of as either as a way of smoothing the latter or as defining an entirely new class of curves. Section 11.15 summarizes the main points of the chapter.

11.2 Early Historical Developments

11.2.1 Lagrange Interpolation

The simplest form of interpolation is Lagrange interpolation.

> **The Lagrange interpolation problem:** Given points (x_0,y_0), (x_1,y_1), . . . , and (x_n,y_n), find a polynomial $p(x)$, so that $p(x_i) = y_i$ for $i = 0,1, \ldots, n$.

11.2.1.1 Theorem. There is a unique such polynomial $p(x)$ of degree n called the *Lagrange polynomial*.

Proof. Define polynomials

$$L_{i,n}(x) = L_{i,n}(x; x_0, x_1, \ldots, x_n) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{11.2}$$

The polynomials $L_{i,n}$ are called the *Lagrange basis functions* with respect to x_0, $x_1, \ldots, x_n$. It is easy to check that

$$L_{i,n}(x_j) = \delta_{ij}, \tag{11.3}$$

so that the nth degree polynomial

$$p(x) = \sum_{i=0}^{n} y_i L_{i,n}(x) \tag{11.4}$$

satisfies the interpolatory conditions. This proves the existence.

To prove uniqueness assume that q(x) is another nth degree polynomial satisfying this condition, then the nth degree polynomial $h(x) = p(x) - q(x)$ has $n + 1$ roots $x_0, x_1, \ldots, x_n$. This is impossible unless h(x) is the zero polynomial, since a nontrivial polynomial of degree n can have at most n roots (Corollary E.5.4 in [AgoM05]). The theorem is proved.

Note that the Lagrange basis functions satisfy the equation

$$\sum_{i=0}^{n} L_{i,n}(x) = 1 \tag{11.5}$$

for all x. This is so because the equation is trivially true for $x = x_j$, and a nontrivial nth degree polynomial can take on the same value at most n times.

The discussion above can be applied to interpolating points in $\mathbf{R}^m$ with a parametric curve because it is simply a case of applying formula (11.4) to the m components of the points separately. In other words, given distinct real numbers u_0, $u_1, \ldots, u_n$ and points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ in $\mathbf{R}^m$, the function

$$p(u) = \sum_{i=0}^{n} L_{i,n}(u; u_0, u_1, \ldots, u_n)\, \mathbf{p}_i \tag{11.6}$$

is the **unique** polynomial curve of degree n which interpolates the points $\mathbf{p}_i$ at the values u_i. It is called the *Lagrange interpolating polynomial curve*. Because cubic curves will be of special interest to us throughout this chapter, we look at the formulas for this case in detail. To begin with

$$\begin{aligned}
L_{0,3}(u) &= \frac{(u-u_1)(u-u_2)(u-u_3)}{(u_0-u_1)(u_0-u_2)(u_0-u_3)}, \\
L_{1,3}(u) &= \frac{(u-u_0)(u-u_2)(u-u_3)}{(u_1-u_0)(u_1-u_2)(u_1-u_3)}, \\
L_{2,3}(u) &= \frac{(u-u_0)(u-u_1)(u-u_3)}{(u_2-u_0)(u_2-u_1)(u_2-u_3)}, \quad \text{and} \\
L_{3,3}(u) &= \frac{(u-u_0)(u-u_1)(u-u_2)}{(u_3-u_0)(u_3-u_1)(u_3-u_2)}.
\end{aligned} \tag{11.7}$$

If we let

$$\Pi_i = \prod_{j=0, j\neq 1}^{n} (u_i - u_j)$$

and

$$\mathbf{M} = \begin{pmatrix} \frac{1}{\Pi_0} & \frac{1}{\Pi_1} & \frac{1}{\Pi_2} & \frac{1}{\Pi_3} \\ \frac{u_1 + u_2 + u_3}{\Pi_0} & \frac{u_0 + u_2 + u_3}{\Pi_1} & \frac{u_0 + u_1 + u_3}{\Pi_2} & \frac{u_0 + u_1 + u_2}{\Pi_3} \\ \frac{u_1u_2 + u_2u_3 + u_3u_1}{\Pi_0} & \frac{u_0u_2 + u_2u_3 + u_3u_0}{\Pi_1} & \frac{u_0u_1 + u_1u_3 + u_3u_0}{\Pi_2} & \frac{u_0u_1 + u_1u_2 + u_2u_0}{\Pi_3} \\ \frac{u_1u_2u_3}{\Pi_0} & \frac{u_0u_2u_3}{\Pi_1} & \frac{u_0u_1u_3}{\Pi_2} & \frac{u_0u_1u_2}{\Pi_3} \end{pmatrix}$$

then it is straightforward to show that p(u) can be written in the following matrix form:

$$p(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \mathbf{M} \begin{pmatrix} \mathbf{p_0} \\ \mathbf{p_1} \\ \mathbf{p_2} \\ \mathbf{p_3} \end{pmatrix}. \tag{11.8}$$

Figure 11.3 shows several cubic curves p(u). We have assumed that

$$\begin{aligned} u_1 &= u_0 + |\mathbf{p}_0\mathbf{p}_1|, \\ u_2 &= u_1 + d|\mathbf{p}_1\mathbf{p}_2|, \text{ and} \\ u_3 &= u_2 + |\mathbf{p}_2\mathbf{p}_3|. \end{aligned}$$

Note how the spacing of the u_i affects the shape of the curve. The smaller the value

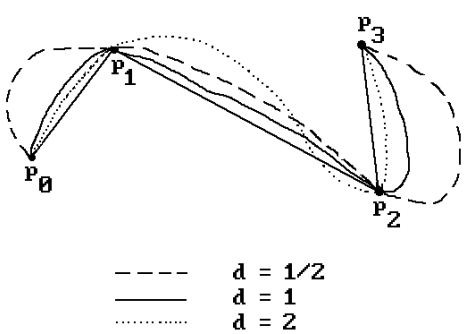


Figure 11.3. How spacing affects cubic curves.

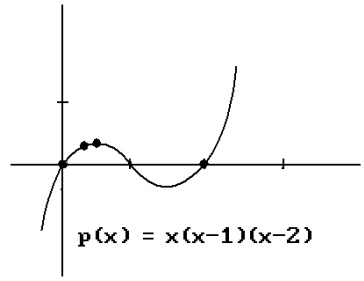


Figure 11.4. Undesired ripples in interpolating polynomials.

of d, the closer the curve gets to the segment from $\mathbf{p}_1$ to $\mathbf{p}_2$ and the worse the curve approximates the other segments.

Although Theorem 11.2.1.1 shows that one can always find an interpolating polynomial, there are some serious drawbacks to using this polynomial as the interpolating curve. First, the degree of the polynomial gets large as n gets large. This would make it computationally expensive to evaluate. Second, the curve that is generated will have ripples so that its shape may not match the shape implied by the data. For example, the unique cubic polynomial that interpolates the points (0,0), (1/3,10/27), (1/2,3/8), and (2,0) is

$$p(x) = x(x-1)(x-2),$$

but its shape has more wiggles than the shape of the polygonal curve through the points. See Figure 11.4. It is not possible to eliminate the ripples in a polynomial because an nth degree polynomial always has potentially a total of n – 1 maxima and minima (up to multiplicity).

11.2.2 Hermite Interpolation

To avoid the polynomial oscillation problem with Lagrange interpolation, one could piece together polynomials of degree two, but one would in general get corners where they meet. However, if we try using cubic polynomials, then we have enough degrees of freedom to force the polynomials to have the same slope where they meet.

11.2.2.1 Lemma. Given real numbers y_0, y_1, m_0, and m_1, there is a unique cubic polynomial p(x) so that

$$p(0) = y_0, \quad p(1) = y_1, \quad p'(0) = m_0, \quad \text{and} \quad p'(1) = m_1.$$

Proof. The general cubic

$$p(x) = a + bx + cx^2 + dx^3$$

has four degrees of freedom. Substituting our constraints gives four equations in four unknowns that have a unique solution for the a, b, c, and d.

Alternatively, one can use a matrix approach. Since

$$p'(x) = b + 2cx + 3dx^2,$$

it is easy to check that

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} \begin{pmatrix} d \\ c \\ b \\ a \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{pmatrix}. \tag{11.9}$$

The square 4×4 matrix on the left of equation (11.9) has an inverse. A straightforward computation shows that its inverse $\mathbf{M}_h$ is given by

$$\mathbf{M}_h = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{11.10}$$

Therefore,

$$\begin{pmatrix} d \\ c \\ b \\ a \end{pmatrix} = \mathbf{M}_h \begin{pmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{pmatrix}, \tag{11.11}$$

and

$$p(x) = \begin{pmatrix} x^3 & x^2 & x & 1 \end{pmatrix} \mathbf{M}_h \begin{pmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{pmatrix}. \tag{11.12}$$

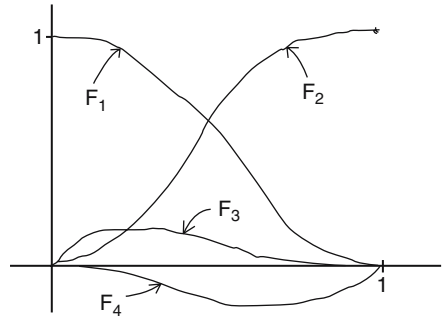
If we define polynomials F_1, F_2, F_3, and F_4 by

$$(F_1(x)\ F_2(x)\ F_3(x)\ F_4(x)) = \begin{pmatrix} x^3 & x^2 & x & 1 \end{pmatrix} \mathbf{M}_h, \tag{11.13}$$

then

$$\begin{aligned} F_1(x) &= (x-1)^2(2x+1), \\ F_2(x) &= x^2(3-2x), \\ F_3(x) &= (x-1)^2 x, \text{ and} \\ F_4(x) &= x^2(x-1). \end{aligned} \tag{11.14}$$

Figure 11.5. The Hermite basis functions.



More importantly,

$$p(x) = y_0F_1(x) + y_1F_2(x) + m_0F_3(x) + m_1F_4(x). \tag{11.15}$$

Notice that the polynomials $F_i(x)$ satisfy

$$\begin{aligned} &F_1(0) = 1, \quad F_1(1) = 0, \quad F_1'(0) = 0, \quad F_1'(1) = 0, \\ &F_2(0) = 0, \quad F_2(1) = 1, \quad F_2'(0) = 0, \quad F_2'(1) = 0, \\ &F_3(0) = 0, \quad F_3(1) = 0, \quad F_3'(0) = 1, \quad F_3'(1) = 0, \\ &F_4(0) = 0, \quad F_4(1) = 0, \quad F_4'(0) = 0, \quad F_4'(1) = 1. \end{aligned} \tag{11.16}$$

Figure 11.5 shows the graph of these functions. The existence of functions with these properties would by itself guarantee that equation (11.15) is satisfied.

Definition. The matrix $\mathbf{M}_h$ defined by equation (11.10) is called the *Hermite matrix*. The polynomials $F_i(x)$ defined by equations (11.13) and (11.14) are called the *Hermite basis functions*.

For future reference, note that the equation

$$F_1(x) + F_2(x) = 1 \tag{11.17}$$

holds for all x. Also, because the F_i are part of a more general pattern of functions similar to that of the Lagrange polynomials, we introduce the following alternate notation for them:

$$H_{0,3} = F_1, \quad H_{1,3} = F_3, \quad H_{2,3} = F_4, \quad \text{and} \quad H_{3,3} = F_2. \tag{11.18}$$

This notation will be useful in the next chapter.

Lemma 11.2.2.1 is a special case of a general interpolation problem.

The piecewise Hermite interpolation problem: Given triples (x_0,y_0,m_0), (x_1,y_1,m_1), . . . , and (x_n,y_n,m_n), find cubic polynomials $p_i(x)$, $i = 0,1, \ldots, n-1$, so that

$$\begin{aligned} p_i(x_i) &= y_i, \\ p_i'(x_i) &= m_i, \\ p_i(x_{i+1}) &= y_{i+1}, \quad \text{and} \\ p_i'(x_{i+1}) &= m_{i+1}. \end{aligned} \tag{11.19}$$

11.2.2.2 Theorem. The piecewise Hermite interpolation problem has a unique solution.

Proof. This is an easy consequence of equation (11.12). A simple change of variables in that equation, where we replace x by

$$s = \frac{x - x_i}{x_{i+1} - x_i},$$

does not quite do the trick though because the chain rule would tell us that we had the wrong slopes at the endpoints since

$$\frac{ds}{dx} = \frac{1}{x_{i+1} - x_i}.$$

We need to modify the input slopes by an appropriate factor. It is easy to check that the correct formula for the *Hermite basis function* $p_i(x)$ is

$$p_i(x) = \begin{pmatrix} s^3 & s^2 & s & 1 \end{pmatrix} \mathbf{M}_h \begin{pmatrix} y_i \\ y_{i+1} \\ m_i(x_{i+1} - x_i) \\ m_{i+1}(x_{i+1} - x_i) \end{pmatrix}.$$

We can simplify this formula. Let

$$\mathbf{M}_h(d) = \begin{pmatrix} 2/d^3 & -2/d^3 & 1/d^2 & 1/d^2 \\ -3/d^2 & 3/d^2 & -2/d & -1/d \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{11.20}$$

Then

$$p_i(x) = \begin{pmatrix} (x - x_i)^3 & (x - x_i)^2 & (x - x_i) & 1 \end{pmatrix} \mathbf{M}_h(\Delta x_i) \begin{pmatrix} y_i \\ y_{i+1} \\ m_i \\ m_{i+1} \end{pmatrix}, \tag{11.21}$$

where $\Delta x_i = x_{i+1} - x_i$. For a less ad hoc derivation of this result, see Exercise 11.2.2.1.

Equation (11.21) in the proof of Theorem 11.2.2.2 shows us that the interpolating polynomials $p_i(x)$ can be expanded in the following way:

$$p_i(x) = f_1(x)y_i + f_2(x)y_{i+1} + f_3(x)m_i + f_4(x)m_{i+1}, \tag{11.22}$$

where the functions $f_j(x)$ (which depend on i) are defined by

$$(f_1(x)\, f_2(x)\, f_3(x)\, f_4(x)) = \left((x - x_i)^3\ (x - x_i)^2\ (x - x_i)\ 1\right) \mathbf{M}_h(\Delta x_i). \tag{11.23}$$

The interesting property that we want to record here is that

$$f_1(x) + f_2(x) = 1 \tag{11.24}$$

for all x. Compare this with equation (11.17). The proof is left as Exercise 11.2.2.2.

Just like in the Lagrange case we can apply the results above to interpolating points in $\mathbf{R}^m$ with a parametric curve. In other words, given distinct real numbers u_0, $u_1, \ldots, u_n$, points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ and tangent vectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$ in $\mathbf{R}^m$, there is a unique curve $p : [u_0,u_n] \to \mathbf{R}^m$, called the *piecewise Hermite interpolating curve*, satisfying

(1) $p\ (u_i) = \mathbf{p}_i$,
(2) $p'\ (u_i) = \mathbf{v}_i$, and
(3) $p_i = p \mid [u_i,u_{i+1}]$ is a cubic polynomial.

In fact,

$$p_i(u) = \left((u - u_i)^3\ \ (u - u_i)^2\ \ (u - u_i)\ 1\right)\mathbf{M}_h(\Delta u_i)\begin{pmatrix} \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{v}_i \\ \mathbf{v}_{i+1} \end{pmatrix}, \tag{11.25}$$

where $\Delta u_i = u_{i+1} - u_i$. Using equation (11.23) we can also express $p_i(u)$ in the form

$$p_i(u) = f_1(u)\mathbf{p}_i + f_2(u)\mathbf{p}_{i+1} + f_3(u)\mathbf{v}_i + f_4(u)\mathbf{v}_{i+1}, \tag{11.26}$$

where the functions $f_j(u)$ are defined as in equation (11.22).

The function p(u) is clearly differentiable by construction. However, this does not completely solve the problem with which we began, because although we now have an interpolating curve of low degree without "corners," we assumed that the tangent vectors $\mathbf{v}_i$ were given to us. Such an assumption may not be convenient and it is in fact unnecessary as we shall see in the next section.

Finally, the answers to the Lagrange and Hermite interpolation problems above show a pattern that ought to be noted. In each case, the problem was solved in an elegant way by finding *basis functions* $a_i(x)$ such that $a_i(x_j) = \delta_{ij}$ and similar functions for derivatives. These functions constructively isolated the effect of each "control datum" so that it occurred only once as an explicit parameter in the solution. This is

called the *blending function* approach to interpolation and approximation. We shall see more examples of this in the future. We want to make one last important observation in this context.

Let $\mathbf{A} \subseteq \mathbf{R}^k$ and consider a function $p : \mathbf{A} \to \mathbf{R}^m$ of the form

$$p(u) = f_1(u)\mathbf{p}_1 + \ldots + f_s(u)\mathbf{p}_s + g_1(u)\mathbf{v}_1 + \ldots + g_t(u)\mathbf{v}_t, \tag{11.27}$$

where $\mathbf{p}_i$, $\mathbf{v}_j \in \mathbf{R}^m$, the $\mathbf{p}_i$ are "points," the $\mathbf{v}_j$ are "vectors," and the $f_i(u)$ and $g_i(u)$ are real-valued functions. The difference between a "point" and a "vector" here has to do with how they transform. We assume that an affine map T of $\mathbf{R}^m$ sends a "vector" $\mathbf{v}$ (thought of as a directed segment from the origin to the **point** $\mathbf{v}$) to the "vector" $T(\mathbf{0})T(\mathbf{v})$. Given an arbitrary affine map T of $\mathbf{R}^m$, express T in the form

$$T(\mathbf{q}) = M(\mathbf{q}) + \mathbf{q}_0,$$

where M is a linear transformation (see Chapter 2 in [AgoM05], in particular, Theorem 2.5.8). Let $\mathbf{X} = p(\mathbf{A}) \subseteq \mathbf{R}^m$. The question we want to ask is how one can compute the tranformed set $\mathbf{Y} = T(\mathbf{X})$. An arbitrary point p(u) of $\mathbf{X}$ gets mapped by T to

$$f_1(u)M(\mathbf{p}_1) + \ldots + f_s(u)M(\mathbf{p}_s) + g_1(u)M(\mathbf{v}_1) + \ldots + g_t(u)M(\mathbf{v}_t) + \mathbf{q}_0. \tag{11.28}$$

On the other hand, if we simply replaced the points $\mathbf{p}_i$ and vectors $\mathbf{v}_i$ in equation (11.27) by their transformed values, we would get

$$\begin{aligned} &f_1(u)(M(\mathbf{p}_1) + \mathbf{q}_0) + \ldots + f_s(u)(M(\mathbf{p}_s) + \mathbf{q}_0) + g_1(u)M(\mathbf{v}_1) + \ldots + g_t(u)M(\mathbf{v}_t) = \\ &\quad f_1(u)M(\mathbf{p}_1) + \ldots + f_s(u)M(\mathbf{p}_s) + \left(\sum_{j=1}^{s} f_j(u)\right)\mathbf{q}_0 + g_1(u)M(\mathbf{v}_1) + \ldots + g_t(u)M(\mathbf{v}_t). \end{aligned} \tag{11.29}$$

Definition. The function p(u) defined by equation (11.27) is said to be *affinely invariant* if expressions (11.28) and (11.29) define the same point.

11.2.2.3 Theorem. The function p(u) defined by an equation of the form (11.27) is affinely invariant if and only if

$$\sum_{j=1}^{s} f_j(u) = 1.$$

Proof. This is easy. One simply equates expressions (11.28) and (11.29).

11.2.2.4 Corollary. The Lagrange and Hermite interpolating curves are affinely invariant.

Proof. This follows from Theorem 11.2.2.3 and equations (11.5) and (11.24).

The importance of being affinely invariant lies in the fact that in order to move a curve (or the set traced out by an arbitrary parameterization defined by equation (11.27)) we do not have to move every point on it (which would not be very feasible even in the case of computer graphics where the screen consists of only a finite

number of pixels) but can simply recompute the curve using the moved "control data."

No real constraints were placed on the basis functions above. However, in later sections on Bézier curves and B-splines we shall see the usefulness of the following properties:

(1) The functions should be nonnegative.
(2) The functions corresponding to point data should sum to 1.
(3) The functions should have compact support.

Properties (1) and (2) would imply that the curve lies in the convex hull of its control points and property (3) essentially means that if a control point is moved only the curve near that point changes. Properties (1)–(3) basically mean that the functions form a partition of unity.

11.2.3 Spline Interpolation

The interpolation problems described in the last section and the functions that solve them can be generalized.

Definition. A *spline of degree m and order m + 1* is a function $S : [a,b] \to \mathbf{R}$ for which there exist real numbers x_i, $i = 0, \ldots, n$, with $a = x_0 \le x_1 \le \ldots \le x_n = b$, so that

(1) S is a polynomial of degree $\le m$ on $[x_i,x_{i+1}]$, for $i = 0, \ldots, n - 1$, and
(2) S is a C^{m-1} function.

The x_i are called *knots* and $(x_0,x_1, \ldots ,x_n)$ is called the *knot vector* of *length* $n + 1$ for the spline. The intervals $[x_i,x_{i+1}]$ are called *spans*. If a knot x_i satisfies $x_{i-1} < x_i = x_{i+1} = \ldots = x_{i+d-1} < x_{i+d}$ ($x_{-1} = -\infty$ and $x_{n+1} = +\infty$), then x_i is said to be a knot of *multiplicity* d. S is called a *linear*, *quadratic*, or *cubic* spline if it has degree 1, 2, or 3, respectively.

Note. One allows $a = -\infty$ and/or $b = +\infty$ in the definition of a spline. Only **finite** x_i are called knots, however.

If one had to list the key terms that should always be intimately associated with the concept of spline they would be "piecewise polynomial function," "knots," and "differentiability." Note that splines are **more** than just piecewise polynomials because they satisfy a **global** differentiability condition. The piecewise Hermite interpolation function described in the last section was not smooth enough to be called a cubic spline.

The physical definition of a spline. A *spline* is a thin metal or wooden strip that is bent elastically so as to pass through certain points of constraint.

Physical splines have been used for ages. For example, in the construction of ships' hulls, the hull was modeled at full or nearly full size on a wooden floor in the "mold's loft." This task, called "lofting," was carried out by skilled "loftsmen" using such physical splines. When one tries to determine a mathematical description of the curves generated by physical splines, one discovers something very interesting. Physics tells

us that the strip will assume a shape that minimizes the strain. The equation for this minimum energy problem is difficult to solve directly, however, an approximation to it can be solved and leads to a solution that is a cubic spline. We shall explain this a little more in Section 11.6. At any rate, it is this ability of cubic splines to model curves from physical splines, plus the fact that their low degree makes them easy to compute, that makes them the most popular spline by far.

The spline interpolation problem: Given an integer k and real numbers x_i and y_i, $i = 0, \ldots, n$, with $x_0 < x_1 < \ldots < x_n$, find a spline $g(x)$ of order k so that the x_i are the knots for g and $g(x_i) = y_i$.

11.2.3.1 Theorem. The spline interpolation problem has a solution.

Proof. For a general solution see [BaBB87] or [deBo78]. One can also prove this using B-splines, which are discussed later. See [RogA90]. Here we shall only show the existence of a solution in the important special case of a cubic spline. We rephrase the problem as follows: Given points (x_0,y_0), (x_1,y_1), . . . , and (x_n,y_n), find cubic polynomials $p_i(x)$, so that for $i = 0,1, \ldots, n-1$,

$$\begin{aligned} p_i(x_i) &= y_i, \\ p_i(x_{i+1}) &= y_{i+1}, \end{aligned}$$

and for $i = 1,2, \ldots, n-1$,

$$\begin{aligned} p_i'(x_i) &= p_{i-1}'(x_i), \\ p_i''(x_i) &= p_{i-1}''(x_i). \end{aligned}$$

In this situation we have 4n degrees of freedom and only 4n – 2 constraints. The two extra degrees of freedom can be handled in several ways depending on how we choose to specify m_0 and m_n, the slope at the beginning and end of the spline, respectively. We mention four approaches, but there are others.

End condition choices for interpolating splines:

(1) (*Clamped end condition*) We can specify the slopes m_0 and m_n explicitly.

(2) (*Bessel end condition*) We can let m_0 and m_n be the end slope of the interpolating parabola for the first, respectively, last three data points.

(3) (*Natural end condition*) We can require that the second derivative of the spline vanishes at the ends. This amounts to requiring zero curvature of the spline at the ends and is closer to what happens in the case of a physical spline. The spline will act like a straight line near its endpoints. This type of spline is called a *natural spline*.

(4) (*Periodic end condition*) We require that the value of the spline and the value of its first and second derivative are the same at both endpoints. This is of interest mainly in the context of closed spline curves.

In the first approach we are obligated to specify the end conditions ourselves, whereas in the other approaches it is done automatically for us. No matter what choice we

make it will definitely influence the curve. As an example, consider the first three approaches in the context of a uniformly spaced spline curve that interpolates points on an arc of a circle (we shall show how our results about spline **functions** applies to spline **curves** shortly). One would like the curvature of the curve to be approximately constant since that holds for the circle. What one finds is the following (see [Fari97]): With the correct choice of start and end tangents, the clamped end condition approach has the best curvatures, the Bessel end condition approach is the next best, and the natural end condition approach is the worst because it forces the biggest deviations from constant curvature near the endpoints.

For more on spline interpolation see [Beac91], [Fari97], or [RogA90]. Here we only sketch the solution to the spline interpolation problem for clamped end conditions.

Let us assume for the moment that we know the slopes $m_i = p_i'(x_i)$ for $i = 1,2, \ldots, n-1$. Assume further that we also know the slopes m_0 and m_n at x_0 and x_n, respectively. Because the slopes are known, equation (11.21) defines the polynomials $p_i(x)$. A simple computation using equation (11.21) shows that

$$p_i''(x) = 6\frac{\Delta x_i(m_i + m_{i+1}) - 2\Delta y_i}{\Delta x_i^3}(x - x_i) + 2\frac{3\Delta y_i - \Delta x_i(2m_i + m_{i+1})}{\Delta x_i^2},$$

where $\Delta x_i = x_{i+1} - x_i$, and $\Delta y_i = y_{i+1} - y_i$. Setting $p_i''(x_i)$ equal to $p_{i-1}''(x_i)$ leads to the equation

$$\delta_{i-1}m_{i-1} + 2(\delta_{i-1} + \delta_i)m_i + \delta_i m_{i+1} = 3\left(\delta_{i-1}^2\Delta y_{i-1} + \delta_i^2\Delta y_i\right),$$

for $i = 0,1, \ldots, n-1$, where $\delta_i = 1/\Delta x_i$. The matrix form of this system of equations is

$$S_n\begin{pmatrix} m_1 \\ m_2 \\ \cdot \\ \cdot \\ m_{n-2} \\ m_{n-1} \end{pmatrix} = \begin{pmatrix} 3\left(\delta_0^2\Delta y_0 + \delta_1^2\Delta y_1\right) - \delta_0 m_0 \\ 3\left(\delta_1^2\Delta y_1 + \delta_2^2\Delta y_2\right) \\ \cdot \\ \cdot \\ 3\left(\delta_{n-3}^2\Delta y_{n-3} + \delta_{n-2}^2\Delta y_{n-2}\right) \\ 3\left(\delta_{n-2}^2\Delta y_{n-2} + \delta_{n-1}^2\Delta y_{n-1}\right) - \delta_{n-1}m_n \end{pmatrix}, \tag{11.30}$$

where S_n is the $(n-1) \times (n-1)$ matrix

$$S_n = \begin{pmatrix} 2(\delta_0 + \delta_1) & \delta_1 & 0 & \ldots & 0 & 0 \\ \delta_1 & 2(\delta_1 + \delta_2) & \delta_2 & \ldots & 0 & 0 \\ 0 & \delta_2 & 2(\delta_2 + \delta_3) & \ldots & 0 & 0 \\ \cdot & \cdot & \cdot & \ldots & \cdot & \cdot \\ \cdot & \cdot & \cdot & \ldots & \cdot & \cdot \\ 0 & 0 & 0 & \ldots & \delta_{n-3} & 0 \\ 0 & 0 & 0 & \ldots & 2(\delta_{n-3} + \delta_{n-2}) & \delta_{n-2} \\ 0 & 0 & 0 & \ldots & \delta_{n-2} & 2(\delta_{n-2} + \delta_{n-1}) \end{pmatrix}. \tag{11.31}$$

The tridiagonal symmetric matrix S_n has all positive entries and is **diagonally dominant,** which implies that it has an inverse. In other words, there is a solution to the cubic spline interpolation problem and this solution is unique. The tridiagonal nature of the matrix means that the system of equations can be solved very efficiently. One needs only one forward substitution sweep (row operations starting from the top, which eliminate the elements below the diagonal and change the diagonal elements to 1) and then one backward substitution sweep starting from the bottom. See [ConD72].

Finally, let us translate the above results to interpolating points in $\mathbf{R}^m$ with a parametric curve. Suppose that we are given distinct real numbers $u_0, u_1, \ldots, u_n$ and points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ in $\mathbf{R}^m$, then there is a unique cubic spline $p : [u_0,u_n] \to \mathbf{R}^m$ satisfying $p(u_i) = \mathbf{p}_i$. The individual cubic curves that make up $p(u)$ are defined by equation (11.25). All that is needed is to find the tangent vectors $\mathbf{v}_i$ at the points $p(u_i)$. Let

$$\Delta\mathbf{p}_i = \mathbf{p}_{i+1} - \mathbf{p}_i \quad \text{and} \quad \delta_i = \frac{1}{u_{i+1} - u_i}.$$

Then equation (11.30) becomes

$$S_n \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \cdot \\ \cdot \\ \mathbf{v}_{n-2} \\ \mathbf{v}_{n-1} \end{pmatrix} = \begin{pmatrix} 3(\delta_0^2\Delta\mathbf{p}_0 + \delta_1^2\Delta\mathbf{p}_1) - \delta_0\mathbf{v}_0 \\ 3(\delta_1^2\Delta\mathbf{p}_1 + \delta_2^2\Delta\mathbf{p}_2) \\ \cdot \\ \cdot \\ 3(\delta_{n-3}^2\Delta\mathbf{p}_{n-3} + \delta_{n-2}^2\Delta\mathbf{p}_{n-2}) \\ 3(\delta_{n-2}^2\Delta\mathbf{p}_{n-2} + \delta_{n-1}^2\Delta\mathbf{p}_{n-1}) - \delta_{n-1}\mathbf{v}_n \end{pmatrix} \tag{11.32}$$

The $\mathbf{v}_i$ are solved for using this equation. The uniform spline case where $u_{i+1} - u_i = 1$ is of special interest. In that case we need to solve the following system:

$$\begin{pmatrix} 4 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 1 & 4 & 1 & \ldots & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & \ldots & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \ddots & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \ldots & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \ldots & 1 & 4 & 1 \\ 0 & 0 & 0 & \ldots & 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{v}_{n-2} \\ \mathbf{v}_{n-1} \end{pmatrix} = \begin{pmatrix} 3(\mathbf{p}_2 - \mathbf{p}_0) - \mathbf{v}_0 \\ 3(\mathbf{p}_3 - \mathbf{p}_1) \\ \cdot \\ \cdot \\ \cdot \\ 3(\mathbf{p}_{n-1} - \mathbf{p}_{n-3}) \\ 3(\mathbf{p}_n - \mathbf{p}_{n-2}) - \mathbf{v}_n \end{pmatrix} \tag{11.33}$$

Section 11.5.5 will look at another solution to the spline interpolation problem.

11.3 Cubic Curves

Cubic curves are the most popular in graphics because, as indicated earlier, the degree is high enough for them to be able to satisfy the typical constraints one wants and yet

they are easy to compute. For that reason it is worthwhile to collect together in one place some facts about them. The emphasis in this section will be on their basic matrix representation and how it can be used to analyze the curves. Additional matrix representations will be encountered in the sections on Bézier curves and B-splines. We should point out though that matrix representations are not always the fastest or the most numerically stable representations. See Section 11.15.

To begin with, it is easy to see, by collecting together terms with the same power of u, that every cubic curve in $\mathbf{R}^m$ can be written in the form

$$p(u) = \mathbf{a}_0 + \mathbf{a}_1 u + \mathbf{a}_2 u^2 + \mathbf{a}_3 u^3, \tag{11.34}$$

where the $\mathbf{a}_i$ are vectors in $\mathbf{R}^m$. For example,

$$p(u) = \left(2u^2 + 3\,u^3 - 3u + 7, 5u^3 + u^2 + u + 2\right)$$

can be written as

$$p(u) = (3,7,2) + (0,-3,1)u + (2,0,1)u^2 + (0,1,5)u^3.$$

Suppose that one wants to use a polynomial as in (11.34) to design curves. The $\mathbf{a}_i$ are then the unknowns and in this representation of the function they are what has to be determined. However, the same function can be specified in many different ways. The most convenient way to specify the parameterization depends on what one is doing. Specifying the $\mathbf{a}_i$ directly is usually the least convenient. Hermite interpolation was basically a case where one wanted to specify the curve by means of its endpoints and its tangent vectors at those points. This is a more geometric approach but there are others. Given that curves can be represented in different ways it is desirable to be able to switch between representations. We show in this and later sections that matrices can be used effectively for this task.

Notation. We shall abbreviate p(c) and p′(c) to $\mathbf{p}_c$ and $\mathbf{p}_c^u$, respectively.

From now on, unless stated otherwise, the domain of our cubic curve is assumed to be [0,1]. This assumption leads to simplified formulas but the results in this case translate easily into corresponding results for other domains. See the comments at the end of this section.

Define matrices

$$\mathbf{U} = \left(u^3\ u^2\ u\ 1\right), \quad \mathbf{A} = \begin{pmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \\ \mathbf{a}_1 \\ \mathbf{a}_0 \end{pmatrix}, \quad \text{and} \quad \mathbf{B}_h = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_0^u \\ \mathbf{p}_1^u \end{pmatrix}. \tag{11.35}$$

Definition. The vectors $\mathbf{a}_i$ are called the *algebraic coefficients* of the cubic curve p(u). The elements of $\mathbf{B}_h$ are called its *geometric* or *Hermite coefficients*.

Note that

$$p(u) = \mathbf{UA} \quad \text{and} \quad p'(u) = \mathbf{U}'\mathbf{A} = \begin{pmatrix} 3u^2 & 2u & 1 & 0 \end{pmatrix}\mathbf{A}.$$

Furthermore, in analogy with equation (11.9),

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix}\mathbf{A} = \mathbf{B}_h.$$

Recall from (11.10) that the Hermite matrix $\mathbf{M}_h$, which is the inverse of the 4×4 matrix on the left, is given by

$$\mathbf{M}_h = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{11.36}$$

By definition, $\mathbf{A} = \mathbf{M}_h\mathbf{B}_h$. We have the following matrix equations:

$$\begin{aligned} p(u) &= \mathbf{UA} \\ &= \mathbf{UM}_h\mathbf{B}_h \\ &= \mathbf{FB}_h, \end{aligned} \tag{11.37}$$

where

$$\mathbf{F} = \mathbf{UM}_h = (F_1(u)\ F_2(u)\ F_3(u)\ F_4(u)).$$

The functions $F_i(u)$ are just the Hermite basis functions defined earlier in (11.14). They will now also be called *blending functions* because they blend the geometric coefficients into p. Equations (11.37) show that a curve can be manipulated by changing either its algebraic or geometric coefficients.

Derivatives of p can also be computed in matrix form:

$$p'(u) = (\mathbf{UM}_b)'\mathbf{B}_h = \mathbf{UM}^u\mathbf{B}_h = \mathbf{F}'\mathbf{B}_h, \tag{11.38}$$

where

$$\mathbf{M}^u = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{11.39}$$

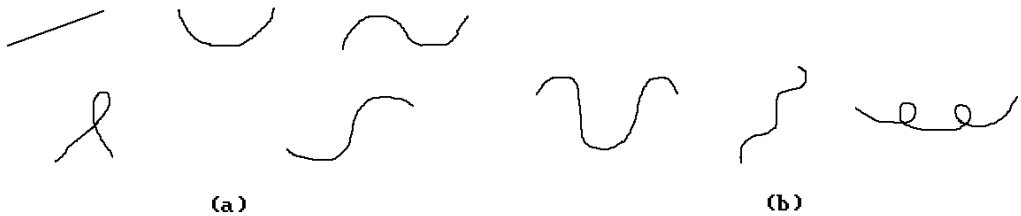


Figure 11.6. Possible and impossible cubic curves.

and

$$p''(u) = (\mathbf{UM}_b)''\mathbf{B}_h = \mathbf{UM}^{uu}\mathbf{B}_h = \mathbf{F}''\mathbf{B}_h, \tag{11.40}$$

where

$$\mathbf{M}^{uu} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 12 & -12 & 6 & 6 \\ -6 & 6 & -4 & -2 \end{pmatrix}. \tag{11.41}$$

Before moving on to other matrix descriptions of a cubic curve we pause to show how just the geometric matrix by itself already tells us a lot about its shape. A more thorough discussion of the shape of curves can be found in Section 11.10. First of all, we need to realize that only a limited number of shapes are possible here because cubic polynomials have the property that their slope can change sign at most twice and they can have only one inflection point. For example, Figure 11.6(a) shows possible shapes and Figure 11.6(b) shows impossible ones. Secondly, although there may be many ways to specify a cubic curve, it is uniquely defined once one knows its geometric coefficients. To put it another way, if one can come up with a cubic curve that has the same geometric coefficients as some other cubic curve, then this will be the **same** curve as the other one, no matter how the other one was defined. Having said that we shall now show how looking at the x-, y-, and z-coordinates of a cubic curve separately and then combining the analysis can tell us a lot about its shape and whether it has loops or cusps.

11.3.1 Example. Consider the following four geometric coefficient matrices $\mathbf{B}_h$:

$$\text{(a)} \begin{pmatrix} 1 & 3 & 1 \\ 7 & 3 & 1 \\ 6 & 0 & 0 \\ 6 & 0 & 0 \end{pmatrix} \quad \text{(b)} \begin{pmatrix} 1 & 3 & 1 \\ 7 & 3 & 1 \\ 6 & 0 & 10 \\ 6 & 0 & -10 \end{pmatrix} \quad \text{(c)} \begin{pmatrix} 1 & 3 & 1 \\ 7 & 3 & 1 \\ 20 & 0 & 40 \\ 20 & 0 & -40 \end{pmatrix} \quad \text{(d)} \begin{pmatrix} 1 & 3 & 1 \\ 7 & 3 & 1 \\ 6 & 0 & 10 \\ 6 & 0 & 10 \end{pmatrix}$$

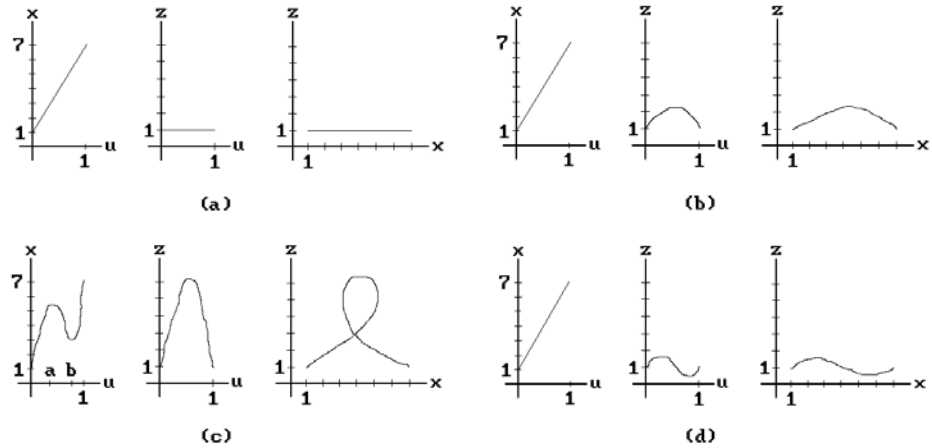


Figure 11.7. The cubic curves for Example 11.3.1(a)–(d).

The problem is to sketch the corresponding curves p(u) without actually computing the polynomials via formula (11.37).

Solution. Sketches of the curves are shown in Figure 11.7, but we want to give a qualitative explanation for why they look like they do.

First of all, note that in all four cases the curve starts at (1,3,1) and ends at (7,3,1). Let

$$p(u) = (x(u), y(u), z(u)).$$

Then $y(0) = y(1) = 3$ and $y'(0) = y'(1) = 0$. It follows that $y(u) = 3$ for all u and that each curve lies in the plane $y = 3$. To analyze each curve we need only find its projection in the x-z plane. This will be done by analyzing x(u) and z(u).

Curve (a). The tangent vector at the beginning and at the end is (6,0,0). Since the straight line

$$p(u) = (1,3,1) + u(6,0,0)$$

from (1,3,1) to (7,3,1) has the same tangent vector, this is the (only) solution. See Figure 11.7(a).

Curve (b). See Figure 11.7(b). The function x(u) is just a linear function since it has the right slope, namely, 6, at both ends. Because the slope of z(u) is 10 at 0 and –10

at 1, the shape of the graph of z(u) can be realized by a parabola as shown in the figure. The function z(u) achieves its maximum value at u = 0.5. (By uniqueness, since a parabola is able to satisfy the given data, it **must** be the curve. Actually solving for z(u) would give us

$$z(u) = -5(u-0.5)^2 + 2.25,$$

but we do not need this precise formula.) It follows that as u moves from 0 to 1, x(u) increases steadily from 1 to 6 in a uniform way and the function z(u) starts at 1 and increases until u = 0.5, then it decreases back to 1. This leads to the indicated sketch of the projection of p(u) to the x-z plane.

Curve (c). See Figure 11.7(c). The graph of x(u) needs to have the shape shown since its slope is 20 at both 0 and 1. It is a cubic. The only fact that we need to believe that requires perhaps a little extra experience with functions is that the local maximum and minimum occur at some values a and b, respectively, where $0 < a < 0.5 < b < 1$. The function z(u) has slope 40 at 0 and –40 at 1. Therefore, since this can again be realized by a parabola which takes on its maximum value at u = 0.5, it is that parabola. Its actual formula happens to be

$$z(u) = -20(u-0.5)^2 + 6,$$

but this is again not important for what we are doing. All that we need to know is that as u moves from 0 to a, the x-coordinate of p(u) is increasing and so is the z-coordinate. As we move from a to 0.5, x is decreasing, but z is still increasing. Moving from 0.5 to b, both x and z are decreasing. Finally, as u moves from b to 1, x is increasing, but z is decreasing. The reader should check that the x- and z-coordinates of the self-intersecting loop shown on the right in Figure 11.7(c) behave in the same way as one moves from the left to the right endpoint of that curve.

Curve (d). See Figure 11.7(d). The graph of x(u) is again a straight line and the shape of the graph of z(u) is forced by its slope of 10 at both ends to be the cubic as shown. The rest of the argument is, like for curve (c), based on an analysis of the regions where x and z are increasing and decreasing. This finishes Example 11.3.1.

Example 11.3.1 and others such as Exercise 11.3.2 should begin to give the reader a feel for how changing $\mathbf{p}_0^u$ and $\mathbf{p}_1^u$ affects a curve.

One other useful matrix form is the one for a cubic curve that interpolates four points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. Although equation (11.8) already described a general solution for this problem, it is worthwhile to state the special uniform case explicitly. That is the case where the u_i are chosen to be the numbers 0, 1/3, 2/3, and 1, in other words, $\mathbf{p}_i = p(i/3)$. Let

$$\mathbf{P} = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix}.$$

Then the so-called four-point matrix form of a cubic curve is

$$\mathrm{p}(\mathrm{u}) = \mathbf{U}\mathbf{M}_4\mathbf{P}, \tag{11.42}$$

where the *four-point matrix* $\mathbf{M}_4$ is defined by

$$\mathbf{M}_4 = \begin{pmatrix} -\frac{9}{2} & \frac{27}{2} & -\frac{27}{2} & \frac{9}{2} \\ 9 & -\frac{45}{2} & 18 & -\frac{9}{2} \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}. \tag{11.43}$$

The geometric matrix $\mathbf{B}_\mathrm{h}$ and $\mathbf{P}$ are related by the equation $\mathbf{B}_\mathrm{h} = \mathbf{L}\mathbf{P}$, where

$$\mathbf{L} = \mathbf{M}_\mathrm{h}^{-1}\mathbf{M}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ -1 & \frac{9}{2} & -9 & \frac{11}{2} \end{pmatrix}. \tag{11.44}$$

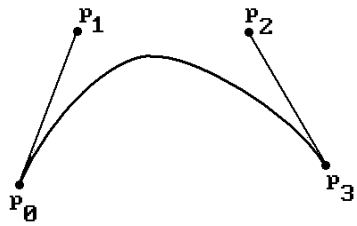
The discussion up to here has centered on cubic curves with domain [0,1]. What happens in the case of a different domain [a,b]? One can give a similar analysis, except that the "geometric coefficients" for such a cubic curve would have to be based on the values and tangents at a and b, rather than at 0 and 1. Other than that one could proceed pretty much as before. Note that the Hermite matrix M_h can no longer be used, but there would be a matrix that plays the same role but based on a and b. Exercise 11.3.4 asks the reader to work out one example of such a change.

11.4 Bézier Curves

This section and the next will deal with curves that are defined by control points but do not interpolate them in general. We shall return to the interpolation problem in Section 11.5.5.

Although the geometric coefficients approach to defining curves is a big improvement over having to specify the algebraic coefficients, specifying tangent vectors in an interactive computer graphics environment is still somewhat technical. A better way allows a user to specify these vectors implicitly by simply picking points that suggest the desired shape of the curve at the same time. Figure 11.8 shows a cubic curve p(u) which starts at $\mathbf{p}_0$ and ends at $\mathbf{p}_3$. It is very easy to specify the tangent lines

Figure 11.8. A Bézier curve and its data.



to the curve at these points by graphically picking any points $\mathbf{p}_1$ and $\mathbf{p}_2$ along these lines. Then

$$\begin{aligned} p'(0) &= \alpha \mathbf{p}_0\mathbf{p}_1, \\ p'(1) &= \beta \mathbf{p}_2\mathbf{p}_3, \end{aligned} \tag{11.45}$$

for some α and β. Now let us turn this construction around. Rather than starting with the curve p(u), let us start with the points $\mathbf{p}_i$ and ask what curve p(u) these points and equations (11.45) define. We could of course let α and β be any fixed positive real numbers, but for reasons that will become apparent shortly, we fix $\alpha = \beta = 3$.

By definition, the Hermite matrix $\mathbf{B}_h$ for the cubic curve p(u) is just $\mathbf{M}_{hb}\mathbf{B}_b$, where

$$\mathbf{M}_{hb} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{B}_b = \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix}.$$

The matrix $\mathbf{B}_b$ defines the geometric data for the Bézier curve. It follows that

$$\begin{aligned} p(u) &= \mathbf{U}\mathbf{M}_h\mathbf{B}_h = \mathbf{U}\mathbf{M}_h\mathbf{M}_{hb}\mathbf{B}_b = \mathbf{U}\mathbf{M}_b\mathbf{B}_b \\ &= \mathbf{F}_b\mathbf{B}_b, \end{aligned} \tag{11.46}$$

where $\mathbf{F}_b = \mathbf{U}\mathbf{M}_b$ and

$$\mathbf{M}_b = \mathbf{M}_h\mathbf{M}_{hb} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{11.47}$$

Definition. The elements of the matrix $\mathbf{B}_b$, namely, the points $\mathbf{p}_i$, are called the *Bézier coefficients* of the curve p(u). The matrix $\mathbf{M}_b$ is called the *Bézier matrix*.

Multiplying the matrices in equation (11.44) leads to the following formula for p(u):

$$p(u) = (1-u)^3\mathbf{p}_0 + 3u(1-u)^2\mathbf{p}_1 + 3u^2(1-u)\mathbf{p}_2 + u^3\mathbf{p}_3. \tag{11.48}$$

This cubic curve is called the *cubic Bézier curve* based on the points $\mathbf{p}_i$. Notice how the coefficients add up to 1, which tells us that the curve lies in the convex hull of the points $\mathbf{p}_i$. This is one reason for our particular choice of α and β in equation (11.45).

We have a new way to design curves. Picking four points produces a curve that starts at the first and ends at the last and has a tangent vector at the beginning and end that is parallel to the lines between the first and last two points, respectively. We manipulate the curve by simply moving one or more of these "control points."

Next, we shall generalize this construction from four points to an arbitrary number. We want to be able to define a curve by outlining its desired shape with some points. We shall describe two approaches to defining general Bézier curves. They can be defined by

(1) starting with the Bernstein polynomial approximation of continuous functions and continuing with a "brute" force approach to derive various properties, or
(2) using a more geometric "multiaffine" approach.

We begin with the first approach. Section 11.5.2 will deal with the second.

Definition. Let $f : [0,1] \to \mathbf{R}^m$ be a continuous function. Define

$$F_n(f)(u) = \sum_{i=0}^{n} f\left(\frac{i}{n}\right) B_{i,n}(u), \tag{11.49}$$

where

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}. \tag{11.50}$$

The polynomial function $F_n(f)(u)$ is called the *Bernstein polynomial approximation of degree n* to f.

Bernstein used these polynomials to give a constructive proof of the Weierstrass approximation theorem, which showed that every continuous function could be approximated by a polynomial. One can show that the Bernstein polynomials converge **uniformly** to f, but they converge very slowly and so they are not normally used for that in mathematics since there are better ways to approximate functions by polynomials. On the other hand, they lead to a representation for curves that is good for interactive curve design.

Definition. Given a sequence of points $\mathbf{p}_i$ in $\mathbf{R}^m$, $i = 0,1, \ldots, n$, define the *Bézier curve* $p(u)$, $u \in [0,1]$, by

$$p(u) = \sum_{i=0}^{n} B_{i,n}(u)\mathbf{p}_i. \tag{11.51}$$

The points $\mathbf{p}_i$ are called the *Bézier* or *control points* for p(u) and the polygonal curve defined by them is said to form the *Bézier* or *characteristic* or *control polygon* for p(u). The functions $B_{i,n}(u)$ are called the *Bézier basis functions*.

Note. P. Bézier, who invented the Bézier curves, originally used coefficient functions in (11.51) that were slightly different from but related to the Bernstein polynomials. See [Fari97]. One does not use his original functions anymore because the Bernstein polynomials are easier to use.

Bézier curves have a number of nice properties. The first of these is that they lie in the convex hull of their characteristic polygon. This follows from the following two facts and Theorem 1.7.2 in [AgoM05]:

(1) $B_{i,n}(u) \geq 0$.
(2) $\sum_{i=0}^{n} B_{i,n}(u) = 1$.

Equation (2) holds because the $B_{i,n}(u)$ are just the terms that one gets in the binomial expansion of the right hand side of the equation

$$1 = 1^n = ((1-u)+u)^n.$$

It is also easy to check that the Bézier curve starts at the first control point and ends on the last, that is, $p(0) = \mathbf{p}_0$ and $p(1) = \mathbf{p}_n$.

Bézier curves are also *symmetric*. What this means is that if we list the control points of the curve in reverse order we get the same curve, although it will be traversed in the opposite direction. This follows from the easily checked fact that

$$B_{i,n}(u) = B_{n-i,n}(1-u), \tag{11.52}$$

so that

$$\sum_{i=0}^{n} B_{i,n}(u)\mathbf{p}_i = \sum_{i=0}^{n} B_{i,n}(1-u)\mathbf{p}_{n-i}.$$

Next, we show that Bézier curves have a simple recursive definition by rewriting the formula that defines them. Using the identity for binomial coefficients

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1}, \tag{11.53}$$

we get

$$
\begin{aligned}
p(u) &= \sum_{i=0}^{n} B_{i,n}(u)\mathbf{p}_i \\
&= (1-u)^n \mathbf{p}_0 + \sum_{i=1}^{n-1} \binom{n}{i} u^i (1-u)^{n-i} \mathbf{p}_i + u^n \mathbf{p}_n \\
&= (1-u)^n \mathbf{p}_0 + \sum_{i=1}^{n-1} \binom{n-1}{i} u^i (1-u)^{n-i} \mathbf{p}_i + \sum_{i=1}^{n-1} \binom{n-1}{i-1} u^i (1-u)^{n-i} \mathbf{p}_i + u^n \mathbf{p}_n \\
&= (1-u)\left[(1-u)^{n-1}\mathbf{p}_0 + \sum_{i=1}^{n-1} \binom{n-1}{i} u^i (1-u)^{n-i-1} \mathbf{p}_i\right] \\
&\quad + u\left[\sum_{i=1}^{n-1} \binom{n-1}{i-1} u^{i-1} (1-u)^{n-i} \mathbf{p}_i + u^{n-1}\mathbf{p}_n\right]
\end{aligned}
$$

The terms in square brackets are actually just Bézier curves on n points. Let $p_{i,j}(u)$ denote the Bézier curve defined by the points $\mathbf{p}_i, \mathbf{p}_{i+1}, \ldots, \mathbf{p}_j$. Clearly, $p_{0,n}(u)$ is just $p(u)$. Furthermore, by changing variables in the summations above it is easy to see that

$$
\begin{aligned}
p(u) &= (1-u)p_{0,n-1}(u) + u\, p_{1,n}(u) \\
&= p_{0,n-1}(u) + u[p_{1,n}(u) - p_{0,n-1}(u)]
\end{aligned}
\tag{11.54}
$$

It follows that the Bézier curve for n + 1 points is a simple convex linear combination of two Bézier curves on n points. This leads not only to an efficient way to evaluate Bézier curves but to a nice geometric construction for sketching such a curve that we shall indicate by looking at a few examples. First, note that the Bézier curve for one point $\mathbf{p}_0$ is just the constant function $p(u) = \mathbf{p}_0$. Using this fact and the recursive formula above gives that the Bézier curve for two points $\mathbf{p}_0$ and $\mathbf{p}_1$ is given by

$$
\begin{aligned}
p(u) &= (1-u)p_{00}(u) + u\, p_{11}(u) \\
&= (1-u)\mathbf{p}_0 + u\,\mathbf{p}_1.
\end{aligned}
$$

In other words, the Bézier curve is just the standard linear parameterization of the segment $[\mathbf{p}_0,\mathbf{p}_1]$. Next, to compute p(u) in the case of three points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, let

$$\mathbf{q}_1 = (1-u)\mathbf{p}_0 + u\,\mathbf{p}_1$$

and

$$\mathbf{q}_2 = (1-u)\mathbf{p}_1 + u\,\mathbf{p}_2.$$

Then $p(u) = (1-u)\mathbf{q}_1 + u\mathbf{q}_2$. Figure 11.9(a) shows how this works to find p(1/3). First, we find the point **A** that is one third of the way on the segment from $\mathbf{p}_0$ to $\mathbf{p}_1$, then **B**, which is one third of the way from $\mathbf{p}_1$ to $\mathbf{p}_2$. Finally, p(1/3) is the point one third of the way from **A** to **B**. Figure 11.9(b) shows an analogous construction for computing p(1/3) in the case of four points. The points **A**, **B**, and **C** are one third of the way on the segment from $\mathbf{p}_0$ to $\mathbf{p}_1$, from $\mathbf{p}_1$ to $\mathbf{p}_2$, and from $\mathbf{p}_2$ to $\mathbf{p}_3$, respectively. The

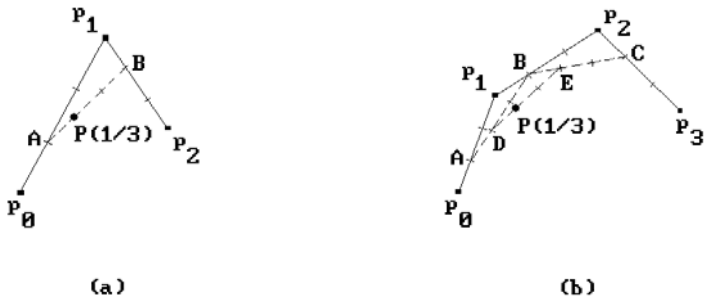


Figure 11.9. Graphing a Bézier curve geometrically.

Inputs: Control points $\mathbf{p}_i$, i = 0,1,...,n in $\mathbf{R}^m$ for a Bézier curve p(u)
$u \in [0,1]$

Output: p(u)

Step 1: Set $\mathbf{p}_i^0 = \mathbf{p}_i$.
Step 2: For r = 1,2,...,n and i = 0,1,...,n–r compute

$$\mathbf{p}_i^r = (1-u)\,\mathbf{p}_i^{r-1} + u\,\mathbf{p}_{i+1}^{r-1} .$$

When one has finished, $\mathbf{p}_0^n = p(u)$.

Algorithm 11.4.1. The de Casteljau algorithm.

points **C** and **D** are one third of the way from **A** to **B** and from **B** to **C**. The point p(1/3) is one third of the way from **D** to **E**.

The geometric construction just described translates into the de Casteljau evaluation algorithm shown in Algorithm 11.4.1.

11.4.1 Theorem. Algorithm 11.4.1 computes p(u).

Proof. Clear.

Algorithm 11.4.1 can be visualized via the following triangular array:

$$\begin{array}{ccccc} \mathbf{p}_0 & \mathbf{p}_1 & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \\ & \mathbf{p}_0^1 & \cdots & \mathbf{p}_{n-2}^1 & \mathbf{p}_{n-1}^1 \\ & & & \cdots & \cdots \\ & & & \mathbf{p}_0^{n-1} & \mathbf{p}_1^{n-1} \\ & & & & \mathbf{p}_0^n \end{array} \tag{11.55}$$

Each row in the array is generated from the row above it. Each element in a row is defined from the two elements in the row above it, which are immediately above and to the left of it. One does not need a two-dimensional array to implement the algorithm however. One needs only one array of length n + 1. As each new row is generated it is written on top of the previous row. Computing Bézier curves with the de Casteljau algorithm is much faster than straightforward evaluation of the polynomial formula.

To compute the derivative of the Bézier curve, note that

$$\begin{aligned}\frac{d}{du}B_{k,n}(u) &= \frac{d}{du}\binom{n}{k}u^k(1-u)^{n-k}\\ &= \frac{k\,n!}{k!(n-k)!}\,u^{k-1}(1-u)^{n-k} - \frac{(n-k)n!}{k!(n-k)!}\,u^k(1-u)^{n-k-1}\\ &= n\,(B_{k-1,n-1}(u) - B_{k,n-1}(u)).\end{aligned} \tag{11.56}$$

It follows that

$$\begin{aligned}\frac{d}{du}p\,(u) &= \sum_{i=0}^{n} n\,(B_{i-1,n-1}(u) - B_{i,n-1}(u))\mathbf{p}_i\\ &= n\sum_{i=1}^{n} B_{i-1,n-1}(u)\mathbf{p}_i - n\sum_{i=0}^{n-1} B_{i,n-1}(u)\mathbf{p}_i\\ &= n\sum_{i=0}^{n-1} B_{i,n-1}(u)\mathbf{p}_{i+1} - n\sum_{i=0}^{n-1} B_{i,n-1}(u)\mathbf{p}_i.\end{aligned}$$

($B_{-1,n-1}(n) = B_{n,n-1}(n) = 0$.) Collecting terms gives us the formula we want

$$\frac{d}{du}p\,(u) = n\sum_{i=0}^{n-1}(\mathbf{p}_{i+1} - \mathbf{p}_i)B_{i,n-1}(u). \tag{11.57}$$

Among other things, this shows that the derivative of a Bézier curve is another Bézier curve. In particular, one can compute the derivative efficiently with a de Casteljau type algorithm. Since higher derivatives are themselves just derivatives of functions, it should not be surprising that there is a recursive formula for the derivative of any order. See Theorem 11.5.2.6. Right now we simply note the following special cases, which follow from equation (11.57) applied to p(u) and p′(u):

$$p'(0) = n(\mathbf{p}_1 - \mathbf{p}_0), \quad p'(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1}) \tag{11.58}$$

and

$$p''(0) = n(n-1)(\mathbf{p}_2 - 2\mathbf{p}_1 + \mathbf{p}_0), \quad p''(1) = n(n-1)(\mathbf{p}_n - 2\mathbf{p}_{n-1} + \mathbf{p}_{n-2}). \tag{11.59}$$

Equation (11.58) explains another important property of Bézier curves: the vector $\mathbf{p}_0\mathbf{p}_1$ is tangent to the curve at the beginning and the vector $\mathbf{p}_{n-1}\mathbf{p}_n$ is tangent to the

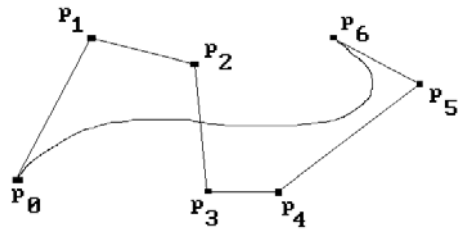


Figure 11.10. A Bézier curve.

curve at the end. The Bézier curve in Figure 11.10 shows some of basic properties possessed by Bézier curves: it lies in the convex hull of the control points, it starts at the first control point, ends at the last one, and is tangent to the first and last segment of the characteristic polygon there.

Another important property of Bézier curves is that they are affinely invariant. This follows both from Theorem 11.2.2.3 and from their recursive definition and the fact that affine maps preserve barycentric coordinates (Theorem 2.5.9 in [AgoM05]). As indicated earlier in Section 11.2.2, this is important for graphics because it says that to compute a Bézier curve that has been moved by an affine map, such as a rotation, all we have to do is compute the image of the control points and then recompute the curve from those new points. Not all curves have this property.

Although it is usually convenient to restrict the domain of a Bézier curve to [0,1], this is not necessary. More precisely, given any interval [a,b], a curve $q:[a,b] \to \mathbf{R}^m$ of the form

$$q(v) = p\left(\frac{v-a}{b-a}\right), \tag{11.60}$$

where p(u) is the Bézier curve on [0,1] defined by formula (11.51), is called a *Bézier curve*. Such a curve satisfies all the properties that the original Bézier curve satisfied. In particular, the important de Casteljau algorithm also applies to q(v) because the algorithm really only uses barycentric coordinates and does not care about the endpoints of the domain. See the general de Casteljau algorithm in Section 11.5.2. All of this is usually summarized by saying that Bézier curves are *invariant under affine parameter transformations*.

In conclusion, the way that one works with Bézier curves in practice on an interactive graphics system is:

(1) One sketches a curve by hand.
(2) One specifies vertices that outline the shape.
(3) One then moves, adds, or deletes vertices as necessary to improve the shape of the Bézier curve that is generated from these vertices.

The advantage of the Bézier curve over the Hermite curve is that it is more intuitive to specify four points than two points and two tangent vectors. Furthermore, the fact that the curve lies in the convex hull of its control points makes clipping easier. One first clips this convex hull against the window. If they do not intersect, then the curve will be outside the window.

Two problems with the general Bézier curve are

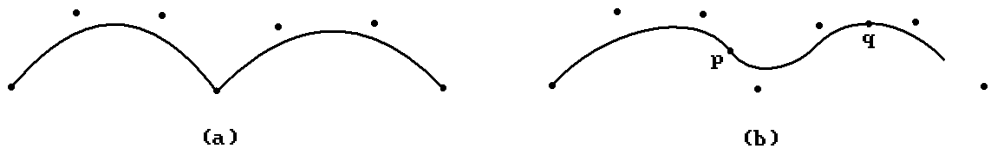


Figure 11.11. Bezier curves.

(1) The degree of the curve increases with the number of control points.
(2) There is no local control. The smallest change in any control points forces a recomputation of the whole curve, although the further that one is from the control point that was changed, the smaller is the change in the curve.

The usual way to avoid the first disadvantage is to use piecewise Bézier curves, but then one has to worry about the overall smoothness of the curve and whether the individual pieces fit together nicely. There are tricks that one can play to preserve visual smoothness, however. For example, by taking four control points at a time, one can get a piecewise cubic Bézier curve. If the last edge of one four-point control polygon is parallel to the first edge of the next control polygon, then the curve will look smooth without any apparent corner. To achieve this one can add control points to the original set that are the midpoints of appropriate segments. Figure 11.11(a) shows two four-point Bézier curves that meet in a corner. By adding the points **p** and **q**, the new curve in Figure 11.11(b), which now consists of three four-point Bézier curves, no longer has a corner. Of course, adding points means that one has changed the curve that a user is trying to design, but the change will probably not be noticeable. Besides, in an interactive environment, a user may not have a specific curve in mind anyway and is only interested in being able to control the general shape.

11.5. B-Spline Curves

11.5.1. The Standard B-Spline Curve Formulas

One common problem with the curves discussed so far is that any change to a control point forces recomputation of the whole curve. This is very undesirable. B-spline curves solve this problem. Changes to control points will have only a local effect on the curve.

B-splines can be defined in several ways. One can define them using

(1) a "brute force" approach by solving equations specifying certain constraints (see Theorem 11.5.1.1 for example),
(2) *truncated* (also called *one-sided*) *power functions* (see [deBo78] or [BaBB87]),
(3) recursion (the functions $N_{i,k}(u)$ defined later in this section),
(4) matrices (typically for the case of quadratic and cubic B-splines), or
(5) a multiaffine map approach (see Section 11.5.2).

We shall look at all of these approaches, except for the second, but begin by looking at a special case to help clarify and motivate the general discussion later on.

Polygons are the simplest examples of B-spline curves, although one normally does not think of them that way. Because they are so simple they are useful in highlighting some basic aspects of B-splines. Consider a polygon $\mathbf{X}$ with vertices $\mathbf{p}_i = (x_i, y_i)$, $i = 0,1, \ldots ,n$. Define functions S_i and S by

$$S_i(t) = (1-t)y_i + t\, y_{i+1} \tag{11.61}$$

and

$$\begin{aligned} S(t) &= y_0 && \text{if } t < x_0 \\ &= S_i((t-x_i)/(x_{i+1}-x_i)) && \text{if } x_i \le t \le x_{i+1}, \\ &= y_n && \text{if } x_n < t. \end{aligned} \tag{11.62}$$

The function S is then a spline function with respect to the knots $t_i = x_i$ and the polygon is just the graph of this function over the interval $[x_0,x_n]$. The graph of S_i over $[x_i,x_{i+1}]$ is just the segment $\mathbf{X}_i = [\mathbf{p}_i,\mathbf{p}_{i+1}]$ of $\mathbf{X}$ and S_i corresponds to a linear parameterization of $\mathbf{X}_i$.

Note that if we move any of the points $\mathbf{p}_i$, then only the two adjacent functions S_{i-1} and S_i are affected. We can write S as a sum of basis functions that localize changes similar to what we saw in the case of Lagrange and Hermite interpolation. Namely, for each i consider the unit *"hat" function* $b_i(t)$ defined by

$$\begin{aligned} b_i(t) &= 0 && \text{if } t < x_{i-1} \\ &= (t-x_{i-1})/(x_i-x_{i-1}) && \text{if } x_{i-1} \le t \le x_i, \\ &= (x_{i+1}-t)/(x_{i+1}-x_i) && \text{if } x_i \le t \le x_{i+1}, \\ &= 0 && \text{if } x_{i+1} < t. \end{aligned} \tag{11.63}$$

See Figure 11.12(a). The b_i are what are called **linear** B-splines because they are splines that are nonzero on only two spans. A property of these hat functions, which may not seem important at the moment, is that they sum to 1, that is,

$$\sum_{i=0}^{n} b_i(t) = 1 \quad \text{for } x_1 \le t \le x_{n-1} \tag{11.64}$$

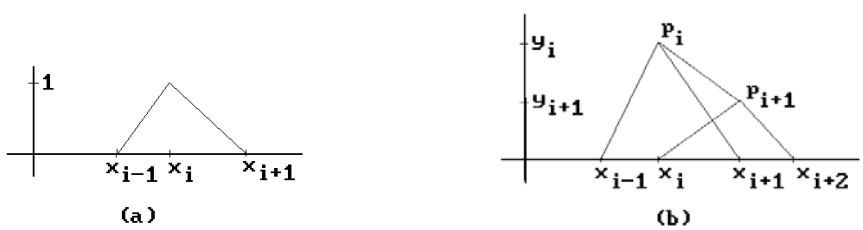


Figure 11.12. Linear B-splines.

Each segment of the polygon, except the first and last, is just the graph of the sum of two of the hat functions. More precisely,

$$S_i(t) = y_i b_i(t) + y_{i+1} b_{i+1}(t) \quad \text{for } 0 < i < n. \tag{11.65}$$

See Figure 11.12(b). In other words, if we stay away from the endpoints, then

$$S(t) = \sum_{i=0}^{n} y_i b_i(t) \quad \text{for} \quad x_1 \le t \le x_{n-1}. \tag{11.66}$$

Furthermore, the natural parameterization of the graph of a function and equation (11.66) leads to the following parameterization of $\mathbf{X}$:

$$t \to (t, S(t)) = \left(t, \sum_{i=0}^{n} y_i b_i(t) \right) = \sum_{i=0}^{n} b_i(t)(x_i, y_i).$$

The last equality is justified by the fact that

$$t = \sum_{i=0}^{n} b_i(t) x_i,$$

which follows from equation (11.66) applied to the special case where $y_i = x_i$, that is, the case $S(t) = t$. In other words, we have another example of a parametric curve in the form

$$p(u) = \sum_{i=0}^{n} b_i(u) \mathbf{p}_i, \tag{11.67}$$

for some functions $b_i(u)$ and "control points" $\mathbf{p}_i$. Basically much of the discussion in this chapter revolves around these are the types of parametric curves and deciding what is the best choice for the functions $b_i(u)$.

Using linear B-splines gave us a continuous curve. If we want a smoother curve, then we need to use higher-degree basis functions. Quadratic B-splines will give us differentiable curves. Cubic B-splines will give us twice differentiable curves. Here are the desirable properties that a general B-spline of degree m should have:

(1) It should be a spline of degree m.
(2) It should vanish outside of m + 1 sequentially contiguous spans. Equivalently, its support should be contained in m + 1 contiguous spans.

We are not going to make this into a formal definition of a B-spline, however, because we shall reserve that word for a specific family of functions, namely, the function $N_{i,k}(u)$, which will be defined shortly, and those functions will also not be quite as dif-

ferentiable as a spline is supposed to be. The important condition is that the splines have finite support because that is what will give us local control. In that regard, we should point out that the number m + 1 of spans that is supposed to contain the support of a B-spline of degree m was not chosen in an arbitrary way. This is in general the smallest number of spans over which an (m – 1)-differentiable piecewise polynomial function is nonzero.

The proof of the next theorem is an example of how a brute force method can be used to show that cubic splines satisfying the finite support condition exist.

11.5.1.1 Theorem. Given distinct knots x_{i-2}, x_{i-1}, x_i, x_{i+1}, and x_{i+2}, there is a **unique** cubic spline $b_i(t)$ such that

(1) $b_i(t) = 0$, for $t < x_{i-2}$ or $x_{i+2} < t$, and
(2) $b_i(x_{i-1}) + b_i(x_i) + b_i(x_{i+1}) = 1$.

Proof. See [BaBB87]. Basically, we have four cubic polynomials, one for each span, which gives us 16 degrees of freedom. The fact that we want the polynomials to meet at the five knots and have the same first and second derivatives there gives us 15 constraints. (At the endpoints x_{i-2} and x_{i+2} the functions and their derivatives are zero.) The normalizing constraint in (2) is the extra one needed for a unique solution. This solution consists of the four polynomials

$$\begin{aligned}
&(1/6)\,u^3, && u = \frac{t - x_{i-2}}{x_{i-1} - x_{i-2}},\\
&(1/6)\left(-3u^3 + 3u^2 + 3u + 1\right), && u = \frac{t - x_{i-1}}{x_i - x_{i-1}},\\
&(1/6)\left(3u^3 - 6u^2 + 4\right), && u = \frac{t - x_i}{x_{i+1} - x_i}, \quad \text{and}\\
&(1/6)\left(-u^3 + 3u^2 - 3u + 1\right), && u = \frac{t - x_{i+1}}{x_{i+2} - x_{i+1}},
\end{aligned} \tag{11.68}$$

where t ranges over the intervals $[x_{i-2},x_{i-1}]$, $[x_{i-1},x_i]$, $[x_i,x_{i+1}]$, and $[x_{i+1},x_{i+2}]$, respectively. See Figure 11.13. Without condition (2), there would be many solutions.

After all these preliminaries, here are the functions we are after. We give the Cox-de Boor recursive definition.

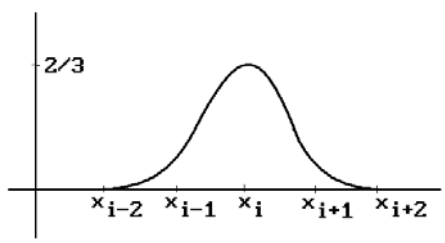


Figure 11.13. A cubic B-spline basis function.

Definition. Given $n \geq 0$, $k \geq 1$, and a nondecreasing sequence of real numbers $U = (u_0, u_1, \ldots, u_{n+k})$, define functions

$$N_{i,k} : \mathbf{R} \to \mathbf{R}, 0 \leq i \leq n,$$

recursively as follows:

$$\begin{aligned} N_{i,1}(u) &= 1, \quad \text{for } u_i \leq u < u_{i+1}, \text{and} \\ &= 0, \quad \text{elsewhere.} \end{aligned} \tag{11.69a}$$

If $k > 1$, then

$$N_{i,k}(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_{i,k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} N_{i+1,k-1}(u), \tag{11.69b}$$

where, if any term is of the form 0/0, we replace it by 0. The function $N_{i,k}(u)$ is called the ith *B-spline*, or *B-spline basis function*, *of order k and degree k–1 with respect to the knot vector U*.

Note. We need to make one technical point. Although a B-spline of order k is a spline of some order, it is not necessarily a spline of order k. If some knot has multiplicity larger than 1, then a B-spline is not differentiable enough at that point. See Theorem 11.5.1.4(2). This causes an occasional awkwardness when talking about such functions. On the other hand, it turns out that **any** spline is a linear combination of B-splines basis functions (Theorem 11.5.2.16), so that one not lose anything if one concentrates on these particular splines. The fact that B-splines form a basis for the space of splines is actually what gave B-splines their name because the "B" in the name stands for "basis" ([Scho67]).

A spline is greatly influenced by how the knots are chosen. Before we work through some examples showing the shape of a few of the splines $N_{i,k}$ (u) it is convenient to introduce some terminology.

Definition. A knot vector for a spline or a B-spline of order k is said to be *clamped* if the first and last knot each has multiplicity k. Otherwise, it is said to be *unclamped*. An (unclamped) knot vector U of length L is said to be *uniform* or *periodic* if the knots u_i are evenly spaced, that is, there is a constant $d > 0$, so that $u_{i+1} = u_i + d$ for $0 \leq i \leq L - 2$. If U is clamped, then it is said to be *uniform* if all the knots u_i except the first and last k knots are evenly spaced, that is, $u_{i+1} = u_i + d$, for $k \leq i < L - k$. A knot vector that is not uniform is said to be *nonuniform*.

Definition. The adjectives *clamped*, *unclamped*, *uniform*, *periodic*, or *nonuniform* are applied to a spline or B-spline if they apply to its knot vector.

Note. Unfortunately, the terminology for splines and their knot vectors did not develop in a consistent way. Other terms can also be found in the literature. For example, the term "open uniform" is sometimes used for what we are calling clamped

uniform. "Nonperiodic" has been used to mean clamped nonuniform. The terms above seem to make the most sense and are starting to be used. See [PieT95].

Two types of knot vectors are common enough to deserve a name.

Definition. Let $(u_0, u_1, \ldots, u_{n+k})$ be the knot vector for a B-spline of order k. It will be called the *standard uniform* knot vector if $u_i = i$. It is called the *standard clamped uniform* knot vector if

$$\begin{aligned} u_0 &= u_1 = \ldots = u_{k-1} = 0 \\ u_i &= i - k + 1, \quad \text{for } k \le i \le n, \\ u_{n+1} &= u_{n+2} = \ldots u_{n+k} = n - k + 2. \end{aligned} \tag{11.70}$$

To help the reader get a feel for the functions $N_{i,k}(u)$ we shall compute a few cases using the standard clamped uniform knot vector. In this case we may assume that $0 \le u \le n - k + 2$ because all the functions vanish outside that interval. Note that to compute the $N_{i,k}(u)$ for a fixed k, we **must** use the **same** knot sequence for all the $N_{i,j}(u)$, $1 \le j < k$.

11.5.1.2 Example. n = 3, k = 1:

Solution. The knots u_i in this case are

$$u_0 = 0, u_1 = 1, u_2 = 2, u_3 = 3, u_4 = 4.$$

Figure 11.14 shows the graphs of $N_{i,1}(u)$.

11.5.1.3 Example. n = 3, k = 2:

Solution. This time the u_i are

$$u_0 = u_1 = 0, u_2 = 1, u_3 = 2, u_4 = u_5 = 3.$$

Since the knot values have changed, we cannot use the $N_{i,1}(u)$ that were computed in Example 11.5.1.2 and must recompute them. The new graphs are shown in Figure 11.15(a). The $N_{i,2}(u)$ reduce to the following:

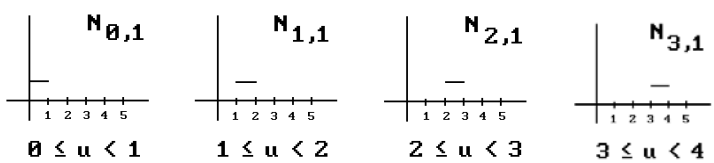


Figure 11.14. The functions $N_{i,1}(u)$ in Example 11.5.1.2.

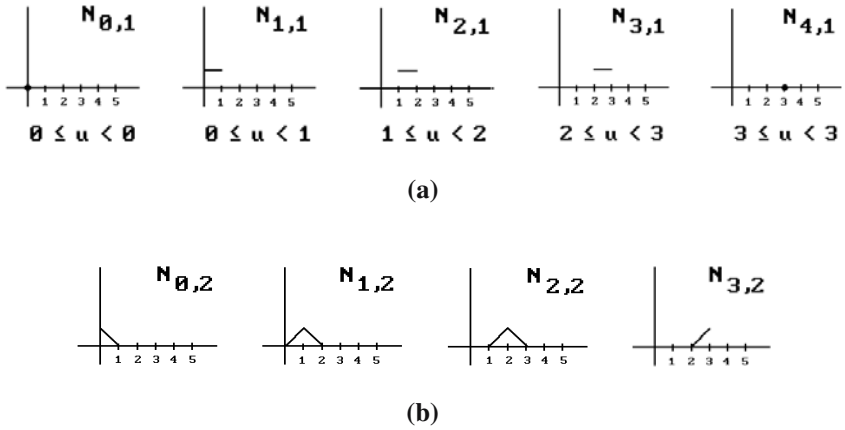


Figure 11.15. The functions $N_{i,j}(u)$ in Example 11.5.1.3.

$$
\begin{aligned}
N_{0,2}(u) &= \frac{u-u_0}{u_1-u_0}N_{0,1}(u) + \frac{u_2-u}{u_2-u_1}N_{1,1}(u) \\
&= (1-u)N_{1,1}(u) \\
N_{1,2}(u) &= \frac{u-u_1}{u_2-u_1}N_{1,1}(u) + \frac{u_3-u}{u_3-u_2}N_{2,1}(u) \\
&= u\,N_{1,1}(u) + (2-u)N_{2,1}(u) \\
N_{2,2}(u) &= \frac{u-u_2}{u_3-u_2}N_{2,1}(u) + \frac{u_4-u}{u_4-u_3}N_{3,1}(u) \\
&= (u-1)N_{2,1}(u) + (3-u)N_{3,1}(u) \\
N_{3,2}(u) &= \frac{u-u_3}{u_4-u_3}N_{3,1}(u) + \frac{u_5-u}{u_5-u_4}N_{4,1}(u) \\
&= (u-2)N_{3,1}(u)
\end{aligned}
$$

The graphs of the $N_{i,2}(u)$ are shown in Figure 11.15(b). We use $N_{1,2}(u)$ as an example to show how the graphs are determined. Since the formula for $N_{1,2}(u)$ is a linear combination of $N_{1,1}(u)$ and $N_{2,1}(u)$, $N_{1,2}(u)$ will be zero except possibly over the intervals [0,1] and [1,2]. If $u \in [0,1]$, then $N_{1,1}(u)$ is 1 and $N_{2,1}(u)$ is zero. Therefore, $N_{1,2}(u) = u$. If $u \in [1,2]$, then $N_{1,1}(u)$ is 0 and $N_{2,1}(u)$ is 1. Therefore, $N_{1,2}(u) = 2 - u$.

Generalizing from Examples 11.5.1.2 and 11.5.1.3, if one wants to compute the $N_{i,k}(u)$ by hand, one would first recursively use formula (11.69b) to express the function as a linear combination of the $N_{j,1}(u)$ in the form

$$N_{i,k}(u) = \sum_j a_j(u)N_{j,1}(u). \tag{11.71}$$

Let $\mathbf{I}_i$ be the interval over which $N_{i,1}(u)$ is nonzero. Then, since $\mathbf{I}_i \cap \mathbf{I}_j = \phi$ if $i \neq j$, the formula for $N_{i,k}(u)$ over $\mathbf{I}_j$ is just the coefficient function $a_j(u)$ of $N_{j,1}(u)$. We shall see

another example of this process shortly when we describe the uniform quadratic B-splines.

11.5.1.4 Theorem. The functions $N_{i,k}(u)$ defined by equations (11.69) satisfy the following properties:

(1) (Compact support property) The function $N_{i,k}(u)$ vanishes on $(-\infty,u_i) \cup [u_{i+k},\infty)$. In particular, only the functions $N_{i-k+1,k}(u)$, $N_{i-k+2,k}(u)$, . . . , $N_{i,k}(u)$ are nonzero on $[u_i,u_{i+1})$.

(2) (Differentiability property) The function $N_{i,k}(u)$ is C^∞ on the interior of the spans and C^{k-1-m} at a knot of multiplicity m. In particular, $N_{i,k}(u)$ is a spline of order k if all knots have multiplicity 1.

(3) $N_{i,k}(u) \geq 0$, for all u.

(4) The identity

$$\sum_{i=0}^{n} N_{i,k}(u) = 1$$

holds for all $u \in [k-1,n+1]$ and fails for any other u. If the knot vector is clamped, then the identity holds for all $u \in [0,n+k]$.

Proof. See [Seid89], [PieT95], or [Fari97]. Parts (1) and (3) are easy to prove using induction on k. Note that they are trivially true when k is equal to 1. Induction also plays a big role in the proof of the other parts. The reason for the restricted domain in part (4) will become clearer in our discussion of uniform quadratic B-splines below. Also, if the knot vector is clamped, then $[k-1,n+1] = [0,n+k]$.

The compact support property of B-splines is important because it means that one can make local modifications to curves based on such functions without having to recompute the whole curve. Theorem 11.5.1.4(3) and (4) show that the B-spline basis functions act like barycentric coordinates, which will be important for convexity issues later.

Definition. Given a sequence of points $\mathbf{p}_i$, i = 0,1, . . . ,n, the curve

$$p(u) = \sum_{i=0}^{n} N_{i,k}(u)\mathbf{p}_i. \tag{11.72}$$

is called the *B-spline curve of order k* (or *degree m = k − 1*) with *control* or *de Boor points* $\mathbf{p}_i$ and *knot vector* $(u_0,u_1, \ldots ,u_{n+k})$. The adjectives *clamped*, *unclamped*, *uniform*, *periodic*, or *nonuniform* are applied to the curve if they apply to its knot vector. The *domain* of the curve is defined to be the interval $[u_{k-1},u_{n+1}]$. (Note that if the curve is clamped, then the domain is the whole interval $[u_0,u_{n+k}]$.) Each piece $p([u_i,u_{i+1}])$, $k-1 \leq i \leq n$, of the whole curve traced out by p(u) is called a *segment* of the curve. The polygonal curve defined by the control points is called the *de Boor* or *control polygon* of the curve.

The reason that nonclamped B-spline curves have a restricted domain is that we want the identity in Theorem 11.5.1.4(4) to hold.

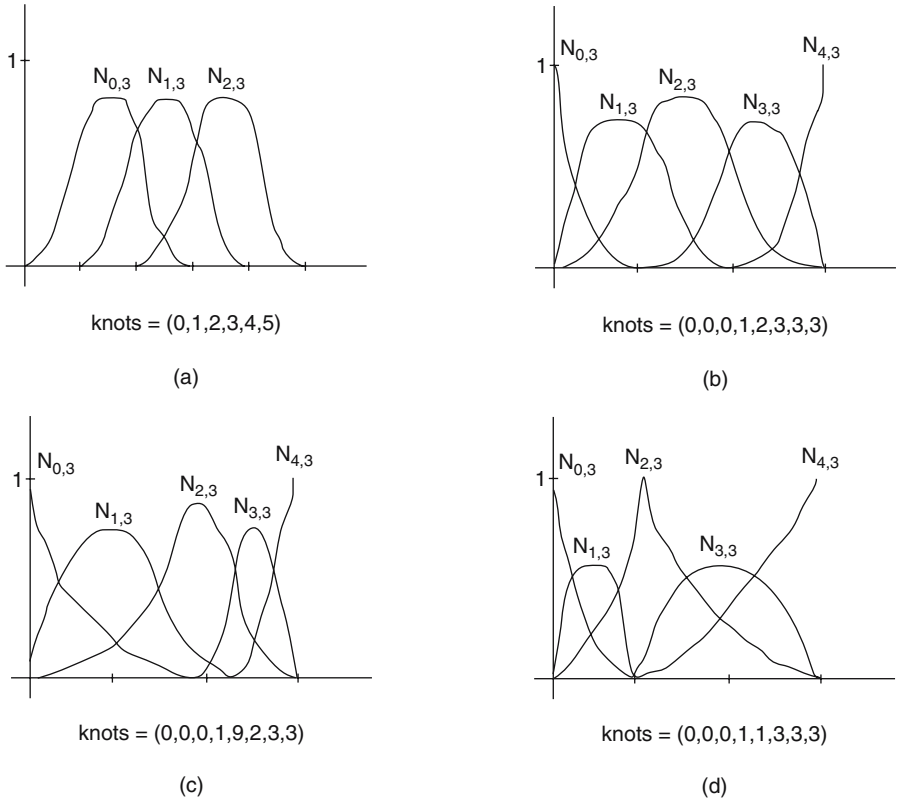


Figure 11.16. Some quadratic B-splines.

Because of the importance of B-spline curves to geometric modeling, much effort has gone into coming up with efficient algorithms for computing them and their derivatives. We leave the discussion of these algorithms to Section 11.5.4, at which time we will also be able to include the rational B-spline curves. In this section will shall only consider some special ways of computing and working with the quadratic and cubic curves which, hopefully, will aid the reader in understanding the general curves.

Consider Figure 11.16, which shows some quadratic B-splines. Note how nice and symmetric the curves look in Figure 11.16(a) and (b) when the knots are spaced evenly and how this symmetry is lost in Figure 11.16(c) and (d) when the knot spacing becomes irregular. The cusp in Figure 11.16(d) is caused by the knot 1 which has multiplicity 2. Figure 11.17 shows all the clamped uniform cubic B-splines $N_{i,4}(u)$ associated to the knot vector (0,0,0,0,1,2,3,4,5,6,6,6,6). Again we have regularly spaced knots and symmetric curves.

The fact is that over uniformly spaced knots the B-splines of a given degree all have the same shape and are just translates of each other. This is easy to see from the definition of the $N_{i,k}(u)$. Assume we have a knot vector (0,1,2, . . .), so that $u_i = i$. A simple induction shows that

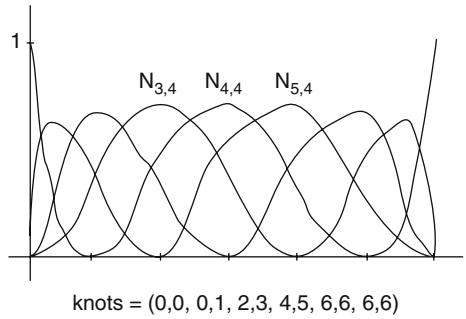


Figure 11.17. The clamped uniform cubic B-splines $N_{i,4}(u)$ for n = 8.

$$N_{i+j,1}(u) = N_{i,1}(u - j). \tag{11.73}$$

Using formula (11.69b) and induction on k then easily leads to

$$N_{i+j,k}(u) = N_{i,k}(u - j). \tag{11.74}$$

This property of splines having the same shape is also true in the case of clamped uniform knot vectors as long as one stays away from the ends of the knots. We saw this in the case of linear B-splines and the hat functions. The middle three curves $N_{3,4}(u)$, $N_{4,4}(u)$, and $N_{5,4}(u)$ in Figure 11.17 are another good example, as are the formulas in Theorem 11.5.1.1. This then is our next goal, to analyze the $N_{i,k}(u)$ for standard uniform knot vectors and to find formulas for the quadratic and cubic uniform B-spline curves.

We start with the uniform quadratic B-splines and assume that $u_j = j$, $0 \le j \le n + 3$. Let $0 \le i \le n$. It is easy to show that formulas (11.69) imply that

$$\begin{aligned} N_{i,1}(u) &= 1, \quad \text{for } i \le u \le i+1, \text{and} \\ &= 0, \quad \text{elsewhere.} \end{aligned} \tag{11.75}$$

$$N_{i,2}(u) = (u - i)N_{i,1}(u) + (i + 2 - u)N_{i+1,1}(u) \tag{11.76}$$

$$\begin{aligned} N_{i,3}(u) = (1/2)\Big[&(u - i)^2 N_{i,1}(u) + (u - i)(i + 2 - u)N_{i+1,1}(u) + \\ &(i + 3 - u)(u - i - 1)N_{i+1,1}(u) + (i + 3 - u)^2 N_{i+2,1}(u)\Big] \end{aligned} \tag{11.77}$$

Next, we want to use the $N_{i,3}(u)$ defined by (11.77) to define a quadratic B-spline curve p(u) for control points $\mathbf{p}_i$, i = 0,1, . . . ,n. Before we do that and use formula (11.72) for p(u), take a look at Figure 11.16(a) again. Over the first and last two spans there are some "missing" B-splines. They are missing in the sense that there are **three** nonzero B-splines over the **middle** spans between 2 and n + 1, whereas there are **fewer** nonzero B-splines over those **end** spans.

Note. There are similar missing B-splines over the end spans in the general case and this explains why the identity in Theorem 11.5.1.4(4) fails outside of $[u_{k-1}, u_{n+1}]$.

Returning to the uniform quadratic B-spline curves p(u) defined by (11.72), we want to show that they can be computed very easily using matrices. Assume that $u \in [i,i+1)$. Since only $N_{i,1}(u)$ is nonzero for such a u, formulas (11.72) and (11.77) show that the coefficient of $\mathbf{p}_j$ vanishes for all j except for $j = i - 2$, $i - 1$, or i. In other words,

$$p(u) = (1/2)(i+1-u)^2\mathbf{p}_{i-2} + (1/2)[(u-i+1)(i+1-u) + (i+2-u)(u-i)]\mathbf{p}_{i-1} + (1/2)(u-i)^2\mathbf{p}_i.$$

Let $q_{i-1}(u)$, $u \in [0,1]$, be the restriction of p(u) to the interval [i,i+1] but reparameterized to [0,1]. Then

$$\begin{aligned} q_{i-1}(u) &= p(i+u) \\ &= (1/2)\left[(1-u)^2\mathbf{p}_{i-2} + \left(-2u^2 + 2u + 1\right)\mathbf{p}_{i-1} + u^2\mathbf{p}_i\right]. \end{aligned}$$

Notice how the coefficients of the points are independent of i. In matrix form (replacing i by i + 1),

$$q_i(u) = \left(u^2\ u\ 1\right)\mathbf{M}_{s2}\begin{pmatrix}\mathbf{p}_{i-1}\\ \mathbf{p}_i\\ \mathbf{p}_{i+1}\end{pmatrix}, \tag{11.78}$$

where $\mathbf{M}_{s2}$ is the *quadratic uniform* or *periodic B-spline matrix* defined by

$$\mathbf{M}_{s2} = (1/2)\begin{pmatrix}1 & -2 & 1\\ -2 & 2 & 0\\ 1 & 1 & 0\end{pmatrix}. \tag{11.79}$$

The curve $q_i(u)$ is defined for $1 \le i \le n - 1$ and $u \in [0,1]$. It traces out the same set as the original quadratic B-spline p(u) restricted to u in $[i - 1,i]$. Derivatives are now also computed easily. For example,

$$q_i'(u) = (2u\ 1\ 0)\mathbf{M}_{s2}\begin{pmatrix}\mathbf{p}_{i-1}\\ \mathbf{p}_i\\ \mathbf{p}_{i+1}\end{pmatrix}. \tag{11.80}$$

Repeating the above steps in the case where p(u) is a uniform cubic B-spline produces a similar result. More precisely, if $1 \le i \le n - 2$ and $u \in [0,1]$, then

$$q_i(u) = \left(u^3\ u^2\ u\ 1\right)\mathbf{M}_{s3}\begin{pmatrix}\mathbf{p}_{i-1}\\ \mathbf{p}_i\\ \mathbf{p}_{i+1}\\ \mathbf{p}_{i+2}\end{pmatrix}, \tag{11.81}$$

where $\mathbf{M}_{s3}$ is the *cubic uniform* or *periodic B-spline matrix* defined by

$$\mathbf{M}_{s3} = (1/6)\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}. \tag{11.82}$$

Note how these formulas are compatible with the formulas in equation (11.68). The derivative can be computed using

$$q_i'(u) = \begin{pmatrix} 3u^2 & 2u & 1 & 0 \end{pmatrix}\mathbf{M}_{s3}\begin{pmatrix} \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{p}_{i+2} \end{pmatrix}. \tag{11.83}$$

The points $\mathbf{p}_j$ in equations (11.78) and (11.81) are called the *ith B-spline coefficients* of the uniform curve p(u). One can also define the nonuniform B-splines with matrices, but this takes more than one matrix. One needs separate matrices for computing the curve near the endpoints. See [PokG89].

One important difference between clamped uniform and uniform B-spline curves is that the former start at the first control point and end at the last one whereas the latter do not. See Figure 11.18. As a partial demonstration of this point we analyze the quadratic uniform case more closely. Using formula (11.78) we see that

$$q_i(0) = (1/2)(\mathbf{p}_{i-1} + \mathbf{p}_i)$$

and

$$q_{n-1}(1) = (1/2)(\mathbf{p}_{n-1} + \mathbf{p}_n),$$

that is, the curve starts at the midpoint of the segment $[\mathbf{p}_0, \mathbf{p}_1]$, ends at the midpoint of $[\mathbf{p}_{n-1}, \mathbf{p}_n]$, and passes through the midpoints of all the other segments. Since

$$q_i'(0) = \mathbf{p}_i - \mathbf{p}_{i-1}$$

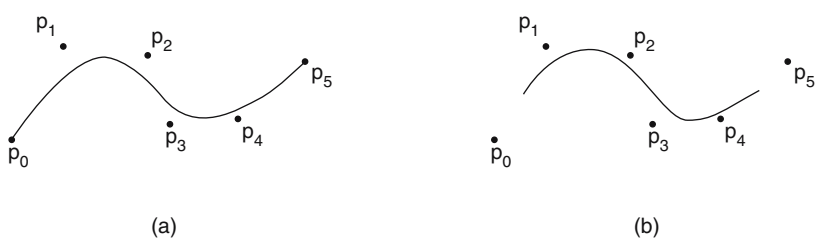


Figure 11.18. Clamped uniform versus uniform B-splines.

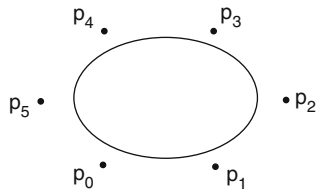


Figure 11.19. A closed cubic uniform B-spline curve for n = 5.

and

$$q_i'(1) = \mathbf{p}_{i+1} - \mathbf{p}_i,$$

we also see that the first and last segment of the control polygon are tangent to the curve at the first and last point, respectively.

There is a trick one can use to force a uniform B-spline to start at the first control point and end at the last one to mimic the clamped uniform case. One can add some "phantom" endpoints. One defines

$$\mathbf{p}_{-1} = 2\,\mathbf{p}_0 - \mathbf{p}_1 \quad \text{and} \quad \mathbf{p}_{n+1} = 2\,\mathbf{p}_n - \mathbf{p}_{n-1}. \tag{11.84}$$

See [BaBB87].

Next, we look at closed B-spline curves. The uniform B-spline curves come in handy here. However, to close a curve we have to do more than simply add the first point to the end of the control point sequence. Figure 11.19 shows a closed cubic uniform B-spline curve with control points ($\mathbf{p}_0$,$\mathbf{p}_1$,$\mathbf{p}_2$,$\mathbf{p}_3$,$\mathbf{p}_4$, $\mathbf{p}_5$,$\mathbf{p}_0$,$\mathbf{p}_1$,$\mathbf{p}_2$). A simple modification to formulas (11.78) and (11.81) leads to the following formulas for closed curves. Let $1 \le i \le n + 1$ and $u \in [0,1]$.

The closed quadratic uniform B-spline curve:

$$q_i(u) = \begin{pmatrix} u^2 & u & 1 \end{pmatrix} \mathbf{M}_{s2} \begin{pmatrix} \mathbf{p}_{i-1 \bmod (n+1)} \\ \mathbf{p}_{i \bmod (n+1)} \\ \mathbf{p}_{i+1 \bmod (n+1)} \end{pmatrix}. \tag{11.85}$$

The closed cubic uniform B-spline curve:

$$q_i(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \mathbf{M}_{s3} \begin{pmatrix} \mathbf{p}_{i-1 \bmod (n+1)} \\ \mathbf{p}_{i \bmod (n+1)} \\ \mathbf{p}_{i+1 \bmod (n+1)} \\ \mathbf{p}_{i+2 \bmod (n+1)} \end{pmatrix}. \tag{11.86}$$

Before we leave the subject of B-splines as matrices, we should point out that, although this is an efficient way to compute them, the disadvantage to using these matrices in a program is that it would involve code for lots of special cases. For that reason and the fact that computers are powerful enough these days, a general

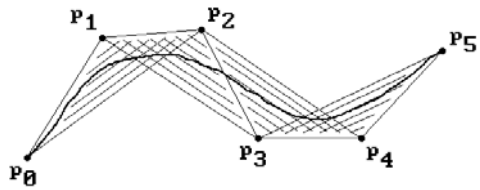


Figure 11.20. The local convex hull property of B-splines.

geometric modeling program would be unlikely to use them anymore. Instead, one uses the algorithms described in Section 11.5.4.

Like Bézier curves, B-spline curves p(u) satisfy a convex hull property. Specifically, it follows from Theorem 11.5.1.4(3) and (4) that B-spline curves defined by equation (11.72) lie in the convex hull of their control points. Actually, a stronger fact is true since most of the coefficients in (11.72) are zero for any given u.

11.5.1.5 Theorem. (Local convex hull property) Successive segments of a B-spline curve of order k are contained in the convex hull of the corresponding sequence of k control points (the coefficients of the other points vanish). More precisely, suppose that the B-spline curve p(u) has knots u_i and control points $\mathbf{p}_i$, then $p([u_i,u_{i+1}))$ is contained in the convex hull of the points $\mathbf{p}_{i-k+1}, \mathbf{p}_{i-k+2}, \ldots, \mathbf{p}_i$.

Proof. Use Theorem 11.5.1.4.

In addition to being useful for clipping, Theorem 11.5.1.5 also tells us more about the curve. For example, in the quadratic case, the B-spline curve lies in the union of the triangles $\mathbf{p}_i\mathbf{p}_{i+1}\mathbf{p}_{i+2}$, $0 \le i \le n - 2$. See Figure 11.20. Similar facts are true in the general case.

Finally, we list, without proof, several additional facts about B-spline curves. Proofs can be found in [Fari97] or could be derived from the material in Section 11.5.2. These are facts that anyone working with B-spline curves should know. They deal with how the shape of such curves changes as certain properties are changed.

(1) If the B-spline curve is clamped, then it interpolates the first and last control point.

(2) Inserting multiple control points eventually forces the curve to go through the point, as does increasing the multiplicity of knots. See [Fari97] or the next section on how to insert knots. See also Section 11.5.3 for situations where one might want to do this.

(3) Increasing the order of a B-spline curve for a fixed number of control points reduces the number of curve segments, so that the influence of any given control point on a segment is reduced. Also, each curve segment now lies in the convex hull of a larger number of control points. Conversely, reducing the order of a B-spline means that each control point influences fewer segments although its influence is stronger.

(4) Given a nonperiodic B-spline defined by equations (11.69) we must have $n \ge k - 1$, otherwise the equation for the integer knots breaks down. In other words, one must have at least as many control points as the order of the spline. One needs $n \ge k$ if one wants a spline with a domain that is larger than a point.

(5) Bézier curves can be considered as a special case of B-spline curves. For example, the four B-splines from the periodic spline over the knot sequence 0,0,0,0,1,1,1,1 give the cubic Bernstein polynomials.

11.5.2 The Multiaffine Approach to B-Splines

This section describes another geometric approach to Bézier curves and B-splines. The approach taken in the previous two sections is in a sense rather ad hoc but it has the advantage that it is easier to follow initially. On the other hand, it gets messy and complicated to prove anything. The multiaffine approach in contrast may seem totally confusing initially with all of its sub- and superscripts, but, once one gets over that, lots of properties of Bézier and B-spline curves become really trivial and very geometric. The core of the material in this section comes from [Seid89]. See also [Seid93] and [Rock93].

It is an old fact in mathematics that there is a kind of duality between degree **k** polynomials of **one** variable and multivariate polynomials that have degree **1** in **each** variable. To show this we need some definitions. Let **V** and **W** be real vector spaces.

Definition. A function $g : \mathbf{V} \to \mathbf{W}$ is said to be an *affine* map if

$$g\left(\sum_{i=1}^{k} a_i \mathbf{v}_i\right) = \sum_{i=1}^{k} a_i g(\mathbf{v}_i)$$

for all $k > 0$, $\mathbf{v}_i \in \mathbf{V}$, and $a_i \in \mathbf{R}$ satisfying $\sum_{i=1}^{k} a_i = 1$. A function $f : \mathbf{V}^d \to \mathbf{W}$ is said to be a *multiaffine* map if for all i and $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{i-1}, \mathbf{v}_{i+1}, \ldots, \mathbf{v}_d \in \mathbf{V}$, the map $g_i : \mathbf{V} \to \mathbf{W}$ defined by

$$g_i(\mathbf{v}) = f(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{i-1}, \mathbf{v}, \mathbf{v}_{i+1}, \ldots, \mathbf{v}_d)$$

is an affine map.

The definition of an affine map given here agrees with the "usual" definition of an affine map, namely, that it is a map that sends lines to lines. (In the case of maps from $\mathbf{R}^n$ to $\mathbf{R}^n$ this follows from Theorem 2.5.9 in [AgoM05].) In this section the barycentric coordinate preserving property of affine maps is emphasized because that is the key to everything that we do here. We will be using this property over and over again. Proofs of results will be trivial as long as one keeps this in mind. Figure 11.21 shows the critical property of linear maps when using barycentric coordinates. If the linear map from one simplex to another maps a point **p** to the point **p′**, then both **p** and **p′** have the **same** barycentric coordinates, albeit with respect to different vertices. The map simply replaced the vertices in the barycentric coordinate representation.

If $\mathbf{V} = \mathbf{R}$ and $\mathbf{W} = \mathbf{R}^m$, which is the special case of interest to us, then $g(u) = (g_1(u), g_2(u), \ldots, g_m(u))$ and g is an affine map if and only if each g_i is a polynomial of degree 1. Furthermore, a multiaffine map is then simply a polynomial of degree 1 in each variable separately.

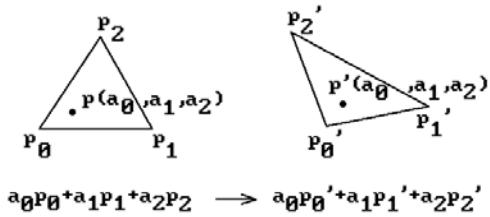


Figure 11.21. Affine maps preserve barycentric coordinates.

11.5.2.1 Theorem. (The **Polar Form** or **Blossoming Theorem**) Let

$$p : \mathbf{R} \to \mathbf{R}^m$$

be a polynomial function of degree d. Then there exists a **unique** symmetric multi-affine map

$$P : \mathbf{R}^d \to \mathbf{R}^m$$

satisfying $P(u, \ldots, u) = p(u)$. Furthermore, the rth derivative of p is given by

$$p^{(r)}(u) = \frac{d!}{(d-r)!} \sum_{i=0}^{r} \binom{r}{i} (-1)^{r-i} P(\underbrace{u, \ldots, u}_{d-i}, \underbrace{u+1, \ldots, u+1}_{i}). \tag{11.87}$$

The map P is called the *polar form* or *blossom* of p.

Proof. See [Rams88]. Express p in the form

$$p(u) = \sum_{i=0}^{d} \mathbf{a}_i \binom{d}{i} u^i,$$

where $\mathbf{a}_i \in \mathbf{R}^m$, and let

$$P(u_1, u_2, \ldots, u_d) = \sum_{i=0}^{d} \mathbf{a}_i \sigma_i(u_1, u_2, \ldots, u_d),$$

where $\sigma_i = \sigma_i(u_1, u_2, \ldots, u_d)$ is the ith elementary symmetric polynomial in the variables $u_1, u_2, \ldots, u_d$. It is easy to check that

$$P(u, \ldots, u) = p(u).$$

Next, let $d : \mathbf{R} \to \mathbf{R}^d$ be the diagonal map

$$d(u) = (u, \ldots, u).$$

Then $p(u) = P(d(u))$. Therefore, by the chain rule

$$p'(u) = DP(d(u))d'(u).$$

This fact and induction leads to a proof of equation (11.87). The uniqueness of P follows from the derivative formula (11.87).

Note. The term "polar form" comes from mathematics. The term "blossom" was introduced by Ramshaw in [Rams88]. A polynomial can be recovered from its blossom.

11.5.2.2 Example. A classical example of a blossom is the dot product function $P(\mathbf{u},\mathbf{v}) = \mathbf{u} \bullet \mathbf{v}$ in $\mathbf{R}^n$, which is the blossom of the length squared function $p(\mathbf{v}) = |\mathbf{v}|^2$.

11.5.2.3 Example. The blossom of the cubic polynomial

$$p(u) = a_0 + a_1 u + a_2 u^2 + a_3 u^3$$

is

$$P(u_1, u_2, u_3) = a_0 + (a_1/3)(u_1 + u_2 + u_3) + (a_2/3)(u_1u_2 + u_1u_3 + u_2u_3) + a_3u_1u_2u_3.$$

What is the point of all of this? Replacing a polynomial of degree k with a function in k variables that is linear in each variable turns out to be very useful. We shall see that it is another example of how nonlinear problems can be solved by linearizing them. To see what is going on here we look at a simple example.

11.5.2.4 Example. Consider the function

$$p(u) = 3\,u^2 + 1.$$

on [0,1] which has blossom

$$p(u_1, u_2) = 3\,u_1u_2 + 1.$$

Suppose that we want to compute p(1/3). If D is the diagonal map D(u) = (u,u), then p(u) = P(D(u)). See Figure 11.22. Let **A**, **B**, . . . be the points as indicated in the figure.

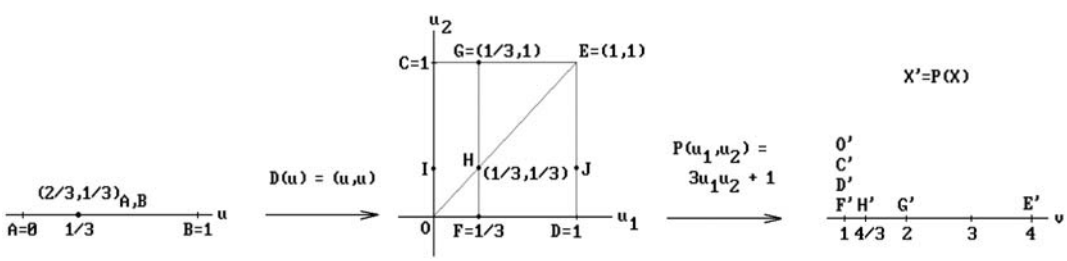


Figure 11.22. How blossoming linearizes maps.

Let $(a,b)_{\mathbf{X},\mathbf{Y}}$ denote the point $a\mathbf{X} + b\mathbf{Y}$ whose barycentric coordinates with respect to the points $\mathbf{X}$ and $\mathbf{Y}$ are (a,b). Since P is linear when restricted to vertical and horizontal lines,

$$\begin{aligned}
\mathbf{F} &= (2/3, 1/3)_{\mathbf{O},\mathbf{D}} \to \mathbf{F}' = P(\mathbf{F}) = (2/3, 1/3)_{O',D'} \\
\mathbf{G} &= (2/3, 1/3)_{\mathbf{C},\mathbf{E}} \to \mathbf{G}' = P(\mathbf{G}) = (2/3, 1/3)_{\mathbf{C}',\mathbf{E}'} \\
\mathbf{H} &= (2/3, 1/3)_{\mathbf{F},\mathbf{G}} \to \mathbf{H}' = P(\mathbf{H}) = (2/3, 1/3)_{\mathbf{F}',\mathbf{G}'}
\end{aligned}$$

and

$$\begin{aligned}
P(\mathbf{F}) &= (2/3)P(\mathbf{O}) + (1/3)P(\mathbf{D}) = (2/3)\,1 + (1/3)\,1 = 1 \\
P(\mathbf{G}) &= (2/3)P(\mathbf{C}) + (1/3)P(\mathbf{E}) = (2/3)\,1 + (1/3)\,4 = 2 \\
P(\mathbf{H}) &= (2/3)P(\mathbf{F}) + (1/3)P(\mathbf{G}) = (2/3)\,1 + (1/3)\,2 = 4/3.
\end{aligned}$$

It follows that we can compute p(1/3) by applying the linear condition to P three times, once each to compute $P(\mathbf{F})$ and $P(\mathbf{G})$ with respect to the segments $[\mathbf{O},\mathbf{D}]$ and $[\mathbf{C},\mathbf{E}]$, respectively, and then once to the segment $[\mathbf{F},\mathbf{G}]$. Note that because of the symmetry of the function P we could have also used the segments $[\mathbf{O},\mathbf{C}]$, $[\mathbf{D},\mathbf{E}]$, and $[\mathbf{I},\mathbf{J}]$. Note further that P is not linear on the diagonal segment $[\mathbf{O},\mathbf{E}]$. For example, $\mathbf{H} = (2/3,1/3)_{\mathbf{O},\mathbf{E}}$ but $(2/3,1/3)_{\mathbf{O}',\mathbf{E}'} = 2 \neq \mathbf{H}'$.

Next, suppose that P is the blossom of a polynomial function p as in Theorem 11.5.2.1 and that s and t are fixed **distinct** real numbers. If $u \in \mathbf{R}$, then let (a_0,a_1) be the barycentric coordinates of u with respect to s and t, that is, a_0 and a_1 satisfy

$$u = a_0 s + a_1 t \quad \text{and} \quad a_0 + a_1 = 1.$$

It is easy to check that

$$a_0 = \frac{t-u}{t-s} \quad \text{and} \quad a_1 = \frac{u-s}{t-s}.$$

Define

$$\mathbf{b}_i^r(u) = P(\underbrace{u,\ldots,u}_{r}, \underbrace{s,\ldots,s}_{d-r-i}, \underbrace{t,\ldots,t}_{i}). \tag{11.88}$$

Notice that

$$p(u) = P(u,\ldots,u) = \mathbf{b}_0^d(u).$$

In the computations that follow, keep in mind that since P is a symmetric function **we can permute its parameters** in any way we want without changing its value. Expanding the first parameter gives

$$\begin{aligned}\mathbf{b}_0^d(u) &= a_0P(s,u,\ldots,u)+a_1P(t,u,\ldots,u)\\ &= a_0\mathbf{b}_0^{d-1}(u)+a_1\mathbf{b}_1^{d-1}(u).\end{aligned}$$

Similarly, one can show that

$$\mathbf{b}_i^r(u) = a_0\mathbf{b}_i^{r-1}(u)+a_1\mathbf{b}_{i+1}^{r-1}(u), \quad \text{for } 0 \le i \le d-r \text{ and } 0 \le r \le d. \tag{11.89}$$

Let

$$\mathbf{b}_i = \mathbf{b}_i^0(u) = P(\underbrace{s,\ldots,s}_{d-i},\underbrace{t,\ldots,t}_{i}) \tag{11.90}$$

and

$$B_{i,d}(u) = \binom{d}{i}a_1^i(1-a_1)^{d-i} = \binom{d}{i}\frac{(u-s)^i(t-u)^{d-i}}{(t-s)^i(t-s)^{d-i}}. \tag{11.91}$$

11.5.2.5 Theorem.

$$p(u) = \sum_{i=0}^{d} B_{i,d}(u)\mathbf{b}_i. \tag{11.92}$$

Proof. Use the fact that $p(u) = \mathbf{b}_0^d(u)$ and the recursive formula for the $\mathbf{b}_i^r$. Basically, all that one has done is expand each of the u parameters in P. See [Seid89].

Definition. Equation (11.92) is called the *Bézier representation* of the polynomial p(u). The points $\mathbf{b}_i$ are called the *poles* or *Bézier points* or *control points* of p(u). The functions $B_{i,d}(u)$ are called the *Bézier polynomials* for p(u).

The reason for the terminology is obvious because the curve p(u) is a generalization of the Bézier curves that were studied in Section 11.4.

Returning to Example 11.5.2.4 and Figure 11.22, letting [s,t] = [0,1] and u = 1/3, we get that $(a_0,a_1) = (2/3,1/3)$ and

$$\begin{aligned}
\mathbf{b}_0^0(u) &= P(s,s) = P(0,0) = \mathbf{O}' = 1\\
\mathbf{b}_1^0(u) &= P(s,t) = P(0,1) = P(1,0) = \mathbf{C}' = \mathbf{D}' = 1\\
\mathbf{b}_2^0(u) &= P(t,t) = P(1,1) = \mathbf{E}' = 4\\
\mathbf{b}_0^1(u) &= P(u,s) = P(1/3,0) = \mathbf{F}' = (2/3)\mathbf{b}_0^0(u)+(1/3)\mathbf{b}_1^0(u)\\
\mathbf{b}_1^1(u) &= P(u,t) = P(1/3,1) = \mathbf{G}' = (2/3)\mathbf{b}_1^0(u)+(1/3)\mathbf{b}_2^0(u).
\end{aligned}$$

Our control points are just **O′**, **D′**, and **E′** and the image of the map lies in their convex hull [1,4].

More generally, expanding two of the u in an arbitrary blossom P leads to

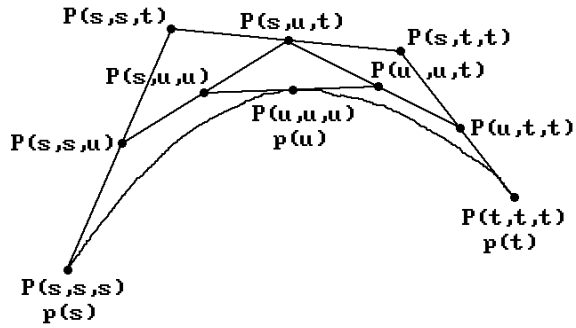


Figure 11.23. The geometry of the Bézier polynomial construction.

$$
\begin{aligned}
p(u) &= P(u,u,\ldots,u) \\
&= a_0 P(s,u,u,\ldots u) + a_1 P(t,u,u,\ldots u) \\
&= a_0(a_0 P(s,u,u,\ldots,u) + a_1 P(s,t,u,\ldots,u)) \\
&\quad + a_1(a_0 P(t,s,u,\ldots,u) + a_1 P(t,t,u,\ldots,u)) \\
&= (1-a_1)^2 P(s,s,u,\ldots,u) + 2a_1(1-a_1)P(s,t,u,\ldots,u) + a_1^2 P(t,t,u,\ldots,u). \qquad (11.93)
\end{aligned}
$$

Figure 11.23 shows the geometry in all this for a cubic polynomial curve in the plane. Recall that we can permute parameters, so that

$$P(u,t,t,\ldots) = P(t,t,u,\ldots) \quad \text{and} \quad P(s,u,t,\ldots) = P(s,t,u,\ldots).$$

Again, the reader needs to stare at Figure 11.23 and match its geometry with equations (11.93) until the relationship is clear. The mathematics in this section is basically simple, but it may look complicated because it is easy to get lost in the subscripts and parameters.

The recursive formula for the $\mathbf{b}_i^r(u)$ in (11.89) leads to the recursive algorithm for computing p(u) from its control points $\mathbf{b}_i$ shown in Algorithm 11.5.2.1. Note how this algorithm generalizes the corresponding Algorithm 11.4.1 in Section 11.4. In particular, letting d = n, s = 0, and t = 1, equations (11.51) and (11.92) define the same curve. See also Exercise 11.5.2.1.

There is also a formula for the derivatives of p(u) using forward differences. Forward difference methods lead to fast evaluation of polynomials at uniformly spaced intervals. They are discussed in most books on numerical analysis. Wallis ([Wall90]) has a brief tutorial.

Definition. For any sequence of points $\mathbf{b}_i$, define the *rth forward difference operator* Δ^r as follows:

$$
\begin{aligned}
\Delta^0 \mathbf{b}_i &= \mathbf{b}_i \\
\Delta^r \mathbf{b}_i &= \Delta^{r-1}\mathbf{b}_{i+1} - \Delta^{r-1}\mathbf{b}_i, \; r > 0.
\end{aligned}
$$

The operator Δ^1 is usually abbreviated to Δ.

Inputs: Control points $\mathbf{b}_i$, $i = 0,1, \ldots ,d$ in $\mathbf{R}^m$
Fixed **distinct** real numbers s and t
$u \in \mathbf{R}$
Output: p(u) as defined by equation (11.92)

Let $u = a_0 s + a_1 t$, where $a_0 + a_1 = 1$.

Step 1: Set $\mathbf{b}_i^0 = \mathbf{b}_i$.
Step 2: For $r = 1,2, \ldots ,d$ and $i = 0,1, \ldots ,d-r$ compute

$$\mathbf{b}_i^r = a_0 \mathbf{b}_i^{r-1} + a_1 \mathbf{b}_{i+1}^{r-1} .$$

When one has finished, $\mathbf{b}_0^d = p(u)$.

Algorithm 11.5.2.1. The de Casteljau algorithm.

11.5.2.6 Theorem. The rth derivative of the Bézier curve p(u) defined by equation (11.92) is given by

$$\frac{d^r}{du^r} p(u) = \frac{d!}{(d-r)!} \Delta^r \mathbf{b}_0^{d-r}(u).$$

Proof. See [Fari97]. To show the reader some more sample computations with blossoms, we will work through the main steps for the case r = 1. Let $P(u_1, \ldots ,u_d)$ be the blossom of p(u). To simplify the notation we assume that [s,t] = [0,1]. Then

$$\begin{aligned}
P(u_1, \ldots, u_i + h, \ldots, u_d) &= (1 - (u_i + h))P(u_1, \ldots, u_{i-1}, 0, u_{i+1}, \ldots, u_d) \\
&\quad + (u_i + h)P(u_1, \ldots, u_{i-1}, 1, u_{i+1}, \ldots, u_d) \\
&= (1 - u_i)P(u_1, \ldots, u_{i-1}, 0, u_{i+1}, \ldots, u_d) \\
&\quad + u_i P(u_1, \ldots, u_{i-1}, 1, u_{i+1}, \ldots, u_d) \\
&\quad + h[P(u_1, \ldots, u_{i-1}, 1, u_{i+1}, \ldots, u_d) - P(u_1, \ldots, u_{i-1}, 0, u_{i+1}, \ldots, u_d)] \\
&= P(u_1, \ldots, u_{i-1}, u_i, u_{i+1}, \ldots, u_d) \\
&\quad + h[P(u_1, \ldots, u_{i-1}, 1, u_{i+1}, \ldots, u_d) - P(u_1, \ldots, u_{i-1}, 0, u_{i+1}, \ldots, u_d)].
\end{aligned}$$

This formula shows that

$$\begin{aligned}
\frac{\partial P}{\partial u_i} &= \lim_{h \to 0} \frac{P(u_1, \ldots, u_i + h, \ldots, u_d) - P(u_1, \ldots, u_i, \ldots, u_d)}{h} \\
&= P(u_1, \ldots, 1, \ldots, u_d) - P(u_1, \ldots, 0, \ldots, u_d).
\end{aligned}$$

Finally, using the chain rule and the fact that blossoms are symmetric functions we get

$$p'(u) = d[P(1,u,\ldots,u) - P(0,u,\ldots,u)] = d[\mathbf{b}_1^{d-1} - \mathbf{b}_0^{d-1}],$$

which is the case r = 1 of the theorem.

Theorem 11.5.2.6 generalizes the formula in equation (11.57).

Next, let us see where an analysis using blossoms leads when applied to the B-spline functions $N_{i,k}(u)$ given the nondecreasing knot sequence

$$t_0 = t_1 = \ldots = t_{k-1}, t_k, \ldots, t_n, t_{n+1} = t_{n+2} = \ldots = t_{n+k}.$$

Let $N_{i,k}^j(u)$ denote the restriction of $N_{i,k}(u)$ to the interval $I_j = [t_j, t_{j+1})$. The function $N_{i,k}^j(u)$ is a polynomial of degree k – 1. Let $n_{i,k}^j$ be its blossom.

11.5.2.7 Theorem. The functions $n_{i,r}^j(\mathbf{u})$ satisfy the recurrence relation

$$\begin{aligned} n_{i,1}^j(\ ) &= \delta_{ij}, \\ n_{i,r}^j(u_1,u_2,\ldots,u_{r-1}) &= \frac{u_{r-1}-t_i}{t_{i+r-1}-t_i} n_{i,r-1}^j(u_1,u_2,\ldots,u_{r-2}) \\ &\quad + \frac{t_{i+r}-u_r}{t_{i+r}-t_{i+1}} n_{i+1,r-1}^j(u_1,u_2,\ldots,u_{r-2}), \quad 2 \le r \le k. \end{aligned} \tag{11.94}$$

Proof. The theorem is proved by induction. One shows that the relation defines symmetric multiaffine maps and that the diagonals clearly agree with the $N_{i,r}$. See [Seid89].

11.5.2.8 Corollary. If $t_j < t_{j+1}$ and $j-r+1 \le \ell \le j$, then

$$n_{i,r}^j(t_{\ell+1}, t_{\ell+2}, \ldots, t_{\ell+r-1}) = \delta_{i\ell}.$$

Proof. One uses the formulas in Theorem 11.5.2.7 and induction on r. See [Seid89].

Corollary 11.5.2.8 leads to an important algorithm that relates the control points of a B-spline to associated blossoms. Let

$$p(u) = \sum_{i=0}^{n} N_{i,k}(u)\mathbf{p}_i \tag{11.95}$$

be a B-spline curve of order k. Since p(u) is a polynomial over each knot interval, let $p_j(u)$ be the polynomial which defines p(u) over the interval $I_j = [t_j, t_{j+1}]$ and let P_j be its blossom.

11.5.2.9 Theorem. If $t_j < t_{j+1}$ and $j-k+1 \le \ell \le j$, then the *de Boor point* $\mathbf{p}_\ell$ is defined by

$$\mathbf{p}_\ell = P_j(t_{\ell+1}, t_{\ell+2}, \ldots, t_{\ell+k-1}). \tag{11.96}$$

Proof. It follows from Theorem 11.5.1.4(1) that

$$p_j(u) = \sum_{i=j-k+1}^{j} N_{i,k}^{j}(u)\mathbf{p}_i,$$

so that

$$P_j(u_1, u_2, \ldots, u_{k-1}) = \sum_{i=j-k+1}^{j} n_{i,k}^{j}(u_1, u_2, \ldots, u_{k-1})\mathbf{p}_i.$$

Corollary 11.5.2.8 now implies the result. See [Seid89].

11.5.2.10 Theorem. Let p(u) be a B-spline of order k with knot vector $(t_0, t_1, \ldots t_{n+k})$ and control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$. If $t_j \le u < t_{j+1}$, then Algorithm 11.5.2.2 computes p(u).

Proof. See [Seid89]. The function

$$P_j(t_{i+1}, \ldots, t_j, \underbrace{u, \ldots, u}_{r}, t_{j+1}, \ldots, t_{i+k-r-1})$$

satisfies the same recurrence relation as $\mathbf{p}_i^r$ and so the two quantities must be equal. Now use Theorem 11.5.2.9.

Notice the similarity in the recurrence relations in Algorithm 11.5.2.1 and 11.5.2.2. The only real difference is in the coefficients (barycentric coordinates) of the points.

Input: A B-spline curve p(u) of order k with knots $t_0, t_1, \ldots t_{n+k}$
$t_j \le u < t_{j+1}$
Output: p(u) as defined by equation (11.95)

Step 1: Set $\mathbf{p}_i^0 = \mathbf{p}_i$.
Step 2: For $r = 1, 2, \ldots, k-1$ and $i = j-k+1+r, \ldots, j$ compute

$$\mathbf{p}_i^r = \left(1 - \frac{u - t_i}{t_{i+k-r} - t_i}\right)\mathbf{p}_{i-1}^{r-1} + \frac{u - t_i}{t_{i+k-r} - t_i}\mathbf{p}_i^{r-1}.$$

When one has finished, $\mathbf{p}_j^{k-1} = p(u)$.

Algorithm 11.5.2.2. The de Boor Algorithm.

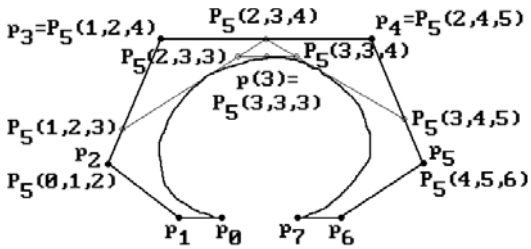


Figure 11.24. The de Boor algorithm for B-splines using blossoms.

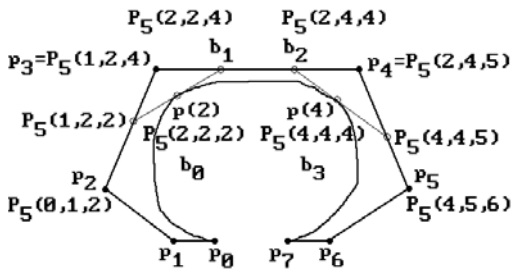


Figure 11.25. Computing Bézier points from the de Boor points.

Figure 11.24 shows how Algorithm 11.5.2.2 computes the value of the cubic B-spline p(u) with knot vector (0,0,0,0,1,2,4,5,6,6,6,6) at u = 3. In that case, i = 5 and p(3) = $P_5(3,3,3)$.

There is more geometry hidden in the formalism above. Given a blossom we can use equations (11.90) and (11.96) to define the Bézier and de Boor points, respectively. In other words, we have a way to switch between a Bézier and a B-spline representation for a B-spline curve. We show how this works with an example.

11.5.2.11 Example. Let p(u) be a cubic B-spline p(u) with knot vector (t_i) = (0,0,0,0,1,2,4,5,6,6,6,6) and consider the interval [2,4]. Using the notation of Theorem 11.5.2.9 that interval corresponds to the values j = 5, k = 4, and $2 \le \ell \le 5$. In equation (11.90) we would have d = 3. Using the blossom of the curve p(u) over [2,4] = I_5 = $[t_5,t_6]$ we can compute both the associated de Boor points $\mathbf{p}_2 = P_5(0,1,2)$, $\mathbf{p}_3 = P_5(1,2,4)$, $\mathbf{p}_4 = P_5(2,4,5)$, and $\mathbf{p}_5 = P_5(4,5,6)$ and the Bézier points $\mathbf{b}_0 = P_5(2,2,2)$, $\mathbf{b}_1 = P_5(2,2,4)$, $\mathbf{b}_2 = P_5(2,4,4)$, and $\mathbf{b}_3 = P_5(4,4,4)$. See Figure 11.25. From this it is easy to see the relationship between the Bézier and de Boor points. Consider the three points $\mathbf{p}_3 = P_5(1,2,4) = P_5(t_4,t_5,t_6)$, $\mathbf{b}_1 = P_5(2,2,4) = P_5(t_5,t_5,t_6)$, and $\mathbf{p}_4 = P_5(5,2,4) = P_5(t_7,t_5,t_6)$. Using barycentric coordinates and the linearity of P_5 with respect to its first coordinate, we see that

$$\mathbf{b}_1 = \frac{t_7 - t_5}{t_7 - t_4}\mathbf{p}_3 + \frac{t_5 - t_4}{t_7 - t_4}\mathbf{p}_4 = (3/4)\mathbf{p}_3 + (1/4)\mathbf{p}_4.$$

Similarly, $\mathbf{b}_2 = P_5(4,2,4) = P_5(t_6,t_5,t_6)$, and

$$\mathbf{b}_2 = \frac{t_7 - t_6}{t_7 - t_4}\mathbf{p}_3 + \frac{t_6 - t_4}{t_7 - t_4}\mathbf{p}_4 = (1/4)\mathbf{p}_3 + (3/4)\mathbf{p}_4.$$

In particular, the points $\mathbf{p}_3$, $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{p}_4$ are collinear. Furthermore, the point $\mathbf{b}_1$ divides the segment $[\mathbf{p}_3,\mathbf{p}_4]$ in the same proportion as t_5 divides the interval $[t_4,t_7]$. Similarly, the point $\mathbf{b}_2$ divides the segment $[\mathbf{p}_3,\mathbf{p}_4]$ in the same proportion as t_6 divides the interval $[t_4,t_7]$.

The computations in Example 11.5.2.11 easily generalize. There was nothing special about our knots and j = 5. Consider an arbitrary cubic B-spline p(u) with knots t_j and control points $\mathbf{p}_i$. With respect to $[t_j,t_{j+1}]$, $t_j < t_{j+1}$, we have

$$\mathbf{p}_{j-2} = P_j(t_{j-1}, t_j, t_{j+1}) \quad \text{and} \quad \mathbf{p}_{j-1} = P_j(t_{j+1}, t_j, t_{j+1}).$$

Define points

$$\mathbf{b}_{3j-6} = P_j(t_j, t_j, t_j), \quad \mathbf{b}_{3j-5} = P_j(t_j, t_j, t_{j+1}),$$
$$\mathbf{b}_{3j-4} = P_j(t_j, t_{j+1}, t_{j+1}), \quad \mathbf{b}_{3j-3} = P_j(t_{j+1}, t_{j+1}, t_{j+1}).$$

Then we can easily show like in Example 11.5.2.11 that

$$\mathbf{b}_{3j-5} = \frac{t_{j+2} - t_j}{t_{j+2} - t_{j-1}}\mathbf{p}_{j-2} + \frac{t_j - t_{j-1}}{t_{j+2} - t_{j-1}}\mathbf{p}_{j-1}$$

and

$$\mathbf{b}_{3j-4} = \frac{t_{j+2} - t_{j+1}}{t_{j+2} - t_{j-1}}\mathbf{p}_{j-2} + \frac{t_{j+1} - t_{j-1}}{t_{j+2} - t_{j-1}}\mathbf{p}_{j-1}.$$

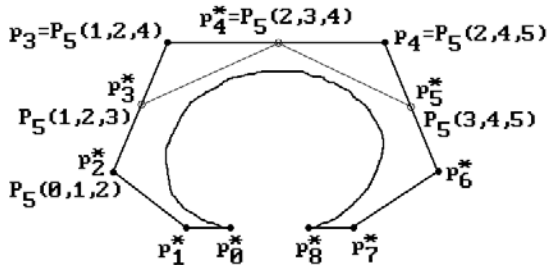
11.5.2.12 Theorem. If $t_j \le u \le t_{j+1}$, then the cubic curve traced out by the function p(u) defined by equations (11.95) and the curve traced out by the Bézier curve p(u) defined on [0,1] by equation (11.49) for control points $\mathbf{b}_{3j-6}$, $\mathbf{b}_{3j-5}$, $\mathbf{b}_{3j-4}$, and $\mathbf{b}_{3j-3}$ are the same.

Proof. This is an easy consequence of the validity of the de Casteljau and the de Boor algorithms.

Theorem 11.5.2.12 can be rephrased as saying that each cubic B-spline curve can be thought of as a collection of cubic Bézier curves. Comparing control points, each knot interval contributes two de Boor points and four Bézier points under this correspondence.

The de Boor algorithm also shows us how to insert knots. The reason one might want to insert knots is to allow more flexibility in subsequent manipulations. As usual, assume that we have a B-spline p(u) of order k with knot vector $K = (t_0,t_1, \ldots t_{n+k})$ defined by equation (11.95) and suppose that we want to add a new knot t, where $t \in (t_h,t_{h+1})$. Let $K^* = (t_0,t_1, \ldots, t_h,t,t_{h+1}, \ldots, t_{n+k})$ be the new knot vector and let $N^*_{i,k}(u)$ be the spline functions defined recursively by equations (11.69) but with respect to the new knot vector K*. We want to find new control points $\mathbf{p}^*_i$ so that

Figure 11.26. Boehm knot insertion.



$$p(u) = \sum_{i=0}^{n+1} N_{i,k}^{*}(u)\mathbf{p}_i^{*}. \tag{11.97}$$

11.5.2.13 Theorem. The new control points for equation (11.97) are defined by

$$\mathbf{p}_i^{*} = a_i\mathbf{p}_i + (1-a_i)\mathbf{p}_{i-1},$$

where

$$\begin{aligned} a_i &= 1, && 0 \le i \le h-k+1 \\ &= \frac{t-t_i}{t_{i+k-1}-t_i}, && h-k+2 \le i \le h, \\ &= 0, && h+1 \le i \le n+1 \end{aligned}$$

Proof. See [Boeh80] or [Seid89]. Figure 11.26 shows the idea behind the proof. We have inserted the knot t = 3 into the knot vector (0,0,0,0,1,2,4,5,6,6,6,6) used in Figure 11.25.

Inserting knots does not change the shape of the curve but increases the number of control points; however, in contrast to Bézier curves we do not raise the degree of the curve by doing this.

Theorem 11.5.2.13 shows how to insert a single knot. Sometimes one wants to insert more than one knot at a time into a knot vector. The next theorem shows how to do that. We again assume that we have a B-spline p(u) with knot vector $K = (t_0, t_1, \ldots t_{n+k})$, but this time we want to replace K with a new knot vector $K^* = (s_0, s_1, \ldots, s_{m+k})$, $m \ge n$. If $N_{i,k}^{*}(u)$ are the spline functions associated to K^*, we want to find the new set of control points $\mathbf{p}_j^{*}$, so that

$$p(u) = \sum_{j=0}^{m} N_{j,k}^{*}(u)\mathbf{p}_j^{*}. \tag{11.98}$$

11.5.2.14 Theorem. (The Oslo Algorithm) The control points for equation (11.98) are defined by

$$\mathbf{p}_j^{*} = \sum_{i=0}^{n} a_{i,j}^{k}\mathbf{p}_i, \quad 0 \le j \le m,$$

where the $a^k_{i,j}$ are defined recursively by

$$a^1_{i,j} = 1, \quad \text{if } t_i \le s_j \le t_{i+1},$$
$$= 0, \quad \text{elsewhere.}$$
$$a^r_{i,j} = \frac{s_{j+k-1} - t_i}{t_{i+k-1} - t_i} a^{r-1}_{i,j} + \frac{t_{i+k} - s_{j+k-1}}{t_{i+k} - t_{i+1}} a^{r-1}_{i+1,j}, \quad r > 1.$$

Proof. See [CoLR80].

We finish this section with several other theorems that follow easily from the multiaffine map approach to splines. First of all, an important fact that drops out of the formalism is the differentiability of the functions $N_{i,k}(u)$, which is not totally obvious from their recursive definitions.

11.5.2.15 Theorem. If $t = t_{j+1} = t_{j+2} = \ldots = t_{j+m}$ is a knot of multiplicity $m \le k$ for $N_{i,k}(u)$, then $N_{i,k}(u)$ is C^{k-1-m} at t.

Proof. See [Seid89].

11.5.2.16 Theorem. (Curry-Schoenberg Theorem) All splines are linear combinations of B-splines.

Proof. See [Seid89].

One can use Theorems 11.5.2.12 and 11.5.2.15 to find the Bézier control points of a spline of order k. Simply keep inserting knots until all have multiplicity k – 1. At that point the de Boor points reduce to the Bézier points.

11.5.2.17 Theorem. (Variation diminishing property) A plane (line in planar case) intersects a B-spline in no more points than it intersects the control polygon.

Proof. See [LanR83], [Seid89], or [PieT95]. In particular, this theorem applies to Bézier curves.

One important point about **all** the results in this section is that the proofs are very short and straightforward. The reader should have little trouble filling in those that are omitted.

11.5.3 Rational B-spline Curves

Although B-splines curves represent a very large class of curves, they are unable to represent some very simple curves **exactly**. It is easy to show that conics like circles and ellipses cannot be represented by polynomial curves and so B-spline curves can only approximate them. This is a drawback because conics are curves that one often wants to represent.

Fortunately, all is not lost. Conics can be represented by rational curves via a simple trick. We show how this works in the case of a circle. Consider the circle of

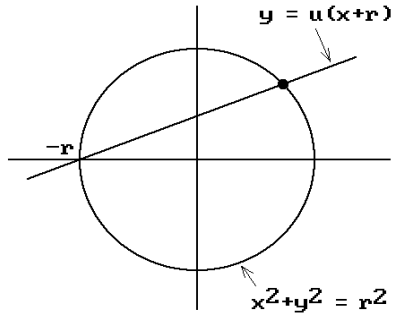


Figure 11.27. Defining a rational parameterization of the circle.

radius r shown in Figure 11.27. Every nonvertical line through (–r,0) can be parameterized by its slope u and satisfies the equation

$$y = u(x + r). \tag{11.99}$$

Solving for the intersection of this line and the circle

$$x^2 + y^2 = r^2$$

leads to the solution

$$x = \frac{r(1-u^2)}{1+u^2} \quad \text{and} \quad y = \frac{2ru}{1+u^2}$$

and the parameterization

$$u \to \left(\frac{r(1-u^2)}{1+u^2}, \frac{2ru}{1+u^2} \right). \tag{11.100}$$

Another argument for showing that conics have rational parameterizations comes about by using projective geometry and homogeneous coordinates. It is a well-known fact (Theorem 3.6.1.1 in [AgoM04]) that all conics are projectively equivalent. In fact, every conic **X** in the plane z = 1 in $\mathbf{R}^3$ is the central projection of a parabola **Y** in some other plane. See Figure 11.28. Furthermore, a parabola is the only conic that has a polynomial parameterization. Now the standard parabola $y = x^2$ in $\mathbf{R}^2$ can be parameterized by $u \to (u,u^2)$, so that our parabola **Y** can be parameterized by a quadratic curve

$$u \to (x(u), y(u), z(u)) \tag{11.101}$$

since it is obtained from the standard one by a linear change of variables and such a transformation does not change the degree of the parametrization. It follows that the conic **X** has a rational parametrization of the form

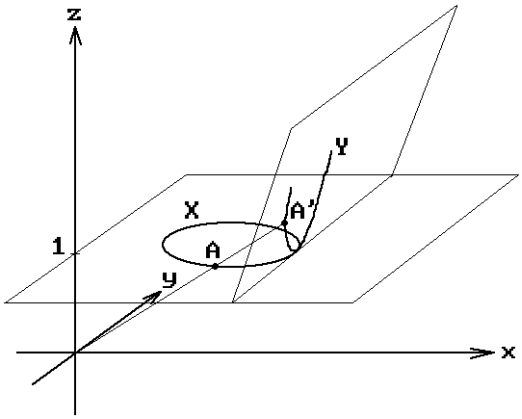


Figure 11.28. Projectively equivalent ellipse and parabola.

$$p(u) = \left(\frac{x(u)}{z(u)}, \frac{y(u)}{z(u)} \right)$$

because the central projection is gotten simply by dividing by the z-coordinate. But we can think of equation (11.101) as defining a conic with homogeneous coordinates in projective space $\mathbf{P}^2$. The important observation is then that conics do have polynomial representations if we use homogeneous coordinates.

In summary, we have shown that we can handle a larger class of curves if we use homogeneous coordinates. In that setting, the analog of equation (11.67) is

$$P(u) = \sum_{i=0}^{n} b_i(u)\mathbf{P}_i, \tag{11.102}$$

where the $b_i(u)$ are suitable basis or blending functions and the $\mathbf{P}_i$ are points described with homogeneous coordinates. Everything we did earlier for polynomial curves applies to the curves defined by equation (11.102) since the nature of the coordinates did not play a role. In particular, we have the obvious notions of Bézier and B-spline curves for homogeneous coordinates. Furthermore, if we write $\mathbf{P}_i$ in the form $\mathbf{P}_i = (x_iw_i, y_iw_i, z_iw_i, w_i)$, then the projective space curve defined by P(u) projects to the curve

$$p(u) = \frac{\sum_{i=0}^{n} w_i b_i(u)\mathbf{p}_i}{\sum_{i=0}^{n} w_i b_i(u)} \tag{11.103}$$

where $\mathbf{p}_i = (x_i, y_i, z_i)$. There are several important special cases of such curves.

Definition. The curve p(u) defined by equation (11.103) is called a *rational Bézier curve* if its domain is [0,1] and $b_i(u) = B_{i,n}(u)$. (The $B_{s,t}(u)$ are the functions defined by equation (11.50).) The curve p(u) is called a *rational B-spline curve of order k* if the

$b_i(u)$ are B-splines of order k. The curve p(u) is called a ***n**onuniform **r**ational **B**-**s**pline (NURBS) curve of order k* with domain [a,b] if $b_i(u) = N_{i,k}(u)$ with respect to a knot vector

$$U = (u_i) = (\underbrace{a, \ldots, a}_{k}, u_k, u_{k+1}, \ldots, u_n, \underbrace{b, \ldots, b}_{k}).$$

(The $N_{i,k}(u)$ are the B-splines defined by equations (11.69).) In any case, the points $\mathbf{p}_i$ are called the *control points* of the curve p(u) and the numbers w_i are called its *weights*.

The ordinary Bézier and B-spline curves are clearly a special case of the rational ones since we get them by using weights that are all equal to 1. Note further that if we define the function $R_i(u)$ by

$$R_i(u) = \frac{w_i b_i(u)}{\sum_{j=0}^{n} w_j b_j(u)}, \tag{11.104a}$$

then

$$p(u) = \sum_{i=0}^{n} R_i(u)\mathbf{p}_i, \tag{11.104b}$$

so that p(u) is again a curve of a form (like that of equation (11.67)) that we have seen many times before.

Definition. The functions $R_i(u)$ in equations (11.104) are called the *rational basis functions* for the curve p(u).

NURBS curves (and surfaces) have become very popular in recent years and a number of modeling programs are based on them. Some general references for these and rational Bézier curves are [PieT95], [Pieg91], [Fari95], [Roge01], or [RogA90]. In the rest of this section we shall look at some examples and properties of NURBS curves, leaving a discussion of how to compute them efficiently to the next section.

11.5.3.1 Example. Suppose that we want to find a NURBS representation for the unit circle.

Solution. Consider the first quadrant of the unit circle. By equation (11.100) we have the rational parameterization

$$p(u) = \left(\frac{1-u^2}{1+u^2}, \frac{2u}{1+u^2}\right), \quad \text{for} \quad u \in [0,1].$$

In homogeneous coordinates this can be written as

$$\begin{aligned} P(u) &= \left(1-u^2, 2u, 0, 1+u^2\right) \\ &= (1,0,0,1) + u(0,2,0,0) + u^2(-1,0,0,1). \end{aligned} \tag{11.105}$$

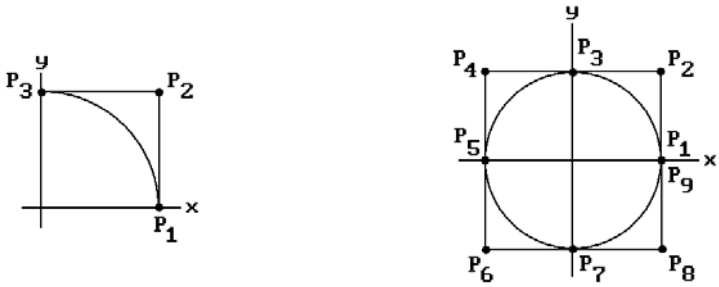


Figure 11.29. The circle as a rational B-spline.

The Bézier approach to describing this curve is to look for three homogeneous control points $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ so that

$$\begin{aligned} P(u) &= (1-u^2)\mathbf{P}_1 + 2u(1-u)\mathbf{P}_2 + u^2\mathbf{P}_3 \\ &= \mathbf{P}_1 + 2\,u(\mathbf{P}_2 - \mathbf{P}_1) + u^2(\mathbf{P}_3 - 2\mathbf{P}_2 + \mathbf{P}_1). \end{aligned} \tag{11.106}$$

Equating the coefficients of the u's in the two equations (11.105) and (11.106) for P(u) gives that

$$\mathbf{P}_1 = (1,0,0,1), \quad \mathbf{P}_2 = (1,1,0,1), \quad \text{and} \quad \mathbf{P}_3 = (0,2,0,2).$$

The corresponding $\mathbf{p}_i$ are shown in Figure 11.29(a). Using these $\mathbf{P}_i$ as control points and the knot vector (0,0,0,1,1,1) gives us the NURBS curve that describes the first quadrant of the unit circle. A NURBS representation for the second quadrant can easily be obtained from this one by rotating the control points about the y-axis by 180 degrees. Alternatively, reparameterizing to [1,2], a parameterization q(u) for this second quadrant is

$$q(u) = \left(\frac{u^2-4u+3}{u^2-4u+5}, \frac{-2(u-2)}{u^2-4u+5}\right), \quad \text{for} \quad u \in [1,2],$$

and we can solve for the new control points as before. At any rate, the new control points are

$$\mathbf{P}_3 = (0,2,0,2), \quad \mathbf{P}_4 = (-1,1,0,1), \quad \text{and} \quad \mathbf{P}_5 = (-1,0,0,1)$$

Finally, rotating our control points by 180 degrees about the x-axis gives us the complete NURBS representation for the whole unit circle. It is easy to check that the parameterization can be written in the form

$$p(u) = \frac{\sum_{i=1}^{9} w_i N_{i,3}(u)\mathbf{p}_i}{\sum_{i=1}^{9} w_i N_{i,3}(u)},$$

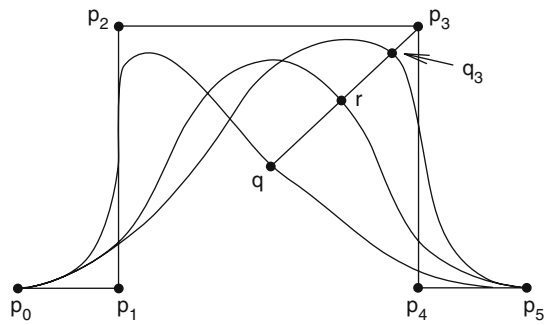


Figure 11.30. The geometric interpretation of rational B-spline weights.

where the weight sequence $(w_1, w_2, \ldots, w_9)$ is (1,1,2,1,1,1,2,1,1) and the knot vector is (0,0,0,1,1,2,2,3,3, 4,4,4). The points $\mathbf{p}_i$ are shown in Figure 11.29(b).

Although Example 11.5.3.1 found a NURBS representation for the unit circle, it is not a good one because it does not distribute points uniformly along the circle. The problem is with the rational function parameterization that we used as a starting point. [Till83] shows how one can get a better parameterization by a rational transformation of the form

$$u = \frac{at + b}{ct + b}.$$

Next, there is a geometric interpretation of the weights. To emphasize the dependence of the function p(u) defined by equation (11.103) on its weights we shall include a reference to the weights in the parameters of the functions below along with any values that may have been assigned to them. See Figure 11.30, where

$$\mathbf{q} = p(u; w_i = 0), \quad \mathbf{r} = p(u; w_i = 1), \quad \text{and} \quad \mathbf{q}_i = p(u; w_i \neq 0 \text{ or } 1).$$

If

$$\alpha = R_i(u; w_i = 1) \quad \text{and} \quad \beta = R_i(u),$$

where the R_i are the rational basis functions of p(u), then one can show that

$$\mathbf{r} = (1-\alpha)\mathbf{q} + \alpha\, \mathbf{p}_i, \quad \text{and}$$
$$\mathbf{q}_i = (1-\beta)\mathbf{q} + \beta\, \mathbf{p}_i.$$

It follows that

$$w_i = \frac{|\mathbf{p}_i\mathbf{r}|}{|\mathbf{q}\mathbf{r}|} \Big/ \frac{|\mathbf{p}_i\mathbf{q}_i|}{|\mathbf{q}\mathbf{q}_i|} = \frac{1-\alpha}{\alpha} \Big/ \frac{1-\beta}{\beta}.$$

In other words, the weight w_i is just the cross-ratio of the four points $\mathbf{p}_i$, $\mathbf{q}$, $\mathbf{r}$, and $\mathbf{q}_i$. The following geometric facts can be proved:

(1) Increasing or decreasing w_i will increase or decrease β which pulls the curve toward $\mathbf{p}_i$ or pushes it away from $\mathbf{p}_i$, respectively.
(2) Increasing or decreasing w_i will push the curve away from $\mathbf{p}_j$ or pull it toward $\mathbf{p}_j$, $j \neq i$, respectively.
(3) The points $\mathbf{q}_i$ lie on the line segment $[\mathbf{q},\mathbf{p}_i]$.
(4) As $\mathbf{q}_i$ approaches $\mathbf{p}_i$, β approaches 1 and w_i approaches infinity.

We finish by listing a few of the important properties of NURBS curves.

11.5.3.2 Theorem. Let p(u) be a NURBS curve of order k with domain [0,1], knots u_i, control points $\mathbf{p}_i$, and weights $w_i > 0$.

(1) The rational basis functions $R_i(u)$ for p(u) satisfy $R_i(u) \geq 0$ and

$$\sum_{i=0}^{n} R_i(u) = 1.$$

(2) The curve p(u) interpolates the first and last point. More precisely, $p(0) = \mathbf{p}_0$ and $p(1) = \mathbf{p}_n$.

(3) (Local control) Changing the control point $\mathbf{p}_i$ or weight w_i only changes the formula for p(u) over the interval (u_i,u_{i+k}).

(4) (Projective invariance) If the curve p(u) is transformed by a projective transformation, the formula for the new curve is gotten simply by transforming the **homogeneous** control points (equation (11.102) and then projecting back to $\mathbf{R}^3$ to get another formula like equation (11.103).

(5) (Local convex hull property) The curve p(u) satisfies a strengthened convex hull property like the ordinary B-splines, namely, for each i, $p([u_i,u_{i+1}])$ is contained in the convex hull of the control points $\mathbf{p}_{i-k+1}, \mathbf{p}_{i-k+2}, \ldots, \mathbf{p}_i$.

(6) (Variation diminishing property) A plane (line in planar case) intersects the curve p(u) in no more points than it intersects the control polygon.

Proof. See [PieT95]. The projective invariance property is stronger than affine invariance. Ordinary B-splines are affinely invariant but not projectively invariant.

Finally, although rational Bézier and B-spline curves are defined as projections of ordinary Bézier and B-spline curves **in** $\mathbf{R}^4$ to the plane w = 1, it turns out that the associated correspondence between ordinary splines in $\mathbf{R}^4$ and ordinary splines in $\mathbf{R}^3$ is not as natural as one might want. For example, not every C^1 Bézier and B-spline curve p(u) in $\mathbf{R}^3$ with simple knots is a projection of a spline curve q(u) in $\mathbf{R}^4$ with simple knots. To find a curve q(u) that projects to p(u) we would have to allow q(u) to have multiple knots. See [Fari89] for a discussion of this and the condition that guarantees that a C^1 curve with simple knots is a projection of a C^1 curve with simple knots.

11.5.4 Efficient B-Spline and NURBS Curve Algorithms

As mentioned earlier, B-spline and NURBS curves are used a lot. Fortunately, although their definitions seem somewhat complicated and it is certainly more work than

dealing with simple polynomial functions, there are efficient algorithms so that using them in a modeler is not all that bad.

We start with the problem of computing a B-spline curve p(u) of order k and degree $m = k - 1$ with knots vector $U = \{u_0, u_1, \ldots, u_{n+k}\}$. Actually, because tangent vectors and derivatives of parametric curves are often needed in geometric modeling, we present an efficient algorithm that computes not only the value p(u) but also its derivatives up to any order d at the same time.

The first thing that we must do is find the nondegenerate span to which u belongs. This can be done using Algorithm 11.5.4.1. The function SpanIndex returns the index r so that $u \in [u_r, u_{r+1})$, where $u_r < u_{r+1}$, unless u is the right endpoint u_{n+1} of the domain of the curve and there might not be such an r, in which case we return n and will have $u \in (u_n, u_{n+1}]$. To avoid the special case one could also restrict the domain of the curve to $[u_m, u_{n+1} - \varepsilon]$, where ε is some small positive value.

Now assume that $u \in [u_r, u_{r+1})$, where $u_r < u_{r+1}$, or $u = u_{n+1}$ and $u \in (u_r, u_{r+1}]$. We know that

$$p(u) = \sum_{i=0}^{n} N_{i,k}(u)\mathbf{p}_i = \sum_{i=r-m}^{r} N_{i,k}(u)\mathbf{p}_i,$$

```
integer function SpanIndex (real array knots [0.. ]; integer n, m; real u)
{ Inputs:
    u_i = knots[i] ,  0 ≤ i ≤ n + m + 1  –  the knots
    n + 1 = number of control points
    m  –  the degree of the B-spline basis functions }
begin
    integer lo, hi, mid;

    if  (u = knots[n+1])  return n;

    { Now do a binary search of  u_m, u_m+1, … , u_n+1 }
    lo := m;      hi := n + 1;      mid := (lo + hi)/2;
    while  (u < knots[mid]) or (u >= knots[mid+1]  do
        begin
            if  (u < knots[mid])
                then  hi := mid
                else   lo := mid;
            mid := (lo + hi)/2;
        end;
    return mid;
end;
```

Algorithm 11.5.4.1. A B-spline span-finding algorithm.

because the coefficients of the other terms vanish. See Theorem 11.5.1.4. This expression only uses the knots $u_{r-m}, u_{r-m+1}, \ldots, u_{r+m+1}$. Let $N_{i,j} = N_{i,j}(u)$. The Cox-de Boor definition of the $N_{i,j}$ computes p(u) based on the following diagram, which shows the nonzero terms:

$$\begin{array}{ccccccc}
 & & & 0 & N_{r,1} & 0 & \\
 & & 0 & N_{r,2} & N_{r-1,2} & 0 & \\
 & 0 & N_{r,3} & N_{r-1,3} & N_{r-2,3} & 0 & \\
 & & & \cdot & & & \\
 & & & \cdot & & & \\
0 & N_{r,k} & N_{r-1,k} & \ldots & N_{r-k+2,k} & N_{r-k+1,k} & 0
\end{array}$$

Furthermore, to compute p(u) one just needs a one-dimensional array. Starting with some appropriate initial values, one can compute the final B-spline value by computing new values of the array from previous ones using the recursive formulas for the $N_{i,j}$. With regard to derivatives, by differentiating the $N_{i,k}(u)$ using formula (11.69b) and using a simple inductive argument it is easy to show that the following recursive formula holds for the first derivative:

$$\frac{d}{du}N_{i,k}(u) = \frac{k-1}{u_{i+k-1} - u_i}N_{i,k-1}(u) + \frac{k-1}{u_{i+k} - u_{i+1}}N_{i+1,k-1}(u).$$

Higher derivatives can be expressed similarly, with the dth derivative being a linear combination of the functions $N_{i+j,k-d}(u)$, $0 \le j \le d$. See [PieT95]. This means that both the value of the curve and its derivatives can be computed in a recursive fashion using a single array. In order to simplify the notation in the algorithm, we re-index the knots so that they become $u_0, u_1, \ldots, u_{2m+1}$. The control points $\mathbf{p}_{r-m}, \mathbf{p}_{r-m+1}, \ldots, \mathbf{p}_r$, are re-indexed with new index varying from 0 to m. This will put u into the interval $[u_m, u_{m+1})$ (or $(u_m, u_{m+1}]$ in the one special case). After the re-indexing, the procedure **Derivatives** in Algorithm 11.5.4.2 then computes the values we want. The algorithm used comes from [LeeE82]. The proof of its correctness uses a knot insertion–type argument. When d = 0, that is, when we only want the value of the function, then the algorithm reduces to the standard de Boor algorithm described in Section 11.5.2. The Boolean variable fromRight in the algorithm determines which end of the span is closest to u. It is used to select the code that will produce the most accurate result numerically. If one only needs the value and first derivative of a curve, then Luken and Cheng ([LukC96]) describe a two stage Cox–de Boor method that is somewhat better than Lee's algorithm.

Next, we consider the evaluation problem for NURBS curves p(u). Fortunately, by switching to homogeneous coordinates we do not need anything new because the hard part, the evaluation of the ordinary B-splines $N_{i,k}(u)$, has already been done. Algorithm 11.5.4.3 computes p(u). Finding derivatives is more complicated because we have quotients. We describe the algorithm in [PieT95]. Let A(u) and w(u) be the numerator and denominator functions of p(u), respectively, that is,

$$p(u) = \frac{\sum_{i=0}^{n} w_i N_{i,k}(u)\mathbf{p}_i}{\sum_{i=0}^{n} w_i N_{i,k}(u)} = \frac{A(u)}{w(u)}.$$

The quotient rule for derivatives implies that

$$p'(u) = \frac{A'(u) - w'(u)p(u)}{w(u)},$$

so that $p'(u)$ can be computed applying Algorithm 11.5.4.2 to the functions $p(u)$, $w(u)$, and $A(u)$. For higher derivatives, note that

$$A^{(d)}(u) = (w(u)p(u))^{(d)}$$

Applying Leibnitz's formula to the product on the right-hand side of this equation and reorganizing the result leads to

$$p^{(d)}(u) = \frac{A^{(d)}(u) - \sum_{i=1}^{d} \binom{d}{i} w^{(i)}(u) p^{(d-i)}(u)}{w(u)}, \tag{11.107}$$

The Problem: Given an arbitrary B-spline curve

$$p(u) = \sum_{i=0}^{n} N_{i,k}(u)\,\mathbf{p}_i$$

of order k and degree $m = k - 1$, to compute **all** the derivatives of $p(u)$ up to order d at u.

procedure Derivatives (**real array** knots [0..]; **point array** ctrls [0..];
integer m; **real** u; **integer** d; **ref point array** derivs [0..])
{ **Inputs:**
u_i = knots[i] – the knots
$\mathbf{p}_i$ = ctrls[i] – the control points
m – the degree of the B-spline basis functions
u – parameter value at which the function's derivatives are to be evaluated
d – highest derivative desired
$u_m < u_{m+1}$, $u \in [u_m, u_{m+1})$, $d \le m$
See text as to how knot and control point arrays need to be re-indexed before calling this procedure and for the one special case where $u \in (u_m, u_{m+1}]$.
Outputs:
derivs[0..d] – the 0th through dth derivatives of the curve at u
}

Algorithm 11.5.4.2. A B-spline evaluation algorithm ([LeeE82]).

```
begin
    real array  a [0..] , b [0..];
    boolean     fromRight;

    if  m = 0  then
        begin
            derivs[0] := ctrls[0];
            return;
        end;

    for  i:=1 to m  do
        begin
            a[i] := u − knots[i];
            b[i] := knots[m+i] − u;
        end;

    fromRight := (b[1] > a[m]);
    if  fromRight
        then
            begin
                for  i:=0 to m  do  derivs[i] := ctrls[i];
                for  j:=1 to  m  do
                    for  i:=0 to m−j  do
                        derivs[i] := (1.0/(a[i+j]+b[i+1]))*
                                        (a[i+j]*derivs[i+1] + b[i+1]*derivs[i]);
                for  j:=1 to d  do
                    for  i:=d downto j  do
                        derivs[i] := ((m−j+1)/b[i−j+1])*(derivs[i] − derivs[i−1]);
            end
        else
            begin
                for  i:=0 to m  do  derivs[i] := ctrls[m−i];
                for  j:=1 to  m  do
                    for  i:=0 to m−j  do
                        derivs[i] := (1.0/(a[m−i]+b[m−j+1−i]))*
                                        (b[m−j+1−i]*derivs[i+1] + a[m−i]*derivs[i]);
                for  j:=1 to d  do
                    for  i:=d downto j  do
                        derivs[i] := (−(m−j+1)/a[m−i+j])* (derivs[i] − derivs[i−1]);
            end
end;
```

Algorithm 11.5.4.2. *Continued*

The Problem: Given an arbitrary NURBS curve

$$p(u) = \frac{\sum_{i=0}^{n} w_i N_{i,k}(u)\mathbf{p}_i}{\sum_{i=0}^{n} w_i N_{i,k}(u)}$$

of order k , knot vector $U = \{u_0, u_1, \ldots, u_{n+k}\}$, weights w_i, and control points $\mathbf{p}_i = (x_i, y_i, z_i)$, to compute p(u).

Step 1: Use Algorithm 11.5.4.1 to find the span that contains u.
Step 2: Let $\mathbf{P}_i = (x_i w_i, y_i w_i, z_i w_i, w_i)$. Use Algorithm 11.5.4.2 to evaluate

$$P(u) = \sum_{i=0}^{n} N_{i,k}(u)\,\mathbf{P}_i = (P_1(u), P_2(u), P_3(u), P_4(u)) .$$

(The only change to Algorithm 11.5.4.2 is that we allow control points to be 4-tuples rather than 3-tuples.)

Step 3: $p(u) = \left(\frac{P_1(u)}{P_4(u)}, \frac{P_2(u)}{P_4(u)}, \frac{P_3(u)}{P_4(u)}\right)$

Algorithm 11.5.4.3. A NURBS curve evaluation algorithm.

which provides us with a recursive way to compute $p^{(d)}(u)$. Algorithm 11.5.4.4 uses equation (11.107) to compute the derivative of NURBS curves. Note that to speed up the algorithm we have assumed that the binomial coefficients have been precomputed and stored in an array binom[..,..].

11.5.5 B-Spline Interpolation

This section returns to the interpolation problem and describes another approach using B-splines.

The cubic B-spline interpolation problem: Given parameter values u_i , $i = 0, \ldots, n$, with $u_0 < u_1 < \ldots < u_n$, and points $\mathbf{p}_i$, find a cubic B-spline curve p(u) with the u_i as knots and control points $\mathbf{q}_j$, $j = -1, \ldots, n + 1$, so that $p(u_i) = \mathbf{p}_i$.

Following [Fari97] we shall solve the B-spline interpolation problem by using the fact that this spline can be represented as a piecewise cubic Bézier curve with control points $\mathbf{b}_j$, so that $\mathbf{p}_i = \mathbf{b}_{3i}$. See Theorem 11.5.2.12. The idea will be to use the well-defined relationships between the $\mathbf{b}_j$ and the $\mathbf{p}_i$ and also those between the $\mathbf{b}_j$ and the $\mathbf{q}_k$. Eliminating the $\mathbf{b}_j$ from these relationships will give us the relationship between the $\mathbf{p}_i$ and $\mathbf{q}_k$ that we are after.

The Problem: Given an arbitrary NURBS curve

$$p(u) = \frac{\sum_{i=0}^{n} w_i N_{i,k}(u)\, \mathbf{p}_i}{\sum_{i=0}^{n} w_i N_{i,k}(u)} = \frac{A(u)}{w(u)},$$

to compute **all** the derivatives of p(u) up to order d at u.

Given:

Aderivs[0..d] – the 0th through dth derivatives of A(u) at u
wderivs[0..d] – the 0th through dth derivatives of w(u) at u
binom[..,..] – precomputed table of binomial coefficients, binom[i,j] = $\binom{i}{j}$

Outputs:

derivs[0..d] – the 0th through dth derivatives of the curve p(u) at u

```
integer i, j;
real s;

for i:=0 to d do
    begin
        s := Aderivs[i];
        for j:=1 to i do s := s − binom[i,j]*wderivs[j]*derivs[i−j];
        derivs[i] := s/wderivs[0];
    end;
```

Algorithm 11.5.4.4. A NURBS curve derivatives algorithm.

Let $\Delta_i = u_{i+1} - u_i$. It follows from the discussion following Example 11.5.2.11 that

$$\mathbf{p}_i = \frac{\Delta_i \mathbf{b}_{3i-1} + \Delta_{i-1}\mathbf{b}_{3i+1}}{\Delta_{i-1} + \Delta_i}, \tag{11.108}$$

and that

$$\begin{aligned}
\mathbf{b}_2 &= \frac{\Delta_1 \mathbf{q}_0 + \Delta_0 \mathbf{q}_1}{\Delta_0 + \Delta_1}, \\
\mathbf{b}_{3i-1} &= \frac{\Delta_i \mathbf{q}_{i-1} + (\Delta_{i-2} + \Delta_{i-1})\mathbf{q}_i}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i}, \quad i = 2, \ldots, n-1, \\
\mathbf{b}_{3i+1} &= \frac{(\Delta_i + \Delta_{i+1})\mathbf{q}_i + \Delta_{i-1}\mathbf{q}_{i+1}}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}}, \quad i = 1, \ldots, n-2, \quad \text{and} \\
\mathbf{b}_{3n-2} &= \frac{\Delta_{n-1}\mathbf{q}_{n-1} + \Delta_{n-2}\mathbf{q}_{n-2}}{\Delta_{n-2} + \Delta_{n-1}}.
\end{aligned} \tag{11.109}$$

Eliminating the $\mathbf{b}_j$ from equations (11.108) and (11.109) leads to the system of equations

$$(\Delta_{i-1} + \Delta_i)\mathbf{p}_i = \alpha_i \mathbf{q}_{i-1} + \beta_i \mathbf{q}_i + \gamma_i \mathbf{q}_{i+1}, \tag{11.110}$$

where

$$\alpha_i = \frac{(\Delta_i)^2}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i},$$
$$\beta_i = \frac{\Delta_i(\Delta_{i-2} + \Delta_{i-1})}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i} + \frac{\Delta_{i-1}(\Delta_i + \Delta_{i+1})}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}}, \quad \text{and}$$
$$\gamma_i = \frac{(\Delta_{i-1})^2}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}}.$$

After an arbitrary choice of points $\mathbf{b}_1$ and $\mathbf{b}_{3n-1}$, equation (11.110) can be represented in the following matrix form:

$$\begin{pmatrix} 1 & 0 & 0 & & & & \\ \alpha_1 & \beta_1 & \gamma_1 & & & 0 & \\ & & & \ddots & & & \\ & 0 & & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ & & & & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_{n-1} \\ \mathbf{q}_n \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_{n-1} \\ \mathbf{b}_{3n-1} \end{pmatrix}, \tag{11.111}$$

where $\mathbf{r}_i = (\Delta_{i-1} + \Delta_i)\mathbf{p}_i$. Setting $\mathbf{q}_{-1} = \mathbf{p}_0$ and $\mathbf{q}_{n+1} = \mathbf{p}_n$ completes the definition of the $\mathbf{q}_i$.

In the special case where the knots are uniformly spaced, the system (11.111) becomes

$$\begin{pmatrix} 1 & 0 & 0 & & & & & \\ \frac{3}{2} & \frac{7}{2} & 1 & & & & 0 & \\ 0 & 1 & 4 & 1 & & & & \\ & & & \ddots & & & & \\ & & & & 1 & 4 & 1 & 0 \\ & 0 & & & 0 & 1 & \frac{7}{2} & \frac{3}{2} \\ & & & & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \mathbf{q}_2 \\ \vdots \\ \mathbf{q}_{n-2} \\ \mathbf{q}_{n-1} \\ \mathbf{q}_n \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ 6\mathbf{p}_1 \\ 6\mathbf{p}_2 \\ \vdots \\ 6\mathbf{p}_{n-2} \\ 6\mathbf{p}_{n-1} \\ \mathbf{b}_{3n-1} \end{pmatrix}. \tag{11.112}$$

In the other interesting case of a closed curve, we do not need the last point $\mathbf{p}_n$, and matrix equation (11.111) becomes

$$\begin{pmatrix} \beta_0 & \gamma_0 & 0 & & & 0 & \alpha_0 \\ \alpha_1 & \beta_1 & \gamma_1 & & & 0 & 0 \\ & & & \ddots & & & \\ 0 & 0 & & & \alpha_{n-2} & \beta_{n-2} & \gamma_{n-2} \\ \gamma_{n-1} & 0 & & & 0 & \alpha_{n-1} & \beta_{n-1} \end{pmatrix} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_{n-2} \\ \mathbf{q}_{n-1} \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_{n-2} \\ \mathbf{r}_{n-1} \end{pmatrix}. \tag{11.113}$$

Now in practice the knots are not usually given. See [Fari97] for some ways to define them. The uniform spacing case, although simple, does not give the best results usually. The standard choices for spacing are shown below but there are others:

Uniform Spacing. The knot intervals all have the same lengths, that is,

$$\Delta_j = u_{i+1} - u_i = \frac{u_n - u_0}{n}.$$

Chord-length (Chordal) Spacing. The lengths of the knot intervals are proportional to the lengths of the polygon segments, that is,

$$\frac{\Delta_i}{\Delta_{i+1}} = \frac{|\mathbf{p}_{i+1} - \mathbf{p}_i|}{|\mathbf{p}_{i+2} - \mathbf{p}_{i+1}|}.$$

Centripetal Spacing. The lengths of the knot intervals are related to the lengths of the polygon segments via the following relationship

$$\frac{\Delta_i}{\Delta_{i+1}} = \sqrt{\frac{|\mathbf{p}_{i+1} - \mathbf{p}_i|}{|\mathbf{p}_{i+2} - \mathbf{p}_{i+1}|}}.$$

Of the spacing choices listed here only uniform spacing is invariant under affine transformations of the control points. In terms of increasing quality of the shape of the curve, the ranking of the spacing methods would be: uniform, chordal, centripetal. This ranking also applies to the amount of work involved. See [Fari97] or [HosL93] for more on the advantages and disadvantages of the various spacing choices.

The advantage B-spline interpolation has over Hermite and Bézier interpolation is that it needs fewer control points. The Hermite and Bézier interpolations need $3n - 1$ control points whereas the B-spline interpolation needs only n.

What might be considered another variant of the B-spline interpolation problem is the following:

> **The cubic Bézier interpolation problem:** Given points $\mathbf{p}_i$, $i = 0, \ldots, n$, find, for each i, additional Bézier control points $\mathbf{q}_i$ and $\mathbf{r}_i$, so that the Bézier curve defined by the four points $\mathbf{p}_i$, $\mathbf{q}_i$, $\mathbf{r}_i$, and $\mathbf{p}_{i+1}$ matches the uniform cubic spline curve that interpolates these points.

A solution to this problem was already indicated in the discussion leading up to Theorem 11.5.2.12. [Rasa90] describes an interesting answer to this problem which

is actually an approximation to the answer but is accurate enough for computer graphics display purposes. His answer reduces to a simple formula.

The interpolation in this section assumed that the values u_i at which the interpolating spline took on the value $\mathbf{p}_i$ were also its knots. One can be more general and look for an interpolating spline for which not only the parameter values but also the knots are specified (the two sequences could be different). For a solution to that problem and interpolation by higher-order B-splines see [PieT95].

11.6 Nonlinear Splines

When we defined splines we pointed out that the reason for cubic splines being so popular is that they are low-degree polynomials and yet provide very good approximations to many of the curves needed in CAD and CAGD. Specifically, they are a good substitute for the physical splines that were used in the past. To make this argument, however, we need to know what physics tells as about how flexible rods bend. It is not possible to delve into the justification from physics here, but one can show that the so-called mechanical and wooden splines defined below are two models for a curve that describes the shape of a bent rod.

Definition. Let F(s) be an arc-length parameterized curve with domain [0,L] and curvature function $\boldsymbol{\kappa}(s)$ that interpolates a fixed set of points. The curve F(s) is called a *mechanical spline* if it minimizes the energy functional

$$\int_0^L \boldsymbol{\kappa}^2 ds. \tag{11.114}$$

It is called a *wooden spline* if

$$\frac{d^2\boldsymbol{\kappa}}{ds^2} = 0. \tag{11.115}$$

The mechanical and wooden splines are called *nonlinear splines* whereas the polynomial splines defined in Section 11.2.3 would be called "linear" splines. The reason for this is that a linear combination of two polynomial splines with the same knots and degree would again be a spline of that type. In fact, such splines form a finite dimensional linear vector space. On the other hand, a linear combination of mechanical or wooden splines would not have the right curvature and hence not be a spline of that same type. The rest of this section will point out a few facts about nonlinear splines that make them interesting in graphics. Good references are [Mehl74], [Malc77], or [HosL93]. For a more mathematical introduction to nonlinear splines see [Wern79]. We shall stick to the case of planar curves and use signed curvature in the discussion below.

First, consider a mechanical spline. If it is the graph of a function y = f(x) over some interval [a,b], then the integral in (11.114) that defines it turns into

$$\int_a^b \frac{f''^2(x)}{\left(1+f'^2(x)\right)^{5/2}}dx. \tag{11.116}$$

Here we have used the facts that

$$ds = \sqrt{1+f'^2(x)}dx \quad \text{and} \quad \boldsymbol{\kappa}_s(x) = \frac{f''(x)}{\left(1+f'^2(x)\right)^{3/2}},$$

where $\boldsymbol{\kappa}_s(x)$ is the signed curvature function. (They follow from equation (9.2) and Proposition 9.3.4 in [AgoM05]). Assuming that $f'(x)$ is very small, one can drop the denominator in (11.116) and conclude that

$$\int_a^b f''^2(x)dx \tag{11.117}$$

is a good approximation to the integral. An easy application of the calculus of variations shows that the function $f(x)$ that minimizes the integral in (11.117) must satisfy $f^{(4)}(x) = 0$, so that it is a cubic polynomial, that is, a cubic spline. On the other hand, without the simplifying hypothesis that $f'(x)$ is small, the integral in (11.116) is harder to solve. The interested reader should consult [Mehl74], [Malc77], or [HosL93].

Next, consider a wooden spline $F(s)$ defined by equation (11.115). These curves can also be approximated by cubic splines because for graphs of functions $y = f(x)$, the signed curvature $\boldsymbol{\kappa}_s(x)$ can be approximated by $f''(x)$ and so, like for mechanical splines, they are approximated by functions satisfying $f^{(4)}(x) = 0$. Integrating equation (11.115) shows that

$$\boldsymbol{\kappa}_s(s) = as + b, \tag{11.118}$$

for some constants a and b. From this we could already guess at the shape of such a curve. Its curvature increases with s and hence would have to spiral in on itself like the spring of a clock. To get an actual formula, let $\theta(s)$ be the turning angle function for $F(s)$. We know (see Chapter 9 in [AgoM04]) that

$$\boldsymbol{\kappa}_s(s) = \frac{d\theta}{ds} \tag{11.119}$$

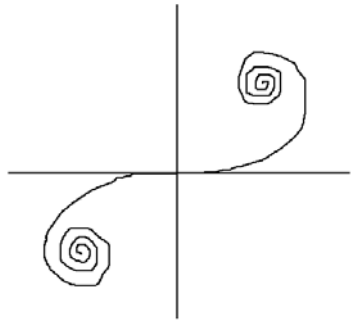
and

$$F(s) = \left(\int_0^s \cos\theta(s)ds + c, \int_0^s \sin\theta(s)ds + d\right), \tag{11.120}$$

where c and d are constants. Integrating equation (11.119) and using (11.118) implies that

$$\theta(s) = \int_0^s \boldsymbol{\kappa}_s(s)ds + \theta_0 = \int_0^s (as+b)ds + \theta_0 = \frac{1}{2}as^2 + bs + \theta_0, \tag{11.121}$$

Figure 11.31. A clothoid or Cornu spiral.



for some constant θ_0. Substituting this into equation (11.120) gives us our general wooden spline F(s). There are well-known special cases.

Definition. The planar curve

$$F(s) = \left(\int_0^s \cos\left(\frac{x^2}{2}\right) dx, \int_0^s \sin\left(\frac{x^2}{2}\right) dx \right)$$

is called a *clothoid* or *Cornu spiral*. The curve

$$F(s) = \left(\int_0^s \cos\left(\frac{x^{n+1}}{n+1}\right) dx, \int_0^s \sin\left(\frac{x^{n+1}}{n+1}\right) dx \right)$$

is called a *generalized clothoid* or *Cornu spiral*.

Figure 11.31 shows the clothoid. The integrals in the definition are Fresnel integrals and since

$$\int_0^{\pm\infty} \cos\left(\frac{x^2}{2}\right) dx = \int_0^{\pm\infty} \sin\left(\frac{x^2}{2}\right) dx = \pm\frac{\sqrt{\pi}}{2}$$

we see the the clothoid converges to the two points

$$\pm\left(\frac{\sqrt{\pi}}{2}, \frac{\sqrt{\pi}}{2}\right).$$

It is easy to show that the generalized clothoid has signed curvature function

$$\kappa_s(s) = s^n.$$

Clothoids play an important role in the construction of freeways and railroad tracks. As an example for why this might be so, note that the curve corresponding to an exit ramp for a highway needs to start off with zero curvature and then reach some

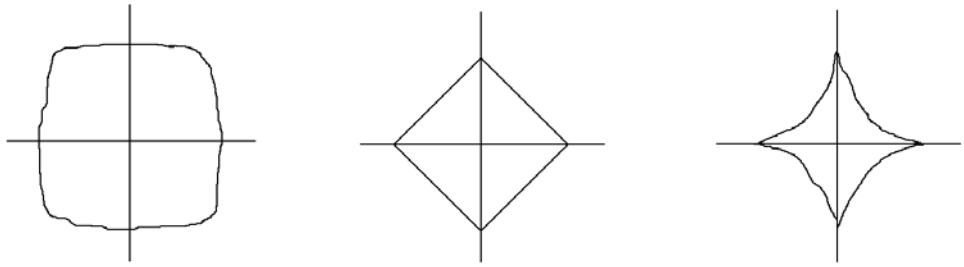


Figure 11.32. Superellipses: $0 < n < 1$, $n = 2$, and $n > 1$.

nonzero curvature in a continuous way. That is exactly what clothoids do. If one were to use a circular arc, then one would have a curvature discontinuity since we would jump from zero to nonzero curvature.

[HosL93] defines interpolating mechanical and wooden splines. In the case of wooden interpolating splines, each segment is required to be a clothoid.

11.7 Superellipses

This section looks briefly at a very special class of curves that generalize ellipses.

Definition. The parametric curve $p(\theta)$ defined by

$$p(\theta) = (a \cos^n \theta, b \sin^n \theta),$$

$0 \le \theta \le 2\pi$, is called a *superellipse*.

Figure 11.32 shows some superellipses. When $0 < n < 1$, the curve looks like a rounded square. The shape approaches that of a square as $n \to 0$. For $n = 1$, we get an ordinary ellipse. When $n = 2$, the curve looks like a diamond and when $n > 1$, it looks like a pinched diamond. As $n \to \infty$, the shape approaches a plus sign. One can show that the curve always lies in the rectangle $[-a,a] \times [-b,b]$. The points $(\pm\sigma a, \pm\sigma b)$, where $\sigma = 2^{-1/n}$, are called the *corner points* of the superellipse and σ is called its *superness*. The implicit form for a superellipse is

$$\left|\frac{x}{a}\right|^m + \left|\frac{y}{n}\right|^m = 1, \quad m \ge 0. \tag{11.122}$$

Superellipses were first defined and studied by the French mathematician Gabriel Lamé in 1818. Their occasional use in modeling systems stems from the fact that the parameter n allows one to control the fullness of the curve.

11.8 Subdivision of Curves

Bézier and B-spline curves are easy to manipulate by moving control points. On the other hand, we may have too few control points to get the exact shape we want. What is needed is to be able to add control points, or also knots in the case of B-splines, to get the desired freedom in shape manipulation. Besides, the fact is that a given curve can be represented in an infinite number of ways in terms of control points and/or knots. It is up to us to choose which we like the best. For efficiency one usually wants the representation that uses the smallest such number, but as we can see, that is not always the important criterion.

To solve our shape manipulation problem, the general idea will be to subdivide the shape and express each piece by means of its own independent control data with the result that the whole curve is then defined by a larger set of control data. We begin by stating a general form of this problem and then show how it relates to cubic curves. Later we shall see what this means for Bézier and B-spline curves.

The general subdivision problem: Suppose that we are given a curve p(u) with domain [a,b], which is expressed in the form

$$p(u) = f_1(u)\mathbf{p}_1 + \ldots + f_s(u)\mathbf{p}_s + g_1(u)\mathbf{v}_1 + \ldots + g_t(u)\mathbf{v}_t, \tag{11.123}$$

where the $\mathbf{p}_i$ are "points" and the $\mathbf{v}_i$ are "vectors" in $\mathbf{R}^m$ and the $f_i(u)$ and $g_i(u)$ are real-valued functions. Let c and d be real numbers with $a \le c \le d \le b$. The problem is to find a curve q(u) so that

(1) q(u) has the **same** domain [a,b] as p(u),
(2) q(u) traces out the same set as p|[c,d], that is, q([a,b]) = p([c,d]), and
(3) the formula for q(u) has the same form as p(u), that is,

$$q(u) = f_1(u)\mathbf{q}_1 + \ldots + f_s(u)\mathbf{q}_s + g_1(u)\mathbf{w}_1 + \ldots + g_t(u)\mathbf{w}_t.$$

for some points $\mathbf{q}_i$ and vectors $\mathbf{w}_i$ (the basis functions $f_i(u)$ and $g_i(u)$ are kept unchanged).

Let us see how the problem can be handled in the case of an **arbitrary** cubic curve p(u) (higher degree curves can be handled similarly). Suppose p(u) has domain [0,1] and is defined by an equation

$$p(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix}\mathbf{M}\begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix}, \tag{11.124}$$

where **M** is some matrix and the $\mathbf{p}_i$ are **either** points **or** vectors. We shall work out the subdivision problem in the case where the interval is to be divided at the value c. The function $u \to cu$ reparameterizes [0,1] as [0,c]. Substituting cu for u in (11.124) gives

$$q(u) = \begin{pmatrix} c^3u^3 & c^2u^2 & cu & 1 \end{pmatrix} \mathbf{M} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix},$$

which can be written in the form

$$q(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \mathbf{K}_c \mathbf{M} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix}, \tag{11.125}$$

where

$$\mathbf{K}_c = \begin{pmatrix} c^3 & 0 & 0 & 0 \\ 0 & c^2 & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Therefore, if we define the points $\mathbf{q}_i$ by the equation

$$\begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \mathbf{q}_2 \\ \mathbf{q}_3 \end{pmatrix} = \mathbf{M}^{-1} \mathbf{K}_c \mathbf{M} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix},$$

then

$$q(u) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \mathbf{M} \begin{pmatrix} \mathbf{q}_0 \\ \mathbf{q}_1 \\ \mathbf{q}_2 \\ \mathbf{q}_3 \end{pmatrix}. \tag{11.126}$$

This solves the subdivision problem over [0,c].

Next, note that to reparametrizes [0,1] to [c,1] replace u by c+u(1–c) in (11.124) to get

$$q(u) = \begin{pmatrix} (c+u(1-c))^3 & (c+u(1-c))^2 & c+u(1-c) & 1 \end{pmatrix} \mathbf{M} \begin{pmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{pmatrix},$$

which can be written in the form

$$q(u) = (u^3\ u^2\ u\ 1)\mathbf{L}_c\mathbf{M}\begin{pmatrix}\mathbf{p}_0\\ \mathbf{p}_1\\ \mathbf{p}_2\\ \mathbf{p}_3\end{pmatrix}, \tag{11.127}$$

where

$$\mathbf{L}_c = \begin{pmatrix}(1-c)^3 & 3c(1-c)^2 & 3c^2(1-c) & c^3\\ 0 & (1-c)^2 & 2c(1-c) & c^2\\ 0 & 0 & 1-c & c\\ 0 & 0 & 0 & 1\end{pmatrix}.$$

Define points $\mathbf{q}_i$ by the equation

$$\begin{pmatrix}\mathbf{q}_0\\ \mathbf{q}_1\\ \mathbf{q}_2\\ \mathbf{q}_3\end{pmatrix} = \mathbf{M}^{-1}\mathbf{L}_c\mathbf{M}\begin{pmatrix}\mathbf{p}_0\\ \mathbf{p}_1\\ \mathbf{p}_2\\ \mathbf{p}_3\end{pmatrix},$$

Then equation (11.126) will be satisfied in this case also and we are done.

What we have just accomplished is to represent the initial cubic curve p(u) in terms of two curves, so that we now have twice as much control data to use in manipulating the curve.

The cubic curve solution applies to cubic Bézier curves. In that case the matrix **M** is just the Bézier matrix. Figure 11.33 shows how p|[0,c] is now represented as a Bézier curve with control points $\mathbf{q}_0$, $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$. Another four control points $\mathbf{q}_3$, $\mathbf{q}_4$, $\mathbf{q}_5$, and $\mathbf{q}_6$ would represent p|[c,1]. This means that we can now use seven control points to modify the curve, where before we had only four.

A more interesting problem is the general Bézier curve p(u) with control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$. It might seem as if the obvious solution here is simply to let a user pick more control points interactively. The problem with that is that the user is then

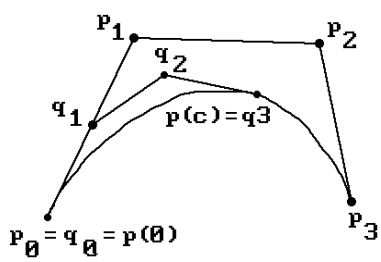


Figure 11.33. Subdividing cubic Bézier curves.

basically starting the curve design from scratch again. The fact is that the user may already have a reasonable shape and may only want to fine tune it. What is needed therefore is that one wants to first add control points near the point of interest in such a way that the larger collection still describes the **original** curve and then let the user continue with more detailed modifications. Fortunately, this is possible. One can increase the number of control points for a Bézier curve and still end up with the same curve. We shall not describe this method, called "degree elevation," here. See [Fari97]. Instead, we describe a curve subdivision method.

Given that [0,1] is the domain for the curve p(u), we shall again divide the domain [0,1] at a value c, $0 \le c \le 1$. The set p([0,c]) is actually the path of a Bézier curve q(v) defined on [0,1] with control points $\mathbf{q}_0 = \mathbf{p}_0, \mathbf{q}_1, \ldots, \mathbf{q}_n = p(c)$, so that

$$p(u) = q\left(\frac{u}{c}\right), \quad u \in [0,c].$$

The problem is to find the control points $\mathbf{q}_i$. See Figure 11.33. Using the notation of the de Casteljau algorithm in Section 11.5.2:

$$\mathbf{q}_i = \mathbf{p}_0^i(c) \tag{11.128}$$

This shows that the de Casteljau algorithm can be used to find the $\mathbf{q}_i$. Furthermore, because of the symmetry property of Bézier curves, the control points $\mathbf{r}_0 = p(c)$, $\mathbf{r}_1, \ldots, \mathbf{r}_n = \mathbf{p}_n$ of the Bézier curve

$$r(v) = p(c + v(1-c)), \quad v \in [0,1],$$

for the set p([c,1]) are

$$\mathbf{r}_i = \mathbf{p}_{n-i}^i(c). \tag{11.129}$$

The points $\mathbf{q}_i$ and $\mathbf{r}_j$ are the extra control points we were looking for.

The subdivision problem for B-spline curves has a solution similar to that of the Bézier problem. We already gave two algorithms in Section 11.5.2 for adding either a single knot (Theorem 11.5.2.13) or multiple knots (Theorem 11.5.2.14).

11.9 Composition of Curves and Geometric Continuity

From an abstract point of view, when one talks about curves one usually has one-dimensional subsets of $\mathbf{R}^m$ in mind and the parameterizations of these subsets are incidental. One is interested in properties of these **sets**. The functions that parameterize them are usually just intermediary concepts. In practice however, parameterizations do play a role. A given curve may be defined by means of several parametric curves, each of which traces out only part of the whole curve. Questions arise as to how the parametric curves meet. This section takes a brief look at some answers to such questions.

Suppose that we have two C^k curves

$$p, q : [0,1] \to \mathbf{R}^m \quad \text{with } p(1) = q(0). \tag{11.130}$$

How "smooth" is

$$\mathbf{X} = p([0,1]) \cup q([0,1])$$

at b? How can we control the smoothness, or lack of it, there? These are the type of questions considered in this section. We might be tempted to say that the composition of these two curves will certainly be continuous because of the hypothesis that $p(1) = q(0)$, but this does not make much sense unless we know what is meant by the composite **map**.

Definition. The parametric curve γ: $[0,1] \to \mathbf{R}^m$ defined by

$$\begin{aligned} \gamma(u) &= p(2u) && \text{for} \quad u \in [0,1/2] \\ &= q(2u-1) && \text{for} \quad u \in [1/2,1] \end{aligned}$$

is called the *composite* of the curves p(u) and q(u) in expression (11.130).

Note that by the chain rule, the speed of the curve $\gamma(u)$ is twice that of the curves p(u) and q(u). This change was forced on us because we wanted the domain of the composite to be [0,1]. On the other hand, the tangent lines of $\gamma(u)$ agree with those of p(u) and q(u). This is what is important to us and not the fact that the speed changed by a common multiple. If we wanted the velocity of the new curve to be the same as the velocities of the old ones then we could have defined a curve with domain [0,2], which agrees with p(u) on [0,1] and with q(u − 1) on [1,2].

Now, when two regular parametric curves meet at a point where they have a common tangent line, they can be reparameterized (for example, using the arc-length parameterization) so that their tangent vectors match where they meet to make the composite differentiable. Mathematically, therefore, it is unimportant whether or not the tangent vectors match exactly because we always get a differentiable manifold. However, there are practical reasons for allowing parametric curves to meet with a common tangent **line** but **distinct** tangent vectors.

Returning to the curves in (11.130), suppose that p([0,1]) has the same tangent line at p(1) as q([0,1]) at q(0), that is, $p'(1) = a\, q'(0)$, where $a \neq 0$. The curve will look smooth but it will not be differentiable unless a = 1. On the other hand, if we had chosen different parameterizations, then this composite might be differentiable. Because it is convenient to allow for "wrong" parameterizations, the general question is whether one can tell from p and q alone whether **X** will be smooth.

Assume that p(u) and q(u) are regular curves.

Definition. We say that the regular curves p(u) and q(u) in (11.130) meet with *kth order geometric continuity*, or G^k *continuity*, if there is an equivalent parameterization r(u) for p(u) with respect to an orientation-preserving change of parameters so that r(u) and q(u) meet with C^k continuity, that is, the composite curve is also C^k at $p(1) = q(0)$. A regular curve is called a G^k *continuous* or simply G^k *curve* if it admits

a reparameterization that is C^k with respect to an orientation-preserving change of parameters.

Note. G^k continuity really corresponds to the composite curve being a C^k manifold in a neighborhood of the point where they join. Unfortunately, this observation came later in the historical development of the concept of G^k continuity. Recognition of this fact from the beginning would have clarified the issue. G^k continuity has to do with continuity of the shape of a curve whereas C^k continuity has to do with the continuity of the parameterization p(u).

Assume that $r(u) = p(\varphi(u))$. The chain rule implies that

$$r'(u) = q'(\varphi(u))\varphi'(u) \quad \text{and} \quad r''(u) = p''(\varphi(u))\varphi'(u)^2 + p'(\varphi(u))\varphi''(u).$$

Letting

$$\beta_1 = \varphi'(1) \quad \text{and} \quad \beta_2 = \varphi''(1), \tag{11.131}$$

it follows that

$$q'(0) = r'(1) = \beta_1 p'(1), \tag{11.132a}$$

$$q''(0) = r''(1) = \beta_1^2 p''(1) + \beta_2 p'(1), \tag{11.132b}$$

with $\beta_1 > 0$.

Definition. The βs are called *shape parameters* and equations (11.132) are called the *beta constraints*. The numbers β_1 and β_2 are called the *bias* and *tension* of the curve, respectively.

Figure 11.34 expresses the geometry of the situation.

11.9.1 Theorem

(1) Two parametric curves meet with G^1 continuity if and only if they have the same unit tangent vector at their common point.

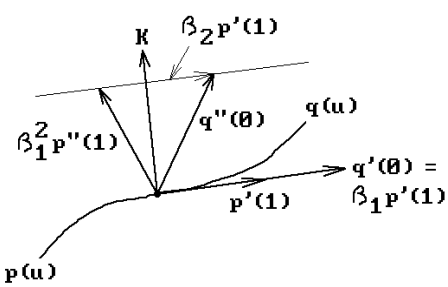


Figure 11.34. The shape parameters and geometric continuity.

(2) Two parametric curves meet with G^2 continuity if and only if they have the same unit tangent and curvature vector at their common point, that is, the arc-length parameterization is C^2.

(3) In general, two parametric curves meet with G^k continuity if and only if the arc-length parameterization of the composite is C^k at their common point.

Proof. See [BarD89].

Because of Theorem 11.9.1(3), G^k continuity is sometimes referred to as *C^k arc-length continuity*. Also, because of the relationship between curvature and the second derivative, G^2 continuous curves are called *curvature continuous*.

Turning things around, one can show that the composite of the two curves in equation (11.130) meet with second-order geometric continuity if there exist two constants $\beta_1 > 0$ and β_2, so that equations (11.132) hold. Therefore, one can define a collection of basis functions, called *Beta-splines*, satisfying (11.132) and parameterized by β_1 and β_2. These functions have all the basic properties as the regular B-spline basis functions. Using them one now has additional control over the curvature and shape of a curve because one can now alter the β_1 and β_2 values. Beta splines are the geometrically continuous analog of ordinary B-splines.

There are other formulations of the geometric continuity problem. See [Fari97]. Each has its own advantages. As usual, it depends on the problem that one is trying to solve as far as deciding on an approach.

A more geometric formulation of geometric continuity in the case of two cubic Bézier curves is the following. Let $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$, $\mathbf{b}_3$ and $\mathbf{c}_0$, $\mathbf{c}_1$, $\mathbf{c}_2$, $\mathbf{c}_3$ be the control points of the two curves with $\mathbf{b}_3 = \mathbf{c}_0$. See Figure 11.35. Let B_-, B_+, C_-, and C_+ be the areas of the triangles $\mathbf{b}_1\mathbf{b}_2\mathbf{b}_3$, $\mathbf{b}_2\mathbf{b}_3\mathbf{d}$, $\mathbf{d}\mathbf{b}_3\mathbf{c}_1$, and $\mathbf{b}_3\mathbf{c}_1\mathbf{c}_2$, respectively. Define $r_- = |\mathbf{b}_1\mathbf{b}_2|/|\mathbf{b}_2\mathbf{d}|$, $r_+ = |\mathbf{d}\mathbf{c}_1|/|\mathbf{c}_1,\mathbf{c}_2|$, and $r = |\mathbf{b}_2\mathbf{b}_3|/|\mathbf{b}_3\mathbf{c}_1|$. Then one can prove

11.9.2 Theorem. We have G^2 continuity at $\mathbf{b}_3$ if $r^2 = r_-r_+$.

Proof. See [Fari97].

The geometric constraint defined in Theorem 11.9.2 can be used to construct a cubic G^2 spline from a set of given control points whose curvature a user can control by specifying suitable tangents. Two interior Bézier points are added for each successive pair of control points subject to the G^2 continuity constraints and the final curve is a collection of Bézier curves. See [Fari97].

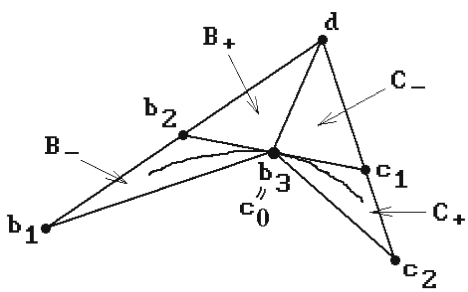


Figure 11.35. Geometric continuity for Bézier curves.

For more information about geometric continuity see [Fari97] or [Greg89]. It is of interest to CAGD not only for curves but also for composite surface patches. See Section 12.15. Another type of continuity has also been defined for curves that is more geometric because it relates directly to the concepts of curvature and torsion of the curve rather than just differentiability.

Definition. We say that the $\mathbf{R}^m$ curves p(u) and q(u) in (11.130) meet with *Frenet frame continuity* if the first m derivatives of the curves are linearly independent and the Frenet basis and generalized curvatures of the composite curve (see Theorem 9.4.11 in [AgoM05]) are continuous at the point p(1) = q(0).

Frenet frame continuity and G^k continuity are similar but not the same. First of all, note that k does not have to equal m. One can show that mth order geometric continuity implies Frenet frame continuity but **not** vice versa. See [Boeh87] and [Greg89].

11.10 The Shape of a Curve

This section looks at some properties that influence the shape of a curve. Specifically, at issue is a geometric determination of whether a curve has loops, cusps, or inflection points.

Definition. Let p: [a,b] $\to \mathbf{R}^n$ be a parametric curve. (We allow $a = -\infty$ or $b = \infty$.) The curve p(u) is said to have a *loop* if the parameterization is not one-to-one on (a,b), that is, it intersects itself. The curve is said to have a *cusp* at $c \in (a,b)$ (or at p(c)) if

(1) its tangent vector vanishes at c, that is, $p'(c) = \mathbf{0}$,
(2) c is an isolated zero of $p'(u)$, and
(3) the unit tangent vectors have an essential discontinuity at c.

The curve is said to have an *inflection point* at $c \in (a,b)$ if the curvature of the curve is 0 at c and c is an isolated 0 of the curvature function. Loops, cusps, and inflection points are called *singularities* of the curve.

The reason that loops, cusps, and inflection points are interesting is that the presence of such points can cause problems in manufacturing processes or in the manufactured objects themselves. For example, surfaces whose cross-sectional curves have loops cannot be manufactured. Milling machines can have problems if they follow curves that have a cusp. Inflection points can cause aerodynamic instabilities.

Since curves are often described by means of control points, it is natural to ask how the position of these control points determines the existence of singularities. This question is discussed at great length in [SuLi89], [SuLi83], [StoD89], and [Wang81]. We shall summarize some of the results that pertain to planar cubic curves, although non-planar and non-cubic curves have been treated as well.

Let

$$p(u) = \mathbf{a}_0 + \mathbf{a}_1 u + \mathbf{a}_2 u^2 + \mathbf{a}_3 u^3 = (x(u), y(u)) \tag{11.133}$$

be a planar cubic curve. It turns out that the singular points u of the curve p(u) are all found among the roots of the equation

$$S(u) = \det\begin{pmatrix} p'(u) \\ p''(u) \end{pmatrix} = x'(u)y''(u) - x''(u)y'(u) = 0. \tag{11.134}$$

This is clear for the cusps. It is also true for the inflection points because of the relationship of the function S(u) with the curvature function of the curve (Proposition 9.3.4 in [AgoM05]). That it also gives information about loops may be a little surprising however. Doing the computation, it is easy to show that

$$S(u) = A\,u^2 + B\,u + C, \tag{11.135}$$

where

$$A = 6\det\begin{pmatrix} \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix}, \quad B = 6\det\begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_3 \end{pmatrix}, \quad \text{and} \quad C = 2\det\begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{pmatrix}.$$

The cubic term, which might have been expected in (11.134), has canceled out.

Note. In this section, a planar cubic curve will be called *nondegenerate* if every line in the plane meets the curve in three points over the **complex** numbers, counting multiplicities. In essence, we are excluding curves that lie in a line or conic.

11.10.1 Theorem. Using the notation in (11.135) for a nondegenerate cubic curve p(u) defined by equation (11.133), let $\Delta = B^2 - 4AC$.

(1) If $A = 0$, then p(u) has exactly one inflection point.
(2) Assume that $A \neq 0$.

  (a) If $\Delta > 0$, then p(u) has exactly two inflection points.
  (b) If $\Delta = 0$, then p(u) has a cusp.
  (c) If $\Delta < 0$, then p(u) has a loop.

Proof. See [SuLi83] and [Wang81]. The domain of p(u) is assumed to be all of **R** here.

It follows from Theorem 11.10.1 that a nondegenerate cubic curve can have at most one type of singularity (a loop, a cusp, or inflection points). It cannot have two types simultaneously.

The analysis of the shape of a cubic curve now proceeds by using a Bézier representation in which the curve is defined by four control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. If the curve is not degenerate, then we can map it into a canonical position with $\mathbf{p}_0 = 0$, $\mathbf{p}_1 = (0,1)$, and $\mathbf{p}_2 = (1,1)$ by a linear change of variables. Such transformations preserve the singularities. We have fixed three of the control points, but the fourth is

free to move. Figure 11.36 now shows how a particular choice of the fourth point can lead to the different types of cubic curves. The object therefore is to describe the regions in the plane, so that, as $\mathbf{p}_3$ varies over the points of a region, we generate the same type of singularity. Figure 11.37 is the diagram one obtains if the domain of the curve (11.133) is restricted to [0,1], where the labels have the following meaning:

Cusp line: The parabolic curve defined by the equation $\Delta = 0$.
Parabolic point: A point where the cubic curve degenerates into a quadratic curve.

See [StoD89] for a more complete analysis.

Other papers on cusps and inflection points are [ManC92a] and [LiCr97].

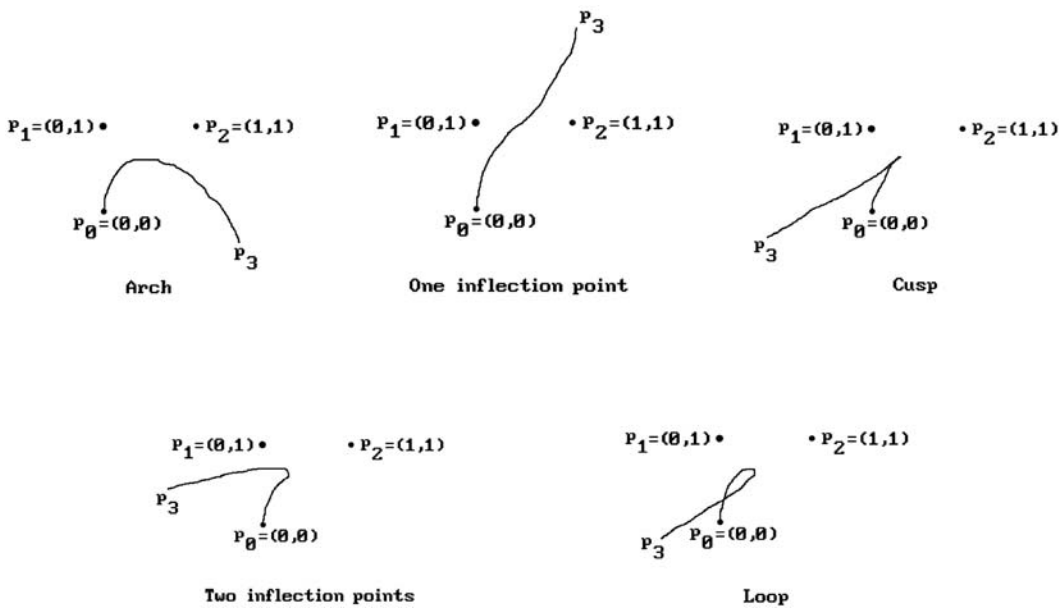


Figure 11.36. Cubic curve singularities.

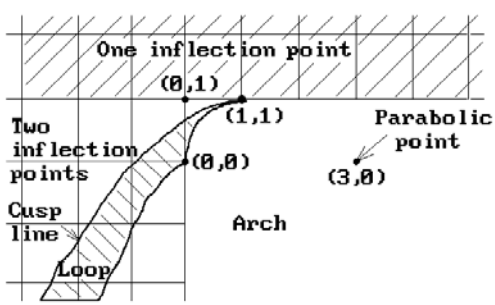


Figure 11.37. The regions of constant singularity type.

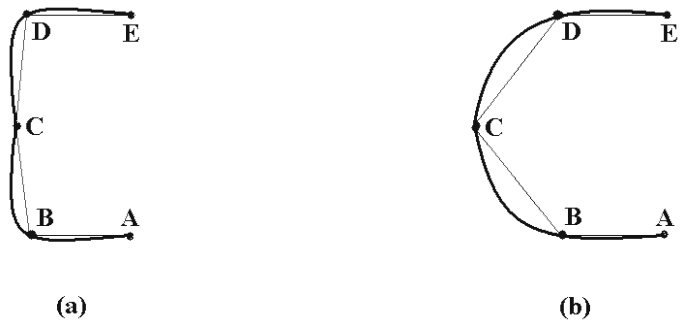


Figure 11.38. Buckling in an interpolating spline.

Another interesting approach to analyzing the shape of cubic curves can be found in the series of papers [Blin89a], [Blin89b], [Blin99], [Blin00a], and [Blin00b]. A related topic has to do with buckling of interpolating spline curves or surfaces. Consider Figure 11.38(a), which shows an interpolating spline through points for which the control polygon is convex but the spline is not. The nonconvex region of the spline near the point **C** is called a *buckle*. A similar phenomenon can happen for interpolating surfaces. Buckling is usually undesirable and the hope is that moving the offending control point/s slightly would remove the flaw. Figure 11.38(b) shows a new position for the point **C** in Figure 11.38(a) that eliminates the buckling. Note that buckling indicates that an inflection point is present. See [VanW96] for a way to detect buckling and remove it.

11.11 Hodographs

Although a well-established term in classical mechanics, the term "hodograph" usually only gets mentioned in passing, if at all, in the geometric modeling literature.

Definition. The *hodograph* of a plane curve p(u) is defined to be the subset of the plane traced out by derivative p′(u).

See Figure 11.39. There are some interesting applications of hodographs. See [Forr72], [Bézi72], [SedW87], [Faro92], and [KimD93]. For example, in [Bézi72] it is shown that the hodograph can be used to characterize geometric properties of Bézier curves, such as inflection points and cusps. Recall from Section 11.4 that the derivative of a Bézier curve is a Bézier curve. In fact, equation (11.57) showed that if the curve had control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ then the control points of the derivative are $n\mathbf{a}_i$, where $\mathbf{a}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$, $i = 0, 1, \ldots, n-1$. [Faro92] shows that offsets to curves whose hodographs satisfy a certain property admit a rational parameterization. This means that one can deal with such offsets directly without approximations. [KimD93] uses hodographs to characterize arbitrary plane cubic curves. The answer has the same flavor as that in [StoD89] in that the plane is divided into different regions which

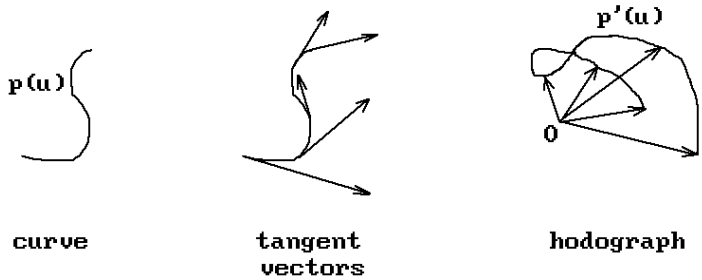


Figure 11.39. Curve and its hodograph.

define the inflection points, loops, etc. It is claimed that this new approach is not as sensitive to degenerate cases as previous solutions. Other aspects of hodographs are described in [SaWS95] and [Moon99].

11.12 Fairing Curves

Producing curves with a pleasing or "fair" shape is not easy. One is confronted with two tasks here. First, one has to define what it means for a curve to be fair and provide tests that one can use to check for this property. The latter step is sometimes called curve interrogation. Second, one has to describe procedures with which a curve that is not fair can be turned into one that is. This process is referred to as "fairing" the curve. A good reference is [HosL93].

There does not seem to be any consensus on a definition of "fair." Curvature clearly has something to do with it. A common way to deal with fairing is to analyze a plot of the curvature. Small differences between curves that may not be visible to the eye can show up in such plots. See [Fari92a]. We shall define the concept as follows (see [SuLi89]):

Definition. A curve p(u) is said to be *fair* if

(1) it is G^2 continuous,
(2) it has no undesirable inflection points, and
(3) its curvature varies in an even way.

As is pointed out in [SuLi89], conditions (2) and (3) are the most important usually, with emphasis on condition (3). Plots of the curvature functions are useful. Basically, one likes curves whose curvature function consists of a few monotone (preferably linear) regions. The points that separate these regions should be a small set. Another way to put this is to say that the curvature has few extrema. With this definition a sine curve is not fair because it has lots of inflection points.

Since curves are often defined by point data, we make the following definition:

Definition. A set of points is said to be *fair* if there exists a fair curve that interpolates these points.

This leads to various solutions to the curve-fitting problem. Two well-known methods are the *least squares method* and the *energy function approach*. The latter is based on the idea that nature will naturally produce fair curves and their shapes are determined by minimizing certain bending energy integrals. A variety of these integrals are described in [HosL93]. Another approach to finding a fair interpolating curve is to start with the obvious polygonal curve that connects the points and then to improve its shape using *interpolatory refinement schemes*. See [Kobb96] for an overview of several of such schemes and how they can be described in a more systematic way.

Now if the control points of a curve are chosen badly, then there is not much one can do to improve its shape. Therefore, part of the task of coming up with fair curves is picking good data to interpolate or approximate. We want to eliminate the hopefully few bad points.

For more about fairing and curvature see Section 15.2. See also [SéCM95] where fairing is obtained by minimizing various functionals. An adaptive approach to fairing digitized point data is described in [LWZL02].

11.13 Parallel Transport Frames

The Frenet frames to a curve provide a moving coordinate system along the points of a curve that often comes in handy. They are, for example, useful in dealing with generalized cylinders. The problem is that Frenet frames are not defined at points of a curve where the second derivative vanishes and the curve is locally flat. This section briefly describes a more general way to define a moving coordinate system that applies to all **regular** curves, whether or not they have a vanishing second derivative. For additional information see [Bish75], [ShaB84], [Bloo90], or [HaMa95].

Given a regular space curve p(t), let T(t) be the unit tangent vector of the curve at p(t), that is,

$$T(t) = \frac{1}{|p'(t)|} p'(t).$$

Our specific problem is to find parameterized unit vectors $n_1(t)$ and $n_2(t)$ so that the triple $F(t) = (T(t), n_1(t), n_2(t))$ defines a continuously varying orthonormal basis for $\mathbf{R}^3$ at p(t). If the second derivative of p(t) does not vanish, then one solution is the Frenet frame (T(t),N(t),B(t)), where N(t) and B(t) are the principal normal and binormal to p(t) at t, respectively. Alternatively, we can pick an arbitrary orthonormal basis $(T(t_0), n_1(t_0), n_2(t_0))$ of vectors at a start parameter t_0 and try to propagate this basis along p(t) by rotating it by a rotation based on the way that T(t) rotates as it moves, independent of the curvature. Such frames, defined more precisely shortly, are called parallel transport frames.

An Algorithm for Computing Parallel Transport Frames $F(t_i)$ at Points $p(t_i)$. Assume that we already have the frame $F(t_{i-1}) = (T(t_{i-1}), n_1(t_{i-1}), n_2(t_{i-1}))$ at $p(t_{i-1})$. Let

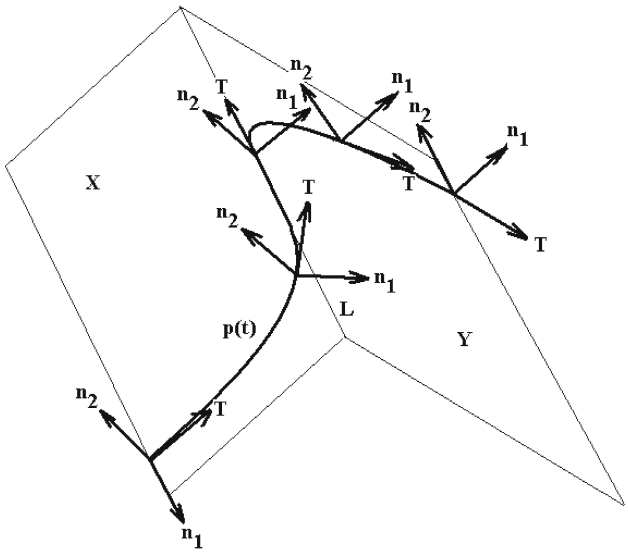


Figure 11.40. Parallel transport frames (T,n_1,n_2).

θ_i be the angle between $T(t_{i-1})$ and $T(t_i)$ and let $\mathbf{w}_i = T(t_{i-1}) \times T(t_i)$ be the unit vector that is orthogonal to the plane generated by $T(t_{i-1})$ and $T(t_i)$. Then $F(t_i) = (T(t_i),n_1(t_i),n_2(t_i))$ is the frame obtained by rotating $(T(t_{i-1}),n_1(t_{i-1}),n_2(t_{i-1}))$ about $\mathbf{w}_i$ through an angle θ_i. See Figure 11.40.

To describe the mathematics behind this algorithm we need to use some facts about frame fields on $\mathbf{R}^3$ and along curves. For the rest of this section we shall assume that

$$p : [a, b] \to \mathbf{R}^3$$

is a regular space curve.

Definition. A *vector field along the curve* p(t) is a vector-valued function $\mathbf{X}$: [a,b] $\to$ $\mathbf{R}^3$. The vector field X is *tangential* or *normal to p(t)* if the vectors $\mathbf{X}(t)$ and $p'(t)$ are parallel or orthogonal, respectively, for all t.

Definition. A normal vector field $\mathbf{X}$ along the curve p(t) is said to be *relatively parallel* to p(t) if $\mathbf{X}'(t)$ is a tangential vector field.

In the case of a relatively parallel normal vector field $\mathbf{X}$ the fact that $\mathbf{X} \bullet \mathbf{X}' = 0$ implies that the vectors $\mathbf{X}(t)$ have constant length. Although the next fact is not needed here, it is an interesting connection to parallel curves that is worth making.

11.13.1 Theorem. A normal vector field $\mathbf{X}$ is relatively parallel to p(t) if and only if p(t) and q(t) = p(t) + X(t) are parallel curves.

Proof. See [Bish75].

The next lemma answers the question as to whether relatively parallel normal vector fields exist.

11.13.2 Lemma. Let $c \in [a,b]$. Given any vector $\mathbf{v}$, there is a unique relatively parallel normal vector field $\mathbf{X}(t)$ to $p(t)$ with the property that $\mathbf{X}(c) = \mathbf{v}$.

Proof. The uniqueness follows from the fact that the difference of relatively parallel vector fields is again such a field and the fact that the difference is 0 at c and hence would have to be constantly equal to 0. The existence part follows by expressing $\mathbf{X}(t)$ in terms of a continuously varying orthonormal basis $(T(t),n_1(t),n_2(t))$ for $\mathbf{R}^3$ along $p(t)$ (which always exists locally) and showing that a solution depends on the existence of a solution to Serret-Frenet type differential equations. See [Bish75].

Definition. A tangential vector field $\mathbf{X}$ along $p(t)$ is said to be *relatively parallel* to $p(t)$ if there is a constant c so that $\mathbf{X}'(t) = cT(t)$ for all t. An arbitrary vector field $\mathbf{X}$ is said to be *relatively parallel* to $p(t)$ if the tangential and normal parts (that is, the vector fields consisting of the orthogonal projections of $\mathbf{X}(t)$ onto the tangent line and normal plane of the curve at $p(t)$, respectively) are relatively parallel.

Lemma 11.13.2 generalizes to the following:

11.13.3 Theorem. Let $p: [a,b] \to \mathbf{R}^3$ be a C^k regular space curve, $k \geq 2$. Given $c \in [a,b]$ and any vector $\mathbf{v}$ there is a unique C^{k-1} relatively parallel vector field $\mathbf{X}(t)$ along $p(t)$ with $\mathbf{X}(c) = \mathbf{v}$. The dot product of any two relatively parallel vector fields is constant.

Proof. See [Bish75].

Now it is clear that the space of relatively parallel vector fields along $p(t)$ is a vector space over $\mathbf{R}$. Furthermore, it follows from Theorem 11.13.3 that this is a three-dimensional vector space.

Definition. A triple (T,n_1,n_2) of orthonormal relatively parallel vector fields along $p(t)$ is called a *relatively parallel adapted frame field* for $p(t)$. (As usual, T denotes the unit tangent vector field for $p(t)$.) The frames $(T(t),n_1(t),n_2(t))$ are called *parallel transport frames*.

11.13.4 Theorem. Let $c \in [a,b]$. Any frame $(T(c),\mathbf{u}_1,\mathbf{u}_2)$ at $p(c)$ defines a unique relatively parallel adapted frame field (T,n_1,n_2) for $p(t)$ so that $n_1(c) = \mathbf{u}_1$ and $n_2(c) = \mathbf{u}_2$.

Proof. One can show that a relatively parallel adapted frame field (T,n_1,n_2) satisfies (and can be obtained as a solution to) the differential equations

$$\begin{aligned} T'(t) &= v(t)k_1(t)n_1(t) + v(t)k_2(t)n_2(t) \\ n_1'(t) &= -v(t)k_1(t)T(t) \\ n_2'(t) &= -v(t)k_2(t)T(t). \end{aligned} \tag{11.136}$$

for some functions $k_1(t)$ and $k_2(t)$, where $v(t) = |p'(t)|$. This follows from the definition of relatively parallel and a translation of the proof of Theorem 9.16.6 in [AgoM05] into this context. One then simply pieces together the unique local solutions defined by their initial conditions.

11.13.5 Theorem. The following relationships exist between the curvature and torsion of a curve p(t) and the functions $k_1(t)$ and $k_2(t)$ in equations (11.136) for its parallel transport frames:

$$\boldsymbol{\kappa}(t) = \sqrt{k_1(t)^2 + k_2(t)^2}$$
$$\boldsymbol{\tau}(t) = \frac{d\theta}{dt}(t), \quad \text{where} \quad \theta(t) = \arctan\left(\frac{k_2(t)}{k_1(t)}\right).$$

Proof. See [Bish75]. The formula for $\boldsymbol{\kappa}(t)$ is clear from the definition of the curvature function. To get the formula for torsion function $\boldsymbol{\tau}(t)$ assume that we have arc-length parameterization and write the principal normal N in the form

$$N = (\cos\theta)n_1 + (\sin\theta)n_2. \tag{11.137}$$

Assuming that the frame field (T, n_1, n_2) has been oriented correctly, one gets that the binormal B is defined by

$$B = (-\sin\theta)n_1 + (\cos\theta)\, n_2,$$

since that is a vector orthogonal to the N (and T). Differentiating equation (11.137) gives

$$\begin{aligned} N' &= -\boldsymbol{\kappa}T + \theta'\left((-\sin\theta)n_1 + (\cos\theta)n_2\right) \\ &= -\boldsymbol{\kappa}T + \theta' B. \end{aligned} \tag{11.138}$$

Finally, comparing equation (11.138) with the Serret-Frenet formulas shows us that $\boldsymbol{\tau} = \theta'$.

Here are some points to keep in mind when deciding on frames for curves. The advantage of Frenet frames is that they are defined locally. Their disadvantage is that one cannot use them if second derivatives of curves are zero. The advantage of parallel transport frames is that they are defined for arbitrary regular curves. Their disadvantage is that, being basically solutions to differential equations, errors may accumulate as one moves far from the start point. One can of course switch back and forth between Frenet and parallel transport frames as appropriate. If one uses curves and Frenet frames to define tubes, one will observe a great deal of twisting near points on the curve with small curvature or large torsion. This undesirable property is readily explained by the generalized Serret-Frenet formulas (Theorem 9.4.8 in [AgoM05]) since the derivatives of the normal and binormal vectors will be large at those places. Reducing such twisting is one reason for using parallel transport frames. On the other

hand, such as in the case of helical curves, the Frenet frames produce more intuitively desirable results than the parallel transport planes.

Finally, Hanson and Ma ([HaMa95]) show how quaternions can be helpful in generating both Frenet and parallel transport frames.

11.14 Recursive Subdivision Curves

The discussion up to now concerned smooth curves. The splines may have had a control polygon driving their shape, but the actual curve had a smooth parameterization. This section addresses the question of smooth curves from a different direction. We shall describe an algorithm that starts with a polygonal curve and smooths out the curve directly by a "subdivision" process without invoking the machinery of splines. At the end we shall still only have a polygonal curve and not actually any function that parameterizes it. The algorithm in question is the Chaikin curve subdivision algorithm ([Chai74]). The idea behind it is to define recursively a sequence of curves where each new curve is obtained from the previous one by "cutting off the corners." If we go far enough out in this sequence we shall have a smooth-looking curve. More precisely, given a polygonal curve Chaikin introduced a new vertex in each edge that alternately was either three quarters or one quarter of the way from the first vertex to the second vertex of the edge. This new sequence of vertices, along with the first and last vertex of the old curve if it was not closed, then defined the next curve. Repeating this process produces a convergent sequence of polygonal curves. Figure 11.41 shows two stages of this algorithm. The initial polygon is defined by the vertices **A**, **B**, **C**, **D**, and **E**. Performing the Chaikin algorithm twice produces the polygon defined by the vertices **A**, **a**, **b**, . . . , **l**, **E**.

Limit curves obtained by a recursive algorithm like Chaikin's are sometimes called *recursive subdivision curves*. Riesenfeld ([Ries75]) observed that the limit curves of the Chaikin algorithm are in fact quadratic B-splines. A good discussion of recursive curve (and surface) algorithms from a historical perspective can be found in [CavM89]. Chaikin's algorithm is a special case of much more general constructions.

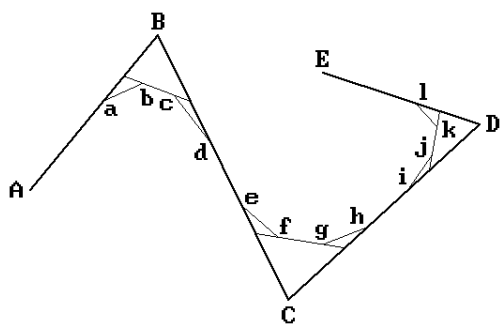


Figure 11.41. Two stages of the Chaikin curve subdivision algorithm.

11.15 Summary

It is worthwhile to summarize the main properties of the various types of parametric curves we discussed in this chapter. First of all, recall that Bézier curves are really special cases of B-splines, so that when one refers to "Bézier curves" and "B-spline curves" one is really referring to their **representation** and not to the underlying curve. They are simply different ways of controlling a curve and based on different basis functions. One can convert between the two representations.

Differences between the Bézier and B-spline curves:

(1) For splines one needs to specify knots t_i. Bézier curves do not need knots.
(2) One can force an order k B-spline to pass through a control point by giving that point a multiplicity $k - 1$, although a cusp may result. Similarly, a knot of order $k - 1$ will cause the B-spline to pass through the corresponding control point.
(3) B-splines use fewer control points than the corresponding Bézier curve.
(4) The Bézier basis functions are easier to compute.

Similarities between Bézier and B-spline curves:

(1) The shape of the curves roughly follow the outlines of the control polygon, although B-spline curves offer more control over the shape.
(2) B-splines can be made to start and end at control points like the Bézier curves.
(3) The tangents at the ends of the curve is determined by the slope of the secants at the ends.
(4) They are symmetric.
(5) They are affinely invariant, but this is true for B-splines only if we use uniformly spaced knots.
(6) They are invariant under affine parameter transformations.
(7) They satisfy the variation diminishing property.

Other properties of Bézier curves:

(1) They satisfy the convex hull property, that is, they lie in the convex hull of their characteristic polygon.
(2) They interpolate the first and last control points.
(3) Their tangent vector at the beginning and end is parallel to the line between the first and last two control points.
(4) They only have pseudo local control: Although changing any control point changes the **entire** curve, the change in the curve drops off rapidly as one moves away from the point that was changed.

Other properties of B-spline curves:

(1) They satisfy a strengthened version of the convex hull property, namely, they lie inside the union of triangles defined from consecutive triples of control points. See Figure 11.20.

(2) They provide local control.
(3) The nonperiodic B-splines interpolate the first and last control points, the periodic ones do not.
(4) The degree of a Bézier curve increases with the number n of points, whereas with B-splines one can vary n and the differentiability via k independently.
(5) Mathematically, a Bézier parameterization is a special case of a B-spline parameterizations.
(6) The higher the multiplicity of a knot, the lower the differentiability at that point.

Periodic B-splines have a simple matrix formulation (which converts the curve to the power form). The main advantage of this is realized if the matrix multiplication is handled by hardware.

[FarR87], [FarR88], and [Faro91] showed that the Bernstein form of Bézier curves is numerically more stable than the power form, the caveat being however that one should not convert between the two. One would have to do **everything**, including algorithms and how the data is stored, in the barycentric form. See [DanD89].

Differences between Hermite and B-spline reprentations (see [Fari97])**:**

(1) B-splines are numerically more stable.
(2) Hermite basis functions are not invariant under affine parameter changes.
(3) B-splines use less storage. For the interpolation of n points, the B-spline curve needs n + 2 control points plus n + 1 knots (assuming that multiple knots are stored just once), whereas the Hermite curve needs 2n points plus n + 1 knots.

Next, some comments about NURBS curves.

Advantages to using NURBS curves:

(1) They can represent a wide variety of objects and enable a uniform approach. Using them one only needs to support **one** type of object. Their ability to represent conics is extremely important for CAD. They are supported by many standards such as IGES and STEP.
(2) Many tools exist for manipulating their geometry, such as, insertion of knots, etc.
(3) They support modification of local geometry.
(4) They are easy to transform under scaling, rotation, translation, shear, parallel and perspective projection.
(5) In general, they have pretty much all the same nice properties as ordinary B-splines.

Disadvantages to using NURBS curves:

(1) They are more complicated to compute in comparison to special types such as circles, spheres, etc. On the other hand, current algorithms are fast and numerically stable.

(2) They take more storage. For example, a circle takes seven control points and ten knots.
(3) They can produce bad parameterizations.

Finally, we point out to the interested reader some topics that were omitted in this chapter (see, for example, [HosL93]):

(1) Splines in tension: These are interpolating splines that try to dampen unwanted oscillations and extraneous inflection points. See also [Niel74] or [BaBB87].
(2) Shape preserving splines: If data is monotone or convex, then so should be the spline.
(3) Various "geometric" spline curves

11.16 Exercises

Section 11.2.1

11.2.1.1 Determine the Lagrange polynomials through the following sequences of points:

(a) (0,5), (1,2), (3,2)
(b) (–1,1), (0,–1), (1,–1), (2,1)

Section 11.2.2

11.2.2.1 Prove the validity of equation (11.21) by a direct argument, that is, let

$$p_i(x) = a + bx + cx^2 + dx^3$$

and solve for the coefficients a, b, c, and d as in the proof of Lemma 11.2.2.1.

11.2.2.2 Prove equation (11.24).

Section 11.2.3

11.2.3.1 Use equations (11.25) and (11.33) to find the cubic spline curve p(u) that has knots 0, 1, 2, and 3 and that passes through the points (0,0), (2,2), (4,2), and (6,4). Assume that p′(0) = (1,3) and p′(3) = (1,6).

Section 11.3

11.3.1 Express the curve

$$p(u) = \left(3u + 2, -u^3 + 5, 2u^3 + 7u^2 - 3\right)$$

in the form of equation (11.34).

11.3.2 Consider the geometric coefficient matrix

$$B_h = \begin{pmatrix} 1 & 3 & 1 \\ 7 & 3 & 1 \\ 18 & 0 & 40 \\ 18 & 0 & -40 \end{pmatrix}$$

for a curve p(u). Analyze p(u) geometrically like we did in Example 11.3.1. In particular, try to sketch the curve without computing its analytic formula.

11.3.3 Check that formula (11.42) for the interpolating cubic curve works when $\mathbf{p}_0 = (0,0)$, $\mathbf{p}_1 = (2,0)$, $\mathbf{p}_2 = (3,2)$, and $\mathbf{p}_3 = (2,3)$.

11.3.4 Find the matrix $\mathbf{M}$ for cubic curves p(u) with domain [0,2] that corresponds to the Hermite matrix $\mathbf{M}_h$ for curves with domain [0,1] so that the equation

$$p(u) = \mathbf{UM}\begin{pmatrix} p(0) \\ p(2) \\ p'(0) \\ p'(2) \end{pmatrix}$$

holds for all such curves.

Section 11.4

11.4.1 Show that the Bézier basis functions $B_{i,n}(u)$ satisfy the recurrence relation

$$B_{0,0}(u) = 1$$
$$B_{i,n}(u) = (1-u)B_{i,n-1}(u) + u\,B_{i-1,n-1}(u), \quad 0 \le i \le n,\ n > 0.$$

11.4.2 Let p(u) be a Bézier curve.

(a) Sketch the value p(2/3) if the control points are (0,0), (2,3), (6,2).
(b) Sketch the value p(3/4) if the control points are (0,0), (2,3), (6,2), (7,5).
(c) Sketch the value p(3/4) if the control points are (0,0), (2,3), (2,3), (6,2), (7,5).

Section 11.5.1

11.5.1.1 Compute the B-spline of order 3 that is defined by equation (11.72) if it has control points (0,0), (2,3), (6,2), (7,5) and knot vector (1,2,4,5,7,8,9).

11.5.1.2 Prove equation (11.74).

Section 11.5.2

11.5.2.1 The blossom of a Bézier curve p(u) with control points $\mathbf{p}_i$ can be obtained using the following generalization of the de Casteljau algorithm (see [Rams89]): Rather than

using the same parameter u at each stage in Step 2 of Algorithm 11.5.2.1, use a different name for the parameter at each stage, so that $\mathbf{p}_i^r$ would become a function of r variables. At the end, $\mathbf{p}_0^n$ would be the blossom. For example, in the cubic case we would have

$$\begin{array}{llll} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \\ & \mathbf{p}_0^1(u_1) & \mathbf{p}_1^1(u_1) & \mathbf{p}_2^1(u_1) \\ & & \mathbf{p}_0^2(u_1, u_2) & \mathbf{p}_1^2(u_1, u_2) \\ & & & \mathbf{p}_0^3(u_1, u_2, u_3) \end{array}$$

with $\mathbf{p}_0^3(u_1,u_2,u_3)$ the blossom of p(u). Check that this works for the quadratic and cubic Bézier curve case.

11.5.2.2 Consider the B-spline curve p(u) of order 3 with control points (1,2), (2,3), (5,1), (6,2) and knot vector (0,0,0,1,4,4,4) defined by equation (11.95). Find the three control points $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$ so that the quadratic Bézier curve q(u) defined on [0,1] with those control points generates the same curve as p(u)|[1,4].

11.5.2.3 Again consider the quadratic B-spline curve p(u) from Exercise 11.5.2.2. Find the control points for the B-spline curve that is obtained after inserting the knot t = 2.

Section 11.5.4

11.5.4.1 Prove equation (11.107).

Section 11.9

11.9.1 Consider the curves p: $[0,1] \to \mathbf{R}^2$ and q: $[0,1] \to \mathbf{R}^2$ defined by

$$p(t) = \left(u^2 - 2u + 1, u^3 - 3u + 3\right) \quad \text{and} \quad q(t) = \left(u^2, u^3 + 1\right).$$

Clearly, p(1) = (0,1) = q(0). Show that the composite curve is C^1 but not G^1.

11.17 Programming Projects

1. Display and manipulation of curves (Sections 11.4, 11.5.1, and 11.5.3)

Rather than listing specific projects, we shall simply suggest that the reader implement any one of the many types of curves we have described in this chapter and provide a user interface that allows one to manipulate the parameters associated to them. Of special interest are the many B-spline curves. Allow comparison of the curves by displaying several modifications of a curve simultaneously. Try to limit the amount of typing one has to do for the program and allow as much geometric input using the mouse as possible.

2. Interpolation (Sections 11.2.3 and 11.5.5)

Implement some interpolating splines for user picked or defined points.

3. Fairing of curves (Sections 11.10 and 11.12)

Another interesting project is on the shapes of curves and giving the user the ability to fair the curves. A starting point for this would be to plot the curvature graphs of a curve.

4. Frenet frames and parallel transports (Section 11.13)

Display the Frenet frames and parallel transports on a curve at specified intervals along the curve.

5. Recursive subdivision (Section 11.14)

Allow a user to define a planar polygonal curve and then show how it changes with recursive subdivision.

CHAPTER 12

Surfaces in Computer Graphics

Prerequisites: Basic calculus and vectors
Sections 8.1–5 in [AgoM05] (parameterization, surface, tangent vectors/planes, manifolds)
Chapter 9 in [AgoM05] (curvature related properties, cyclides)

Terminology. See the important comments on terminology for curves at the beginning of Chapter 11. Basically, the term "surface" by itself means a **set** and if it is preceded by an adjective, like in "parametric surface" or "Bézier surface" we mean a function. By a surface *patch* we shall mean a parameterized surface whose parameterization is defined by a **single** formula.

12.1 Introduction

This chapter describes the main parametric and implicit surfaces that one encounters in computer graphics and geometric modeling. These are the surfaces that are often referred to as "sculptured" or "free-form" surfaces to distinguish them from the piecewise linear ones. By in large, the material is a natural extension of what was developed for curves in the last chapter. For that reason it is important that the reader understand the material in that chapter before starting this one. Many facts and algorithms dealing with surfaces are easy if one understands their curve analogs.

Tangent planes for surfaces, in particular their normals, play an important role just like the tangent lines did for curves. In the parametric case the planes will be defined from certain partial derivatives associated to the parameterization. In the implicit case, they will be defined from their equations.

There are many types of smooth surfaces and different ways to represent them. Some representations involve interpolating data and others only approximate it. A helpful way to organize the subject is by categorizing the general principles motivating the various representations. Using terminology not normally used by mathematicians, but common in practice with engineers and craftsmen, three such principles are lofting, superposition, and a tensor or Cartesian product representation. Ruled

surfaces, Coons surfaces, and Bézier or B-spline surfaces are examples of these three principles, respectively.

Lofting is sometimes considered as a special case of a more general surface construction referred to as a *directrix-generator* representation. This is a representation that describes a surface in terms of sweeping a generator curve along a set of guidelines. The three ingredients for such a representation are

(1) a set of longitudinal curves called *directrices* (also called *meridians* in mathematics),
(2) a *correspondence rule* that relates each point of a directrix with a unique point on every other directrix, and
(3) a *generator rule* that defines a curve through all the points on the directrices that are related by the correspondence rule.

Figure 12.1 shows an example of how a surface can be defined by a directrix-generator type of construction. It is a traditional conic lofting example of an airplane fuselage. The directrices are conics and the generator consists of two conics, one above the maximum width line and one below. The correspondence rule relates points with equal x-values. Many surfaces can be defined by means of a directrix-generator construction. [Sabi90] gives a number of explicit examples. In particular, ruled surfaces and surfaces of revolution clearly fall into this class of surfaces. The directrices for the ruled surface are the bounding curves and the generators are the straight-line segments that connect the corresponding points on these curves. A surface of revolution has a single directrix, namely, the curve being rotated, and the generators are circles. The idea of a directrix-generator representation can be generalized to allow directrices to act as control points rather than being interpolated.

One of the driving forces behind the existence of the many types of surfaces is to make it easy for users (and programmers for that matter) to define the ones they need. Providing interactive descriptions of sets of three-dimensional points is much harder than for sets in the plane. One of the obvious reasons is that one is trying to describe three dimensions on a two-dimensional medium, the computer screen. To make geometric modeling programs "user friendly" one wants to avoid making a user enter complicated mathematical equations. One common approach to accomplish this is to let users define surfaces from one-dimensional sets using natural operations. For example, to define a surface of revolution a user needs only specify the curve to revolve and the axis of rotation.

Sections 12.2–12.12.1 describe some of the well-known basic surface types starting with the simple and intuitively easy to understand surfaces and working up to the more complicated ones. In Section 12.12.2 we show how the multiaffine approach to

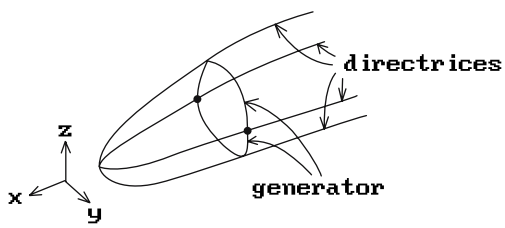


Figure 12.1. A fuselage as a directrix-generator surface.

curves carries over to surfaces. Although parameterizations typically have rectangular domains, there are times when triangular domains are more convenient and Section 12.12.3 discusses those. Rational B-spline and NURBS surfaces are defined in Section 12.12.4 and efficient evaluation algorithms for both B-spline and NURBS surfaces are discussed in Section 12.12.5. Section 12.12.6 is on interpolation using B-spline surfaces. Section 12.13 defines the very special cyclide surfaces. Sections 12.14–12.16 revisit, in the context of surfaces, some of the topics we encountered with curves in Sections 11.8–11.12. We discuss the subdivision of surfaces into smaller patches, the addition of control points and knots, composite surfaces, and fairing surfaces. Next, in Section 12.17, we switch from smooth surfaces and describe the class of polygonal surfaces defined by recursive subdivision. Section 12.18 gives a summary of some of the main points to remember when it comes to curves and surfaces and we finish with a few historical comments in Section 12.19.

12.2 Surfaces of Revolution

Surfaces of revolution are a frequently encountered type of surface. Spheres and cylinders can be thought of as surfaces of revolution. In general, one gets an "object of revolution" by revolving a set about some arbitrary axis. To analyze what this means in more detail, consider the simplest object to revolve, namely, a point. In that case one gets a circle in a plane orthogonal to the axis with center on the axis and radius equal to the distance of the point to the axis. It follows that one can think of an object of revolution as consisting of a union of circles centered on the axis, one for each point of the object being revolved. This also suggests that a way to parameterize a point $\mathbf{p}$ of an object gotten by revolving a curve about an axis is to use two parameters. One parameter is the parameter of the point on the curve, which gave rise to $\mathbf{p}$ and the other is the angle through which it was rotated. For general objects of revolution we would need k + 1 parameters, where k is the number of parameters needed to parameterize the object being revolved.

Our actual definition of a surface of revolution, which will be in terms of a parameterization, will restrict itself to the case where a curve is being revolved about the x-axis. This will simplify the definition. Besides, one can get surfaces of revolution about an arbitrary axis from this using rigid motions. Another simplifying hypothesis will be to assume that the curve lies in the x-y plane. Extending the definition to allow arbitrary space curves is left as an exercise for the reader (see Exercise 12.2.1.). One should note however that although it is easy to define an "object" of revolution, it is not easy to guarantee that the result will be a surface. We shall see that even in the special cases we shall analyze it is not trivial to ensure that the result will not have any singularities.

Definition. Let $g : [a,b] \to \mathbf{R}^2$ be a planar parametric curve and let $g(t) = (g_1(t), g_2(t))$. Define a function

$$p : [a,b] \times [c,d] \to \mathbf{R}^3$$

by

$$p(t,\theta) = (g_1(t), g_2(t)\cos\theta, g_2(t)\sin\theta). \tag{12.1}$$

The subset $\mathbf{X} = p([a,b] \times [c,d]) \subseteq \mathbf{R}^3$ is called a *surface of revolution* about the x-axis for angles c to d with respect to g. The curves $\gamma(t) = p(t,\theta)$ for fixed θ are the *meridians* of the surface of revolution and the curves $\eta(\theta) = p(t,\theta)$ for fixed t are called the *circles of latitude*. If $[c,d] = [0,2\pi]$, then $\mathbf{X}$ is called a *full surface of revolution*. The function p is called the *standard parameterization of* $\boldsymbol{X}$ *with respect to g*.

Figure 12.2 shows the important special case where we revolve the graph of a real-valued function $f : [a,b] \to \mathbf{R}$. Using the standard parameterization $g(t) = (t,f(t))$ of the graph of f and replacing t by x in equation (12.1), the surface obtained by revolving the graph of f about the x-axis is parameterized by

$$p(x,\theta) = (x, f(x)\cos\theta, f(x)\sin\theta). \tag{12.2}$$

Partial derivatives are easy to compute for this surface:

$$\frac{\partial p}{\partial x} = (1, f'(x)\cos\theta, f'(x)\sin\theta) \tag{12.3}$$

$$\frac{\partial p}{\partial \theta} = (0, -f(x)\sin\theta, f(x)\cos\theta) \tag{12.4}$$

From this one right away knows the tangent planes at every point, because the cross product of the partial derivatives is a normal vector (assuming that the partial derivatives do not vanish).

Although surfaces of revolution are conceptually easy to describe, there are some potentially nasty details that can give a programmer a lot of headaches. Some problems are:

Degenerate Cases of the Curve g. The curve may have self-intersections. If g is a constant map, then the "surface" becomes a circle.

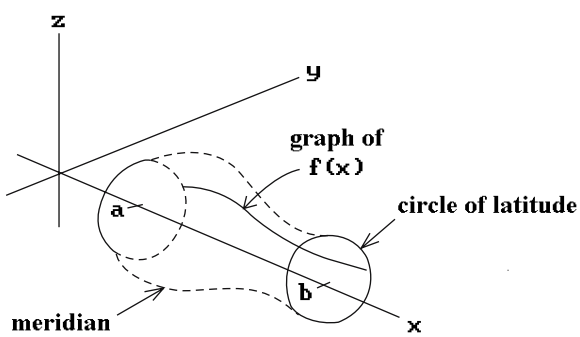


Figure 12.2. A surface of revolution.

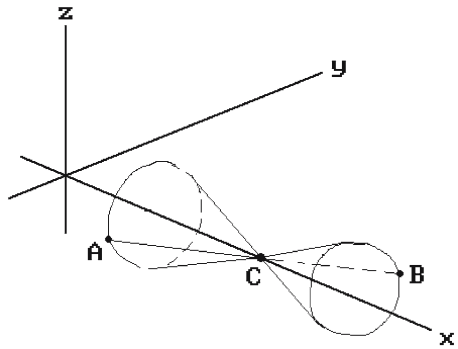


Figure 12.3. A surface of revolution tangent plane problem.

The Curve Crosses the X-axis. For example, the surface of revolution obtained by rotating the segment [**A**,**B**] in Figure 12.3 about the x-axis has no tangent plane at **C** where the curve crossed the axis. Even if a tangent plane exists at such points, such as when one revolves the upper half of the unit circle about the x-axis to get the unit sphere $\mathbf{S}^2$, problems may arise because the standard approach to getting the tangent plane using the partials of the parameterization may fail. See Exercise 12.2.2.

Choosing a Direction for the "Outward" Normal. The direction would most likely depend on the orientation of the curve, but the curve may zigzag.

The main problems are typically caused by partial derivatives vanishing so that it is messy trying to find the tangent plane at a point of the surface.

Next, we look at a number of important special cases of surfaces of revolution and work out some concrete examples.

Consider the full surface of revolution **S** obtained by rotating a line segment **X** about the x-axis.

Case 1: If **X** is parallel to axis, then **S** is a **cylinder**. See Figure 12.4(a).
Case 2: If **X** is skew to axis, then **S** is a **truncated cone**. See Figure 12.4(b).
Case 3: If **X** is orthogonal to the axis, then **S** is an **annulus**. See Figure 12.4(c).

12.2.1 Example. Assume that **S** is the surface of revolution obtained by rotating the segment $\mathbf{X} = [(0,1),(2,3)]$ about the x-axis. We want to find the tangent plane to **S** at $\mathbf{p} = (1,\sqrt{2},\sqrt{2})$.

Solution. The function

$$p(x,\theta) = (x,(x+1)\cos\theta,(x+1)\sin\theta)$$

parameterizes **S** and

$$\frac{\partial p}{\partial x}(x,\theta) = (1,\cos\theta,\sin\theta) \quad \text{and} \quad \frac{\partial p}{\partial \theta}(x,\theta) = (0,-(x+1)\sin\theta,(x+1)\cos\theta).$$

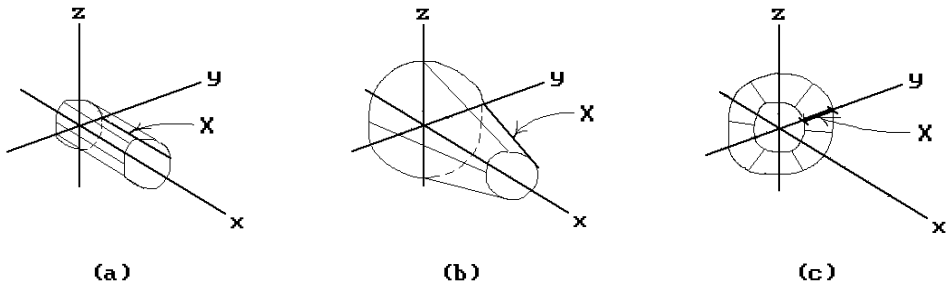


Figure 12.4. Interesting surfaces of revolution.

Since $p(1,\pi/4) = \mathbf{p}$, it follows that

$$\frac{\partial p}{\partial x}(1,\pi/4) = (1,1/\sqrt{2},1/\sqrt{2}) \quad \text{and} \quad \frac{\partial p}{\partial \theta}(1,\pi/4) = (0,-\sqrt{2},\sqrt{2})$$

are a basis for our tangent plane. Finally,

$$\frac{\partial p}{\partial x}(1,\pi/4) \times \frac{\partial p}{\partial \theta}(1,\pi/4) = (2,-\sqrt{2},-\sqrt{2})$$

is a normal vector to the plane, so that its equation is

$$(2,-\sqrt{2},-\sqrt{2}) \bullet ((x,y,z) - (1,\sqrt{2},\sqrt{2})) = 0,$$

or

$$2x - \sqrt{2}y - \sqrt{2}z + 2 = 0.$$

12.2.2 Example. Assume that **S** is the surface of revolution obtained by rotating the segment $\mathbf{X} = [(3,1),(3,3)]$ about the x-axis. We want to find the tangent plane to **S** at $\mathbf{p} = (3,0,2)$.

Solution. The surface **S** is parameterized by

$$p(y,\theta) = (3, y\cos\theta, y\sin\theta)$$

and

$$\frac{\partial p}{\partial y}(y,\theta) = (0,\cos\theta,\sin\theta) \quad \text{and} \quad \frac{\partial p}{\partial \theta}(y,\theta) = (0,-y\sin\theta,y\cos\theta).$$

Since $p(2,\pi/2) = \mathbf{p}$, it follows that

$$\frac{\partial p}{\partial y}(2,\pi/2) = (0,0,1) \quad \text{and} \quad \frac{\partial p}{\partial \theta}(2,\pi/2) = (0,-2,0)$$

are a basis for our tangent plane. Finally,

$$\frac{\partial p}{\partial y}(2,\pi/2) \times \frac{\partial p}{\partial \theta}(2,\pi/2) = (2,0,0)$$

is a normal vector to the plane, so that its equation is

$$(2,0,0) \bullet ((x,y,z) - (3,0,2)) = 0,$$

or

$$x - 3 = 0.$$

Next, if we rotate a semicircle about an axis whose endpoints lie on the axis, then we get a **sphere**. In the case of the semicircle of radius r about the origin, we can parameterize its points with the map

$$\phi \to (r\cos\phi, r\sin\phi) \quad \text{for} \quad \phi \in [0,\pi].$$

This leads to the following parameterization of the sphere of radius r about the origin:

$$p(\phi,\theta) = (r\cos\phi, r\sin\phi\cos\theta, r\sin\phi\sin\theta). \tag{12.5}$$

See Figure 12.5(a). The partial derivatives

$$\begin{aligned} \frac{\partial p}{\partial \phi}(\phi,\theta) &= (-r\sin\phi, r\cos\phi\cos\theta, r\cos\phi\sin\theta) \\ \frac{\partial p}{\partial \theta}(\phi,\theta) &= (0, -r\sin\phi\sin\theta, r\sin\phi\cos\theta) \end{aligned} \tag{12.6}$$

define the tangent planes except at the two poles $(\pm r,0,0)$ where they vanish. The tangent planes at those two points have to be handled as a special case unfortunately.

If we rotate a circle about an axis and if this circle does not meet the axis, then we get a **torus**. As a special case, let **T** be the torus obtained by rotating the circle of radius r with center (0,R,0), $r < R$, about the x-axis. See Figure 12.5(b). Here is another natural way to visualize the standard parameterization of **T**. Let $\mathbf{P}_\theta$ be the plane through the x-axis that makes an angle θ with the x-y plane. This plane intersects **T** in a circle $\mathbf{C}_\theta$ with center $R\mathbf{u}_\theta$, where $\mathbf{u}_\theta = (0,\cos\theta, \sin\theta)$ is the unit vector in the y-z plane that makes an angle θ with the y-axis. Parameterizing the points of $\mathbf{C}_\theta$ by the angle ϕ that the ray from the center of $\mathbf{C}_\theta$ makes with the x-axis corresponds to the map

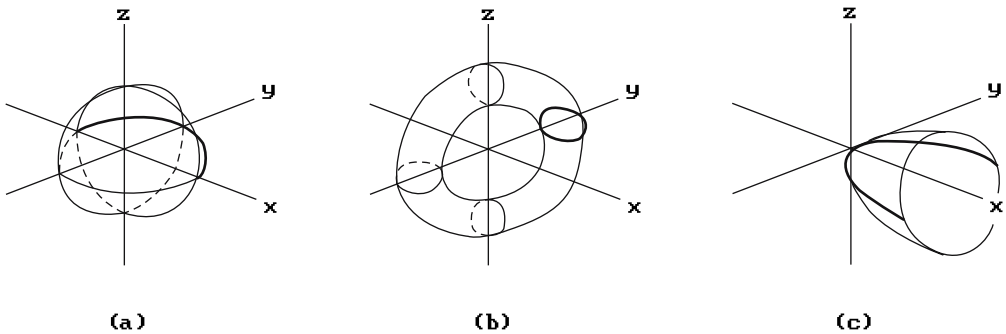


Figure 12.5. Surfaces of revolution: sphere, torus, and paraboloid.

$$\phi \to R\mathbf{u}_\theta + (r\cos\phi)\mathbf{e}_1 + (r\sin\phi)\mathbf{u}_\theta.$$

This induces the following parameterization $p(\phi,\theta)$ of $\mathbf{T}$

$$\begin{aligned} p(\phi,\theta) &= R\mathbf{u}_\theta + (r\cos\phi)\mathbf{e}_1 + (r\sin\phi)\mathbf{u}_\theta \\ &= (r\cos\phi,(R+r\sin\phi)\cos\theta,(R+r\sin\phi)\sin\theta) \end{aligned} \tag{12.7}$$

with partial derivatives

$$\begin{aligned} \frac{\partial p}{\partial \phi}(\phi,\theta) &= (-r\sin\phi, r\cos\phi\cos\theta, r\cos\phi\sin\theta) \\ \frac{\partial p}{\partial \theta}(\phi,\theta) &= (0,-(R+r\sin\phi)\sin\theta,(R+r\sin\phi)\cos\theta). \end{aligned} \tag{12.8}$$

12.2.3 Example. We want to find the normal vector and tangent plane to the torus defined by equation (12.7) with r = 2 and R = 5 at $\mathbf{p} = (0,0,7)$.

Solution. Since $p(\pi/2,\pi/2) = \mathbf{p}$, equations (12.8) implies that

$$\frac{\partial p}{\partial \phi}(\pi/2,\pi/2) = (-2,0,0) \quad \text{and} \quad \frac{\partial p}{\partial \theta}(\pi/2,\pi/2) = (0,-7,0)$$

are a basis for the tangent plane. It follows that $(0,0,14) = (-2,0,0) \times (0,-7,0)$ is a normal vector and the plane has equation

$$(0,0,14) \bullet ((x,y,z) - (0,0,7)) = 0,$$

or

$$z - 7 = 0.$$

If we rotate half of a parabola about its axis, then we get a *paraboloid of revolution* (also called an elliptic paraboloid). See Figure 12.5(c). If we do the same thing to a hyperbola, we get a *hyperboloid of revolution* (also called a *hyperboloid of one sheet*).

12.2.4 Example. Let **S** be the paraboloid of revolution obtained by rotating the part of the parabola $x = y^2$, $y \geq 0$, about the x-axis. We want to find the tangent plane and normal to **S** at $\mathbf{p} = (4,0,2)$.

Solution. The standard parameterization for **S** is

$$p(x,\theta) = (x, \sqrt{x}\cos\theta, \sqrt{x}\sin\theta).$$

Since $p(4,\pi/2) = (4,0,2)$ and

$$\frac{\partial p}{\partial x}(x,\theta) = \left(1, \frac{1}{2\sqrt{x}}\cos\theta, \frac{1}{2\sqrt{x}}\sin\theta\right)$$
$$\frac{\partial p}{\partial \theta}(x,\theta) = (0, -\sqrt{x}\sin\theta, \sqrt{x}\cos\theta),$$

evaluating these vectors at $(4,\pi/2)$ and taking the cross product gives us that $(1/2,0,-2)$ is a normal vector, so that

$$(1/2,0,-2) \bullet ((x,y,z) - (4,0,2)) = 0,$$

or

$$x - 4z + 4 = 0,$$

is the equation for the tangent plane.

12.3 Quadric Surfaces and Other Implicit Surfaces

Like the conic curves, quadric surfaces are an important shape for CAGD. A quadric surface is a subset of points (x,y,z) in $\mathbf{R}^3$ which satisfies a general quadratic equation of the form

$$ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0. \tag{12.9}$$

A complete classification of the solutions to equation (12.9) can be found in Section 3.7 in [AgoM04] and we shall not repeat it here. Omitting the degenerate cases, one gets the basic ellipsoids, cylinders, cones, paraboloids, and hyperboloids. Figure 12.5(a) and (c) showed a sphere (a special case of an ellipsoid) and paraboloid. Figure 12.6 shows examples of the others. Clearly, many mechanical parts have such shapes. In an interesting paper, Goldman ([Gold83]) analyzes quadrics that are surfaces of

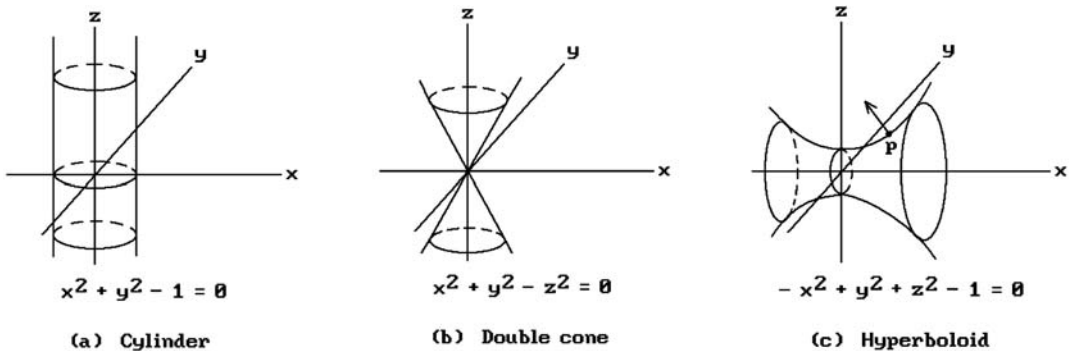


Figure 12.6. Three quadric surfaces.

revolution (like the cyclinder, ellipsoid, paraboloid, cone, and hyperboloids) and shows that they make good candidates for primitives in modeling systems because one can classify their intersections.

A fundamental question for a quadric surface like for all surfaces is how one can find a normal vector and an equation for the tangent plane at a point. Since many of these surfaces are surfaces of revolution, we could apply the methods developed in the last section. We could also use the fact that these surfaces can be parameterized and tangent planes are easily computed from parameterizations. Here, rather than using either of these two approaches, we shall describe another powerful method. Quadric surfaces are a special case of implicit surfaces. An implicit surface $\mathbf{S}$ in $\mathbf{R}^3$ is a surface that is the set of zeros of a function $f : \mathbf{R}^3 \to \mathbf{R}$, that is,

$$\mathbf{S} = \{\mathbf{p} \in \mathbf{R}^3 \mid f(\mathbf{p}) = 0\}.$$

It is a well-known fact that if $\mathbf{p} \in \mathbf{S}$, then the gradient to f at $\mathbf{p}$, $\nabla f(\mathbf{p})$, is a normal vector to $\mathbf{S}$ at $\mathbf{p}$ (assuming that the function f is differentiable). This means that computing normals for implicit surfaces is trivial. Of course, once one has the normal, then the tangent plane is easily expressed in the point-normal form.

12.3.1 Example. We want to find the tangent plane to the hyperboloid $\mathbf{S}$ defined by

$$-x^2 + y^2 + z^2 = 1$$

at $\mathbf{p} = (\sqrt{3},0,2)$. See Figure 12.6(c).

Solution. Let

$$f(x,y,z) = -x^2 + y^2 + z^2 - 1.$$

Since $\nabla f(x,y,z) = (-2x,2y,2z)$, $\nabla f(\mathbf{p}) = (-2\sqrt{3},0,4)$ is a normal vector to **S** at **p**. It follows that the tangent plane is defined by

$$(-2\sqrt{3},0,4) \bullet ((x,y,z) - (\sqrt{3},0,2)) = 0$$

or

$$-\sqrt{3}x + 2z - 7 = 0.$$

From the point of view of finding normals to surfaces it is therefore advantageous to have an implicit representation of it. Unfortunately, except for quadric surfaces, surfaces are usually presented via parameterizations. Finding an implicit representation given a parameterization is a nontrivial problem that is addressed in Chapter 10 in [AgoM04].

It should be pointed out that many surfaces of revolution are implicit surfaces, so that the gradient method for finding normals and tangent planes applies to them. For example, consider the surface of revolution defined by equation (12.2). It is easy to show that this surface is the set of zeros of the function

$$F(x,y,z) = y^2 + z^2 - f^2(x). \tag{12.10}$$

12.3.2 Example. We rework Example 12.2.4 using this approach. Using equation (12.10), the surface is defined by the equation

$$F(x,y,z) = y^2 + z^2 - x = 0.$$

But $\nabla F(x,y,z) = (-1,2y,2z)$ and $\nabla F(4,0,2) = (-1,0,4)$. The latter is, up to a scalar multiple, the same normal as the one we got before.

Finally, we mention Barr's ([Barr81]) superquadric surfaces. These are the surface analogs of superellipses and are useful in representing shapes such as rounded blocks or rounded square toroids. They are defined by trigonometric functions raised to exponents. [Barr92] presents expressions for their volume, center of mass, and rotational inertia tensor.

12.4 Ruled Surfaces

Ruled surfaces are probably the next simplest surfaces after planes. Special cases of these are extrusions. These are surfaces obtained by sweeping a vector along a curve.

Definition. Given a curve $f : [a,b] \to \mathbf{R}^3$ and a vector $\mathbf{v} \in \mathbf{R}^3$, the parametric surface

$$p : [a,b] \times [0,1] \to \mathbf{R}^3$$

defined by

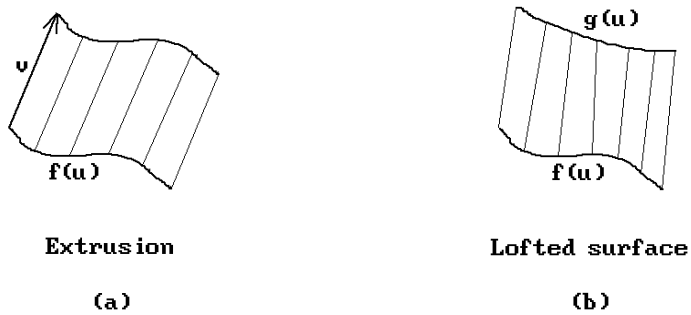


Figure 12.7. Extrusions and lofted surfaces.

$$p(u,t) = f(u) + t\mathbf{v} \tag{12.11}$$

is called an *extrusion*. The vector $\mathbf{v}$ is called the *sweep vector* for the extrusion.

See Figure 12.7(a). The partials are again easy to compute:

$$\frac{\partial p}{\partial u} = f'(u) \quad \text{and} \quad \frac{\partial p}{\partial t} = \mathbf{v}. \tag{12.12}$$

Definition. Given two curves f, g : $[a,b] \to \mathbf{R}^3$, the parametric surface

$$p : [a,b] \times [0,1] \to \mathbf{R}^3$$

defined by

$$p(u,v) = (1-v)f(u) + vg(u). \tag{12.13}$$

is called a *lofted surface*.

Lofted surfaces are ruled surfaces that interpolate two curves. See Figure 12.7(b). The partials are

$$\frac{\partial p}{\partial u} = (1-v)f'(u) + vg'(u) \quad \text{and} \quad \frac{\partial p}{\partial t} = g(u) - f(u). \tag{12.14}$$

Note that extrusions are a special case of a lofted surface. In both the case of extrusions and the case of lofted surfaces, one needs to worry about self-intersections. Of the nondegenerate quadric surfaces, the cylinders and cones are ruled surfaces. The hyperboloid of one sheet and the hyperbolic paraboloid are doubly ruled surfaces. For a more thorough discussion of ruled surfaces that includes algorithms for computing planar intersections and contour outlines see [PePR99].

One question that sometimes arises in the context of ruled surfaces is whether

they are developable. Recall that a surface is developable if it can be "unrolled" into a plane. Neither the hyperboloid of one sheet nor the hyperbolic paraboloid is developable. In general, if a surface is developable, then the Gauss curvature must be zero. See Chapter 9 in [AgoM05]. In engineering, developable and nondevelopable surfaces are sometimes called *singly* and *doubly curved surfaces*. Developable surfaces are desirable in manufacturing because they can be bent without stretching or tearing; however, for a number of reasons one has to be able to deal with nondevelopable surfaces also and so the question arises as to how one can best flatten a nondevelopable surface. One sample paper that addresses this issue is [YuPM00].

12.5 Sweep Surfaces

Sweep surfaces are a special case of generative models that were introduced in Section 5.3.5. They can be thought of as the envelope of the volume that we get when sweeping a set (curve or volume) along a curve. In that sense, surfaces of revolution and ruled surfaces are sweep surfaces. In this section we describe two variants that are encountered in practice.

Assume that we are given

(1) a curve $\gamma : [a,b] \to \mathbf{R}^3$,
(2) a coordinate system or frame $(\mathbf{u}_1(s),\mathbf{u}_2(s),\mathbf{u}_3(s))$ at every point s along the curve $\gamma(s)$ (the Frenet frame of the curve is a possible such coordinate system), and
(3) another curve $f : [c,d] \to \mathbf{R}^3$, $f(t) = (f_1(t),f_2(t),f_3(t))$.

Define

$$p : [a,b] \times [c,d] \to \mathbf{R}^3 \tag{12.15a}$$

by

$$p(s,t) = \gamma(s) + f_1(t)\mathbf{u}_1(s) + f_2(t)\mathbf{u}_2(s) + f_3(t)\mathbf{u}_3(s). \tag{12.15b}$$

See Figure 12.8.

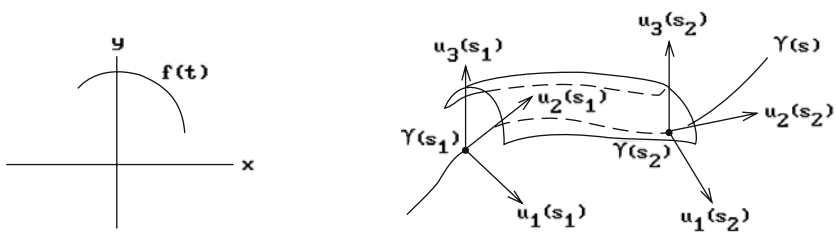


Figure 12.8. Sweeping a curve along a framed curve.

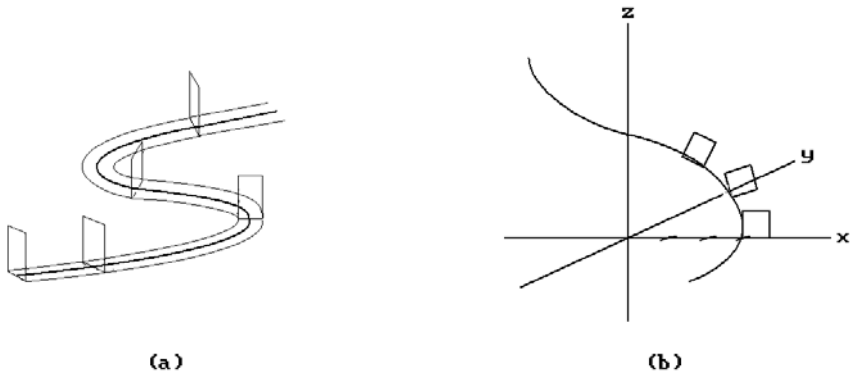


Figure 12.9. Sweep surfaces: a cutter path and square rotated along a spiral.

Definition. The parametric surface p(s,t) defined by equation (12.15b) is called the *sweep surface* obtained by sweeping f(t) along the framed curve $\gamma(s)$.

12.5.1 Example. The path of a cutter as shown in Figure 12.9(a) can easily be described as two sweep surfaces defined by equations like equation (12.15b), where each function f(t) parameterizes a vertical segment.

Equation (12.15b) can be generalized by letting a one-parameter family of affine maps T_s act on the curve f(t) as we sweep it along $\gamma(s)$. Define function $g_i(s,t)$ by

$$T_s(f(t)) = (g_1(s,t), g_2(s,t), g_3(s,t))$$

and the new parametric surface p(s,t) by

$$p(s,t) = \gamma(s) + g_1(s,t)\mathbf{u}_1(s) + g_2(s,t)\mathbf{u}_2(s) + g_3(s,t)\mathbf{u}_3(s). \tag{12.16}$$

Definition. The parametric surface p(s,t) defined by equation (12.16) is called a *screw sweep surface*.

12.5.2 Example. Let $\gamma(s)$ be the spiral

$$\gamma(s) = (3\cos s, 3\sin s, s).$$

Using the Frenet frame (T(s),N(s),B(s)) for this curve we let

$$\mathbf{u}_1(s) = N(s) = (-\cos s, -\sin s, 0),$$
$$\mathbf{u}_2(s) = T(s) = (1/\sqrt{10})(-3\sin s, 3\cos s, 1),$$
$$\mathbf{u}_3(s) = B(s) = (1/\sqrt{10})(\sin s, -\cos s, 3).$$

Let $f : [0,4] \to \mathbf{R}^3$ parameterize the unit square in the x–z plane as follows:

$$\begin{aligned} f(t) &= (t,0,0), & t \in [0,1], \\ &= (1,0,t-1), & t \in [1,2], \\ &= (3-t,0,1), & t \in [2,3], \\ &= (0,0,4-t), & t \in [3,4]. \end{aligned}$$

We shall rotate the square as it is swept along $\gamma(s)$. This can be accomplished by defining

$$T_s(x,y,z) = (x\cos s, y, z\sin s).$$

The functions $g_i(s,t)$ are easily computed. For example, for $t \in [1,2]$,

$$g_1(s,t) = \cos s, \quad g_2(s,t) = 0, \quad \text{and} \quad g_3(s,t) = (t-1)\sin s.$$

Figure 12.9(b) shows the resulting screw sweep surface.

12.6 Bilinear Surfaces

Let four points $\mathbf{p}_{00}$, $\mathbf{p}_{01}$, $\mathbf{p}_{10}$, and $\mathbf{p}_{11}$ be given. The easiest way to define a surface p(u,v) that interpolates these points is by means of a double linear interpolation. Define

$$\begin{aligned} p(u,v) &= (1-v)[(1-u)\mathbf{p}_{00} + u\mathbf{p}_{10}] + v[(1-u)\mathbf{p}_{01} + u\mathbf{p}_{11}], && (12.17a) \\ &= (1-u)[(1-v)\mathbf{p}_{00} + v\mathbf{p}_{01}] + u[(1-v)\mathbf{p}_{10} + v\mathbf{p}_{11}], && (12.17b) \\ &= (1-u)(1-v)\mathbf{p}_{00} + (1-u)v\mathbf{p}_{01} + u(1-v)\mathbf{p}_{10} + uv\mathbf{p}_{11}, \end{aligned}$$

where $0 \le u, v \le 1$. Equation (12.17a) says that to find p(u,v) we first find p(u,0) and p(u,1) by a linear interpolation in the u-direction and then do a linear interpolation of those points in the v-direction. Equation (12.17b) says that we get the same answer if we first interpolate in the v-direction and then in the u-direction. See Figure 12.10(a).

Definition. The parametric surface defined by equations (12.17) is called the *bilinear surface* determined by the four points or the *four-point interpolating surface*.

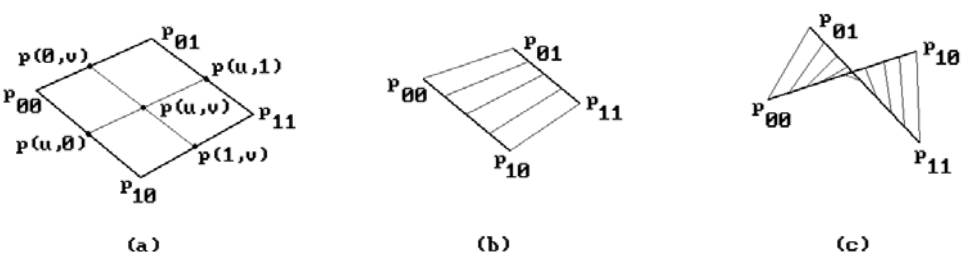


Figure 12.10. Bilinear surfaces.

The bilinear surface is a special case of a lofted surface and also of the Coons surface and the Bézier tensor product surface described later on. We can rewrite the function p(u,v) in matrix form as

$$p(u,v) = (1-u\ u)\begin{pmatrix} \mathbf{p}_{00} & \mathbf{p}_{01} \\ \mathbf{p}_{10} & \mathbf{p}_{11} \end{pmatrix}\begin{pmatrix} 1-v \\ v \end{pmatrix}.$$

If the points $\mathbf{p}_i$ are coplanar and linearly independent (as in Figure 12.10(b)), then the resulting surface is a quadrilateral, otherwise (as in Figure 12.10(c)) it is a surface of degree two, meaning that it is parameterized by quadratic polynomials.

12.6.1 Example. If $\mathbf{p}_{00} = (1,0,0)$, $\mathbf{p}_{01} = (1,3,-1)$, $\mathbf{p}_{10} = (-1,-2,2)$, and $\mathbf{p}_{11} = (-1,1,1)$, then it is easy to check that

$$p(u,v) = (1-2u, 3v-2u, 2u-v).$$

Note that the points all lie on the plane $2x + y + 3z = 1$.

12.6.2 Example. If we use the same points as in Example 12.6.1 but replace $\mathbf{p}_{00}$ by $\mathbf{p}_{00} = (0,0,0)$, then

$$p(u,v) = (v-u-uv, 3v-2u, 2u-v).$$

The points are no longer coplanar and we see a quadratic term "uv" in the formula for p(u,v).

A special case of the four-point interpolating surface arises when two of the points are the same. We are then simply parameterizing a triangular planar patch. However, it makes more sense in that case to use a triangular domain. We can do that if we use barycentric coordinates. Let $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ be three points and define

$$p(u,v) = u\mathbf{p}_0 + v\mathbf{p}_1 + (1-u-v)\mathbf{p}_2, \tag{12.18}$$

where $0 \le u,v$ and $0 \le u + v \le 1$. The function p(u,v) parameterizes the triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$. If we drop all the constraints on u and v, then we get a parameterization of the plane containing the three points called the *interpolating plane*. Note that equation (12.18) can be rewritten in the form

$$p(u,v) = \mathbf{p}_2 + u\mathbf{p}_2\mathbf{p}_0 + v\mathbf{p}_2\mathbf{p}_0$$

which shows more clearly that we are parameterizing a plane that has the vectors $\mathbf{p}_2\mathbf{p}_0$ and $\mathbf{p}_2\mathbf{p}_0$ for a basis.

12.7 Coons Surfaces

The surfaces described so far had a relatively simple description by means of a single formula. There are many other surfaces that one needs to deal with in CAGD, such

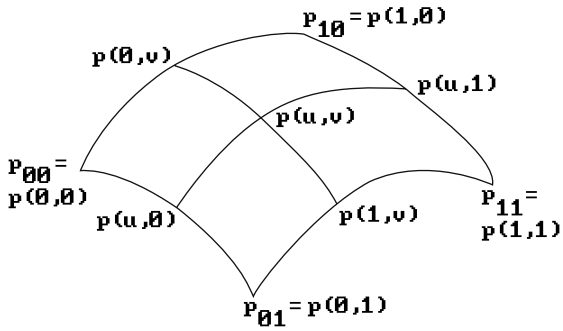


Figure 12.11. A Coons patch.

as car bodies or airplane fuselages, that do not. In these situations, a common approach is to divide the surface into patches defined by a network of curves. Each patch is then parameterized by "filling in" or interpolating the given parameterizations of their boundary. The boundary curves themselves would have been defined by an interpolation of data that comes from measurements on a model. In this section we describe some general methods for defining surfaces that match some given boundary data. Such parameterizations are sometimes called *transfinite interpolants* in the literature because one is interpolating an **infinite** set of points. All of the interpolation problems that we discussed previously (except for the case of lofted surfaces) only interpolated a **finite** set of points. Finally, the boundary data may involve more than just the curves however if one wants adjacent patches to meet in a smooth way.

To state our goal more precisely: it is to find a parameterization p(u,v), $u,v \in [0,1]$, that interpolates the four given boundary curves p(0,v), p(1,v), p(u,0), and p(u,1). See Figure 12.11. A well-known solution to this problem is due to Coons ([Coon67]). His approach builds on the lofted surface parameterizations for the boundary curve pairs p(u,0), p(u,1) and p(0,v), p(1,v). Define operators P_1 and P_2 on a function of two variables p(u,v) by

$$(P_1p)(u,v) = (1-u)p(0,v) + up(1,v), \tag{12.19a}$$

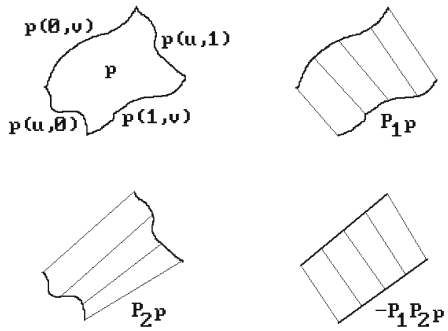
$$(P_2p)(u,v) = (1-v)p(u,0) + vp(u,1). \tag{12.19b}$$

Compare the formulas for P_1p and P_2p with equation (12.13). These functions interpolate the boundary curves in the v- and u-direction, respectively. If our boundary curves are straight lines parameterized in the standard linear way, then we would find that we simply have the bilinear surface defined by equations (12.17) and $P_1p(u,v)$ and $P_2p(u,v)$ are the same functions. If the boundary curves are not straight lines however, then $P_1p(u,v)$ and $P_2p(u,v)$ are not the same, nor do they interpolate the other boundary curves in the u- and v-direction, respectively. See Figure 12.12. For example, the differences between P_1p and the actual values of p along the u-direction boundaries are

$$p(u,0) - P_1p(u,0) \quad \text{and}$$

$$p(u,1) - P_1p(u,1).$$

Figure 12.12. A bilinearly blended Coons patch.



Interpolating this error term in the v-direction is just the function $P_2(p - P_1p)(u,v)$. If we were to arbitrarily define

$$p(u,v) = P_1p(u,v) + P_2(p - P_1p)(u,v) = P_1p(u,v) + P_2p(u,v) - P_2P_1p(u,v), \quad (12.20a)$$

then by construction this function p(u,v) would now interpolate our **entire** boundary. What would have happened if we had applied this argument to P_2p and its difference with the actual values of p along the v-direction boundaries? Fortunately, because the operators P_1 and P_2 commute, that is, $P_2P_1 = P_1P_2$, we would have arrived at the **same** formula (12.20a). As one can see from Figure 12.12, $P_1P_2p(u,v)$ is what is called a doubly ruled surface. Replacing the operators P_i in the formula for p(u,v) in equation (12.20a) by their definitions leads to the original Coons formula

$$\begin{aligned} p(u,v) &= (1-v)p(u,0) + vp(u,1) + (1-u)p(0,v) + up(1,v) \\ &\quad -(1-u)(1-v)p(0,0) - (1-u)vp(0,1) - u(1-v)p(1,0) - uvp(1,1). \end{aligned} \quad (12.20b)$$

Definition. The parametric surface defined by the function p(u,v) in equations (12.20) is called the (*bilinearly blended*) *Coons patch* or *Coons surface* for the curves p(0,v), p(1,v), p(u,0), and p(u,1).

The Coons surface can be expressed in matrix form as follows:

$$p(u,v) = (1-u\ u)\begin{pmatrix} p(0,v) \\ p(1,v) \end{pmatrix} + (1-v\ v)\begin{pmatrix} p(u,0) \\ p(u,1) \end{pmatrix} - (1-u\ u)\begin{pmatrix} p(0,0) & p(0,1) \\ p(1,0) & p(1,1) \end{pmatrix}\begin{pmatrix} 1-v \\ v \end{pmatrix} \quad (12.20c)$$

$$= (1-u\ u\ 1)\begin{pmatrix} -p(0,0) & -p(0,1) & p(0,v) \\ -p(1,0) & -p(1,1) & p(1,v) \\ p(u,0) & p(u,1) & \mathbf{0} \end{pmatrix}\begin{pmatrix} 1-v \\ v \\ 1 \end{pmatrix}. \quad (12.20d)$$

The basic Coons surface is easily generalized by replacing the simple linear blending functions by others. Let $b_0(t)$, $b_1(t)$, $c_0(t)$, and $c_1(t)$ be arbitrary real-valued functions on [0,1] and define new operators P_1 and P_2 on functions p(u,v) by

$$(P_1p)(u,v) = b_0(u)p(0,v) + b_1(u)p(1,v), \tag{12.21a}$$

$$(P_2p)(u,v) = c_0(v)p(u,0) + c_1(v)p(u,1). \tag{12.21b}$$

Notice how the definition of the functions $(P_ip)(u,v)$ only uses the boundary values of $p(u,v)$. These new equations reduce to the ones in equation (12.19) if we let $b_0(t) = c_0(t) = L_{0,1}(t)$ and $b_1(t) = c_1(t) = L_{1,1}(t)$, where $L_{0,1}(t) = 1 - t$ and $L_{1,1}(t) = t$ are the linear Lagrange basis functions defined in equation (11.2) of Section 11.2.1. In fact, the only conditions that the blending functions $b_i(t)$ and $c_i(t)$ have to satisfy in order to have the functions defined by equation (12.21) interpolate the v-, respectively, u-direction boundary are that

$$b_0(0) = 1, \quad b_0(1) = 0, \quad b_1(0) = 0, \quad b_1(1) = 1,$$

$$c_0(0) = 1, \quad c_0(1) = 0, \quad c_1(0) = 0, \quad c_1(1) = 1,$$

The operators P_1 and P_2 are best thought of as projection operators that project the space of vector-valued functions of two variables onto a subspace because $P_i^2 = P_i$. The sum of the two operators is not a projection since $(P_1 + P_2)^2 \neq P_1 + P_2$, but if they commute, that is, $P_1P_2p = P_2P_1p$, then the *Boolean sum operator* P defined by

$$Pp = (P_1 \oplus P_2)p = (P_1 + P_2 - P_1P_2)p \tag{12.22a}$$

is a projection. Equation (12.22a) can be expressed in matrix form as

$$(Pp)(u,v) = (b_0(u)\ b_1(u)\ 1)\begin{pmatrix} -p(0,0) & -p(0,1) & p(0,v) \\ -p(1,0) & -p(1,1) & p(1,v) \\ p(u,0) & p(u,1) & \mathbf{0} \end{pmatrix}\begin{pmatrix} c_0(v) \\ c_1(v) \\ 1 \end{pmatrix}. \tag{12.22b}$$

We summarize the main facts about the operator P.

12.7.1 Theorem. If the functions $b_i(u)$ and $c_i(v)$ and operators P_i in equations (12.21) satisfy

(1) $b_i(j) = c_i(j) = \delta_{ij}$, $i,j \in \{0,1\}$, and
(2) $P_1P_2p = P_2P_1p$ for all p,

then the parametric surface $(Pp)(u,v)$ defined by equations (12.22) interpolates the boundary curves $p(0,v)$, $p(1,v)$, $p(u,0)$, and $p(u,1)$. If also

(3) $b_i'(j) = c_i'(j) = 0$, $i,j \in \{0,1\}$,

then

$$\frac{\partial}{\partial u}(Pp)(u,v) = c_0(v)\frac{\partial p}{\partial u}(i,0) + c_1(v)\frac{\partial p}{\partial u}(i,1)$$

$$\frac{\partial}{\partial v}(Pp)(u,v) = b_0(v)\frac{\partial p}{\partial v}(i,0) + b_1(v)\frac{\partial p}{\partial v}(i,1),$$

for i = 0 or 1.

Proof. Exercise 12.7.2.

As a simple application of Theorem 12.7.1 we can get a smoother Coons surface by using the Hermite basis functions $H_{i,3}$ introduced in equation (11.18) of Section 11.2.2 and defining

$$p(u,v) = (H_{0,3}(u)\ H_{3,3}(u)\ 1)\begin{pmatrix} -p(0,0) & -p(0,1) & p(0,v) \\ -p(1,0) & -p(1,1) & p(1,v) \\ p(u,0) & p(u,1) & \mathbf{0} \end{pmatrix}\begin{pmatrix} H_{0,3}(v) \\ H_{3,3}(v) \\ 1 \end{pmatrix}. \tag{12.23}$$

All three conditions of Theorem 12.7.1 are satisfied by $H_{0,3}$ and $H_{3,3}$.

Definition. The parametric surface defined by the function p(u,v) in equation (12.23) is called the *bicubic Coons patch* or *surface* for the curves p(0,v), p(1,v), p(u,0), and p(u,1).

One of the nice properties of the bicubic Coons surface is that it gives us smooth global surfaces. Specifically, given a network of curves, the global interpolating surface that one would get using the basic bilinearly blended Coons parameterization for each individual patch would be only C^0 even if the curves themselves are C^1. This problem is caused by the use of linear blending functions. If we use the bicubic Coons surface, then we get a globally C^1 surface. This follows from Theorem 12.7.1 and properties of the functions $H_{0,3}$ and $H_{3,3}$.

Another interesting fact about the surface (Pp)(u,v) defined by equations (12.22) is that if

$$b_0(u) + b_1(u) = 1 \quad \text{or} \quad c_0(v) + c_1(v) = 1, \tag{12.24}$$

then it is *affinely invariant* in the sense that if the surface is transformed by an affine transformation, then the transformed version can be computed from equations (12.22) applied to the transformed boundary data. By a simple extension of Theorem 11.2.2.3 we only have to show that the coefficients of equation (12.22b) add to 1, that is,

$$b_0(u) + b_1(u) + c_0(v) + c_1(v) - b_0(u)c_0(v) - b_0(u)c_1(v) - b_1(u)c_0(v) - b_1(u)c_1(v) = 1,$$

but this equation can be rewritten in the form

$$(b_0(u) + b_1(u) - 1)(c_0(v) + c_1(v) - 1) = 0.$$

It follows that both the bilinearly blended and bicubic Coons surface are affinely invariant.

Even though the bicubic Coons patch gives us a C^1 surface if the boundary curves are C^1, we have little control over the derivatives along the boundaries. To get more flexibility assume that we are also given the partial derivatives $p_u(0,v)$, $p_u(1,v)$, $p_v(u,0)$, and $p_v(u,1)$. See Figure 12.13. Define new operators Q_1 and Q_2 by

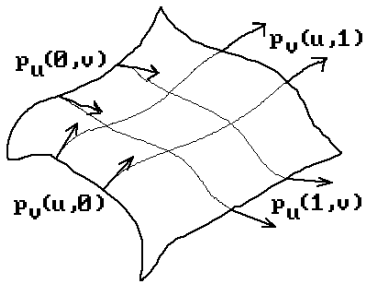


Figure 12.13. A smooth bicubic Coons patch.

$$Q_1p(u,v) = H_{0,3}(u)p(0,v) + H_{1,3}(u)p_u(0,v) + H_{2,3}(u)p_u(1,v) + H_{3,3}(u)p(1,v), \quad (12.25a)$$

$$Q_2p(u,v) = H_{0,3}(v)p(u,0) + H_{1,3}(v)p_v(u,0) + H_{2,3}(v)p_v(u,1) + H_{3,3}(v)p(u,1). \quad (12.25b)$$

Unfortunately, the Boolean sum

$$Qp = (Q_1 \oplus Q_2)p = (Q_1 + Q_2 - Q_1Q_2)p \quad (12.26)$$

refers to the values $p_{uv}(0,0)$, $p_{uv}(0,1)$, $p_{uv}(1,0)$, and $p_{uv}(1,1)$ of the mixed second partials at the corners of the patch and so we also have to assume that we have been given those values, but once we have them then the properties of the functions $H_{i,j}(u)$ imply that the parametric surface $(Qp)(u,v)$ now interpolates all the data.

Definition. The parametric surface $Qp(u,v)$ defined by equation (12.26) and its matrix form (12.29) below is called the *generalized bicubic Coons patch* or *surface*.

Before we describe the matrix form of the generalized bicubic Coons patch, let us generalize equations (12.25) to allow other blending functions $d_0(t)$, $d_1(t)$, $e_0(t)$, and $e_1(t)$. Assume that the operators Q_1 and Q_2 are defined by

$$(Q_1p)(u,v) = b_0(u)p(0,v) + b_1(u)p(1,v) + d_0(u)p_u(0,v) + d_1(u)p_u(1,v), \quad (12.27a)$$

$$(Q_2p)(u,v) = c_0(v)p(u,0) + c_1(v)p(u,1) + e_0(v)p_v(u,0) + e_1(v)p_v(u,1). \quad (12.27b)$$

The necessary constraints on the new blending functions for everything to work are

$$d_i(j) = e_i(j) = 0, \quad d_i'(j) = e_i'(j) = \delta_{ij}, \quad i,j \in \{0,1\}. \quad (12.27c)$$

(We know that the functions $H_{i,j}(t)$ satisfied those constraints.) If we define a matrix B by

$$B = \begin{pmatrix} p(0,0) & p(0,1) & p_v(0,0) & p_v(0,1) \\ p(1,0) & p(1,1) & p_v(1,0) & p_v(1,1) \\ p_u(0,0) & p_u(0,1) & p_{uv}(0,0) & p_{uv}(0,1) \\ p_u(1,0) & p_u(1,1) & p_{uv}(1,0) & p_{uv}(1,1) \end{pmatrix}, \tag{12.28}$$

then equation (12.26) can be written in the form

$$(Qp)(u,v) = \begin{pmatrix} b_0(u) & b_1(u) & d_0(u) & d_1(u) & 1 \end{pmatrix} \begin{pmatrix} & & & & p(0,v) \\ & & -B & & p(1,v) \\ & & & & p_u(0,v) \\ & & & & p_u(1,v) \\ p(u,0) & p(u,1) & p_v(u,0) & p_v(u,1) & \mathbf{0} \end{pmatrix} \begin{pmatrix} c_0(v) \\ c_1(v) \\ e_0(v) \\ e_1(v) \\ 1 \end{pmatrix}. \tag{12.29}$$

The surface Qp(u,v) defined by equation (12.29) will interpolate the boundary curves p(0,v), p(1,v), p(u,0), p(u,1), their derivatives $p_u(0,v)$, $p_u(1,v)$, $p_v(u,0)$, $p_v(u,1)$, and the values $p_{uv}(0,0)$, $p_{uv}(0,1)$, $p_{uv}(1,0)$, $p_{uv}(1,1)$. Using such parameterizations we can now define a network of patches from a given grid of boundary curves p(i,v) and p(u,j), i,j = 1,2, . . . , as shown in Figure 12.14 that is a globally C^1 surface. It is also affinely invariant provided that equations (12.24) are satisfied. Section 12.9 will have more to say about the B matrix in equation (12.28). We summarize our results about networks of Coons surfaces in Table 12.7.1.

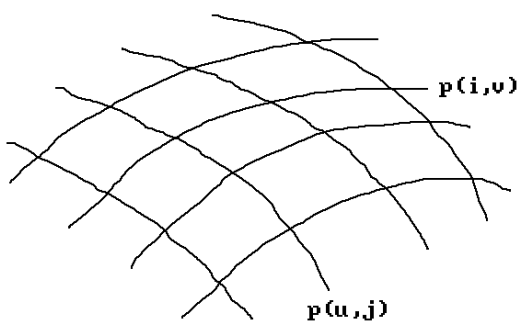


Figure 12.14. A smooth network of Coons surfaces.

Table 12.7.1 Properties of Coons networks.

Individual patch p(u,v) of Coons networks	Continuity of network	Comments
Bilinearly blended (equations (12.20))	C^0	Affinely invariant
Bicubic (equation (12.23))	C^1	Affinely invariant
Beneralized bicubic (equation (12.26))	C^1	Affinely invariant, allows control over partials along boundary

Generalized Coons surfaces p(u,v) are nice because they are flexible. They interpolate arbitrary boundary curves and partials along the boundary. They have some problems however. The first problem is that someone using them would also have to specify $p_{uv}(0,0)$, $p_{uv}(0,1)$, $p_{uv}(1,0)$, and $p_{uv}(1,1)$. This is not easy because, since p(u,v) is unknown, all one can do is estimate those unintuitive values. We describe one way that avoids having to specify the mixed partials. Let $\alpha(v) = p_u(0,v)$, $\beta(v) = p_u(1,v)$, $\gamma(u) = p_v(u,0)$, $\delta(u) = p_v(u,1)$. Replace the 2×2 submatrix of vectors $p_{uv}(0,0)$, $p_{uv}(0,1)$, $p_{uv}(1,0)$, and $p_{uv}(1,1)$ in the matrix B of equation (12.29) by the 2×2 matrix of parameterized vectors

$$\begin{matrix} \dfrac{u\dfrac{d\alpha}{dv}(0)+v\dfrac{d\gamma}{du}(0)}{u+v} & \dfrac{u\dfrac{d\alpha}{dv}(1)+(1-v)\dfrac{d\delta}{du}(0)}{u+1-v} \\ \dfrac{(1-u)\dfrac{d\beta}{dv}(0)+v\dfrac{d\gamma}{du}(1)}{1-u+v} & \dfrac{(1-u)\dfrac{d\beta}{dv}(1)+(1-v)\dfrac{d\delta}{du}(1)}{1-u+1-v} \end{matrix}. \tag{12.30}$$

Definition. The parameterized surface that we get from this new data is called the *Gregory square*.

Note that it might have been tempting to have written expressions like

$$u\frac{d\alpha}{dv}(0)$$

as $up_{vu}(0,0)$, but we did not on purpose. It would have been confusing because it would have looked as if we needed to specify the mixed second partial $p_{vu}(u,v)$ at (0,0), which is not case. We computed the value from the curve $\alpha(v)$ that we were given. Note also that the terms in matrix (12.30) are just convex combinations of the derivatives of the curves $\alpha(v)$, $\beta(v)$, $\gamma(u)$, and $\delta(u)$ and that the mixed partials that we get at the corners of the patch now depend on the direction in which we approach the corner. See [Chiy88], [HosL93], or [Fari97] for more about this surface patch.

A second problem for generalized Coons surfaces p(u,v) is that nice functions, such as C^2 functions, have the property that one can interchange the order of partial differentiation, but this may not be true for the parameterization p(u,v). Specifically, $\partial^2 p/\partial u \partial v$ may not equal $\partial^2 p/\partial v \partial u$ at the corners of the patch. Achieving equality of these two mixed partials is referred to as a *compatibility condition*. (That term is used for other conditions such as having the boundary curves meet at the corners or adjacent patches having the same tangent planes where they meet.) A consequence of unequal mixed partials is that the projection operators Q_1 and Q_2 in equation (12.26) may not commute. The Gregory square does not satisfy the compatibility condition. On the other hand, the Gregory patch discussed later in Section 12.11 does.

Finally, Coons surfaces are a special case of *Gordon surfaces*. The latter interpolate a network of m curves in the u-direction and n curves in the v-direction. The Coons surface is the case m = n = 1. It is also possible to define triangular Coons patches. See [HosL93], [Fari97], or [Salo99].

12.8 Tensor Product Surfaces

Tensor product surfaces are one of the most common surfaces encountered in CAGD. Some simple versions can be computed with matrices. This section only gives an overview, leaving the details with regard to some important special cases for subsequent sections.

Consider a curve

$$p(u) = \sum_{i=0}^{m} f_i(u) p_i,$$

where the $f_i(u)$ are basis functions, and we treat the p_i as a 1-parameter family of vector-valued functions

$$p_i(v) = \sum_{j=0}^{n} g_j(v) \mathbf{p}_{ij}$$

with respect to some other basis functions $g_j(v)$ and points $\mathbf{p}_{ij}$.

Definition. The parametric surface $p(u,v)$ defined by

$$p(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} f_i(u) g_j(v) \mathbf{p}_{ij} \tag{12.31}$$

is called a *tensor product* or *Cartesian product* surface with basis functions $f_i(u)g_j(v)$. In matrix form, equation (12.31) becomes

$$p(u,v) = (f_0(u)\, f_1(u) \cdots f_m(u)) \begin{pmatrix} \mathbf{p}_{00} & \cdot & \cdot & \mathbf{p}_{0n} \\ \cdot & \cdot & & \cdot \\ \cdot & & \cdot & \cdot \\ \mathbf{p}_{m0} & \cdot & \cdot & \mathbf{p}_{mn} \end{pmatrix} \begin{pmatrix} g_0(v) \\ g_1(v) \\ \cdot \\ g_n(v) \end{pmatrix}. \tag{12.32}$$

Note how the partial derivatives of tensor product surfaces are easily obtained from the derivatives for the curves:

$$\frac{\partial p}{\partial u}(u,v) = \sum_{j=0}^{n} g_j(v) \left[\frac{\partial}{\partial u} \sum_{i=0}^{m} f_i(u) \mathbf{p}_{ij} \right] \tag{12.33a}$$

$$\frac{\partial p}{\partial v}(u,v) = \sum_{i=0}^{m} f_i(u) \left[\frac{\partial}{\partial v} \sum_{j=0}^{n} g_j(v) \mathbf{p}_{ij} \right] \tag{12.33b}$$

The next four sections discuss some of the more common tensor product surfaces: the bicubic patch, Bézier surfaces, B-spline surfaces, and rational B-spline surfaces.

12.9 The Bicubic Patch

Definition. A parametric surface p(u,v) defined by

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} u^i v^j \mathbf{p}_{ij}, \tag{12.34}$$

where $0 \le u,v \le 1$ is called the *general bicubic tensor product surface* or *bicubic patch*. Similar to the case of curves, the points $\mathbf{p}_{i,j}$ are called the *algebraic coefficients* of the bicubic patch.

It is easy to see that the bicubic patch can be interpreted as the tensor product of two cubic curves defined by their algebraic coefficients as in equation (11.37). Just as in the case of curves, the algebraic description (12.34) is usually not very convenient for a user. One wants a more geometric way to specify the surface. One way to get this geometric description is with the following observations. There are 16 (vector) degrees of freedom. Some obvious geometric constraints are the four corner points and the eight tangent vectors in the u and v direction at those points. That leaves four degrees of freedom and we can use the mixed partials $p_{uv}(u,v)$ at the corner points. They are called the *twist vectors*.

Definition. The matrix **B** defined by

$$\mathbf{B} = \begin{pmatrix} p(0,0) & p(0,1) & p_v(0,0) & p_v(0,1) \\ p(1,0) & p(1,1) & p_v(1,0) & p_v(1,1) \\ p_u(0,0) & p_u(0,1) & p_{uv}(0,0) & p_{uv}(0,1) \\ p_u(1,0) & p_u(1,1) & p_{uv}(1,0) & p_{uv}(1,1) \end{pmatrix} \tag{12.35}$$

is called the *geometric matrix* for the bicubic patch. Its elements are called the *geometric coefficients* of the patch.

The geometric matrix determines the algebraic coefficients completely. Here is how the geometric coefficients would determine the point p(u,v) on the patch by repeatedly using the Hermite principle that the endpoints and tangents of a curve determine the curve completely: See Figure 12.15.

(1) p(u,0) is determined from p(0,0), p(1,0), $p_u(0,0)$, and $p_u(1,0)$.
(2) Similarly, p(u,1) is determined from p(0,1), p(1,1), $p_u(0,1)$, and $p_u(1,1)$.
(3) Next, $p_v(u,0)$ is determined from $p_v(0,0)$, $p_v(1,0)$, $p_{uv}(0,0)$, and $p_{uv}(1,0)$.
(4) Similarly, $p_v(u,1)$ is determined from $p_v(0,1)$, $p_v(1,1)$, $p_{uv}(0,1)$, and $p_{uv}(1,1)$.
(5) Finally, p(u,v) is determined from p(u,0), p(u,1), $p_v(u,0)$, and $p_v(u,1)$.

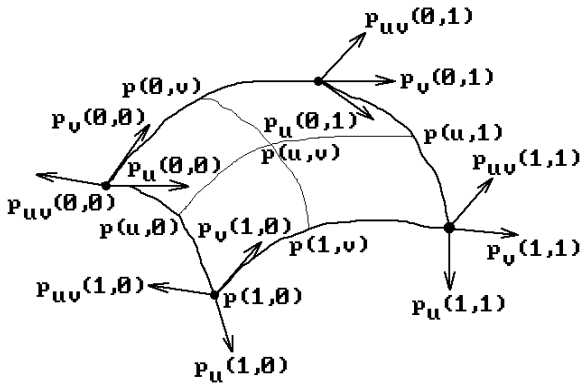


Figure 12.15. Using the geometric matrix to compute p(u,v).

To find a formula for computing a point on the bicubic patch from its geometric coefficients, one needs to take the tensor product of two cubic curves defined via their geometric coefficients as in equation (11.37). This leads to the formula

$$p(u,v) = \mathbf{U}\mathbf{M}_h\mathbf{Q}\mathbf{M}_h{}^T\mathbf{V}^T, \tag{12.36a}$$

for some matrix $\mathbf{Q} = (\mathbf{q}_{ij})$, $0 \le i,j \le 3$. By computing the points and partials of this curve at the four points (0,0), (1,0), (0,1), and (1,1) directly from its formula, it is easy to check that in fact the matrix $\mathbf{Q}$ is just the matrix $\mathbf{B}$ in equation (12.35) (Exercise 12.9.1). Therefore, the geometric form of the equation for the bicubic patch is

$$p(u,v) = \mathbf{U}\mathbf{M}_h\mathbf{B}\mathbf{M}_h{}^T\mathbf{V}^T. \tag{12.36b}$$

This equation can also be written in the form

$$p(u,v) = (F_1(u)\ F_2(u)\ F_3(u)\ F_4(u))\mathbf{B}\begin{pmatrix} F_1(v) \\ F_2(v) \\ F_3(v) \\ F_4(v) \end{pmatrix} \tag{12.37}$$

where the functions F_i are defined by equations (11.14). (See equation (11.37).)

12.9.1 Example. Consider the bicubic patch p(u,v) with geometric matrix:

$$\mathbf{B} = \begin{pmatrix} (0,-4,0) & (0,-2,2) & (0,2,2) & (0,2,2) \\ (2,0,0) & (1,0,2) & (-1,0,2) & (-1,0,2) \\ (2,0,0) & (1,0,0) & (-1,0,0) & (-1,0,0) \\ (2,8,0) & (1,4,0) & (-1,-4,0) & (-1,-4,0) \end{pmatrix}.$$

We want to show that this patch is part of a parabolic cone.

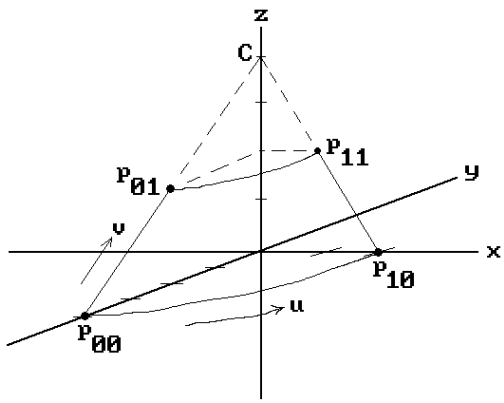


Figure 12.16. Analyzing geometric matrices for bicubic patches.

Solution. See Figure 12.16. Let

$$p(u,v) = (x(u,v), y(u,v), z(u,v)).$$

First of all, p(0,v) parameterizes the straight line segment [(0,–4,0),(0,–2,2)] because the geometric data for x(u,0) is 0 and both y(u,0) and z(u,0) have the right geometric data for a straight line. Similarly, p(1,v) parameterizes the straight line segment [(2,0,0),(1,0,2)]. The geometric data for the curves p(u,0) and p(u,1) indicates that they are parabolas. In fact, the data

$$\begin{aligned} &p(0,0) = (0,-4,0), \quad p(1,0) = (2,0,0), \quad p_u(0,0) = (2,0,0), \quad p_u(1,0) = (2,8,0) \\ &p(0,1) = (0,-2,2), \quad p(1,1) = (1,0,2), \quad p_u(0,1) = (1,0,0), \quad p_u(1,1) = (1,4,0) \end{aligned}$$

is easily solved to give

$$p(u,0) = \left(2u, 4\left(u^2-1\right), 0\right) \quad \text{and} \quad p(u,1) = \left(u, 2\left(u^2-1\right), 2\right).$$

Next, we use the data

$$\begin{aligned} &p_v(0,0) = (0,2,2), \quad p_v(1,0) = (-1,0,2), \quad p_{uv}(0,0) = (-1,0,0), \quad p_{uv}(1,0) = (-1,-4,0) \\ &p_v(0,1) = (0,2,2), \quad p_v(1,1) = (-1,0,2), \quad p_{uv}(0,1) = (-1,0,0), \quad p_{uv}(1,1) = (-1,-4,0) \end{aligned}$$

to get

$$p_v(u,0) = p_v(u,1) = \left(-u, 2\left(1-u^2\right), 2\right).$$

Note how this agrees with p(u,1) – p(u,0). The point **C** = (0,0,4) is clearly the vertex of the parabolic cone which contains this patch. We leave it as an exercise to show that

$$p(u,v) = \left(u, (2-v), 2\left(u^2-1\right)(2-v), 2v\right).$$

12.9.2 Theorem. The bicubic patch defined by equation (12.36) is affinely invariant.

Proof. By Theorem 11.2.2.3 we need to expand equation (12.37) and show that the functions that are the coefficients of $\mathbf{p}_{00}$, $\mathbf{p}_{10}$, $\mathbf{p}_{01}$, and $\mathbf{p}_{11}$ add to 1 (the other elements of the geometric matrix **B** are "vectors"). But this sum is just

$$\sum_{i,j=1}^{2} F_i(u)F_j(v) = \sum_{i=1}^{2} F_i(u)\left(\sum_{j=1}^{2} F_j(v)\right) = 1,$$

using identity (11.17).

Although the twist vectors $p_{uv}(u,v)$ of a bicubic patch do have a geometric interpretation, they are the least geometric part of the geometric matrix. For that reason, they are sometimes set to zero.

Definition. A bicubic patch for which the twist vectors vanish, that is, a patch that has a geometric matrix of the form

$$\begin{pmatrix} p(0,0) & p(0,1) & p_v(0,0) & p_v(0,1) \\ p(1,0) & p(1,1) & p_v(1,0) & p_v(1,1) \\ p_u(0,0) & p_u(0,1) & \mathbf{0} & \mathbf{0} \\ p_u(1,0) & p_u(1,1) & \mathbf{0} & \mathbf{0} \end{pmatrix},$$

is called a *Ferguson patch*.

Ferguson patches may simplify specifying the data for a patch, but they have problems. They make the patch look flat in a neighborhood of its corners. This is especially noticeable for networks of patches and therefore they are used very little. There are better ways to specify the twist vectors automatically without user intervention. Recall the Gregory square.

The *Adini twist vector* is obtained at a vertex by using information from the boundary of a patch or patches. In the single patch case we compute the bilinearly blended Coons patch from the cubic boundary curves of our patch and use its mixed partial derivative at the vertex as the twist vector. In the case of a network of patches this would not give us C^1 continuity at the vertices, but a simple modification works. We compute the bilinearly blended Coons patch for the boundary of the union of the four patches that meet at the vertex and use the mixed partial derivatives of that larger patch at the vertex. If the network of patches over domains $[u_i,u_{i+1}] \times [v_j,v_{j+1}]$ defines the global parameterization $q(u,v)$, then one can show that the Adini twist vectors are

$$\begin{aligned} q_{uv}(u_i,v_j) = {} & \frac{q_u(u_{i+1},v_j) - q_u(u_{i-1},v_j)}{u_{i+1}-u_{i-1}} + \frac{q_v(u_i,v_{j+1}) - q_v(u_i,v_{j-1})}{v_{j+1}-v_{j-1}} \\ & - \frac{q(u_{i+1},v_{j+1}) - q(u_{k+1},v_{j-1}) + q(u_{i-1},v_{j-1}) - q(u_{i-1},v_{j+1})}{(u_{i+1}-u_{i-1})(v_{j+1}-v_{j-1})}. \end{aligned} \tag{12.38}$$

Another way to specify a bicubic patch is by means of a 4×4 grid of points $\mathbf{p}_{ij}$ and requiring that the patch interpolate those points. These points provide 48 con-

straints that define the parameterization p(u,v) completely. We shall derive a formula for p(u,v) in the uniform spacing case, but rather than finding the algebraic coefficients by solving equations, we shall use a matrix approach similar to what we did in the case of the four-point problem for curves. We start with the equation (12.36) and seek a matrix $\mathbf{M}_l$ so that

$$p(u,v) = \mathbf{U}\mathbf{M}_l\mathbf{P}\mathbf{M}_l{}^T\mathbf{V}^T, \tag{12.39}$$

where $\mathbf{P} = (\ \mathbf{p}_{ij} = p(i/3, j/3)\ )$. It is easy to check that the matrix $\mathbf{M}_l$, which solves (12.39) is the same four-point matrix $\mathbf{M}_l$ that was defined in equation (11.43), that is,

$$\mathbf{M}_l = \begin{pmatrix} -\frac{9}{2} & \frac{27}{2} & -\frac{27}{2} & \frac{9}{2} \\ 9 & -\frac{45}{2} & 18 & -\frac{9}{2} \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Furthermore, the $\mathbf{B}$ and $\mathbf{P}$ matrices are related by the equation $\mathbf{B} = \mathbf{L}\mathbf{P}\mathbf{L}^T$, where the matrix $\mathbf{L}$ is as in equation (11.44).

Up to this point we have assumed that the domain for our bicubic patch is $[0,1] \times [0,1]$, but what if we were to reparameterize and use a different domain $[a,b] \times [c,d]$? The answer is similar to the answer we gave in Section 11.3 for cubic curves. The geometric matrix would have to be defined in terms of the values and derivatives of the function at the endpoints a, b, c, and d.

12.10 Bézier Surfaces

Defining tangents and especially the twist vectors for the bicubic patch is not very intuitive for many users. Again, a Bézier approach can be followed, but this time we specify a grid of sixteen points $\mathbf{p}_{ij}$. See Figure 12.17. Let

$$\mathbf{B}_b = \begin{pmatrix} \mathbf{p}_{00} & \mathbf{p}_{01} & \mathbf{p}_{02} & \mathbf{p}_{03} \\ \mathbf{p}_{10} & \mathbf{p}_{11} & \mathbf{p}_{12} & \mathbf{p}_{13} \\ \mathbf{p}_{20} & \mathbf{p}_{21} & \mathbf{p}_{22} & \mathbf{p}_{23} \\ \mathbf{p}_{30} & \mathbf{p}_{31} & \mathbf{p}_{32} & \mathbf{p}_{33} \end{pmatrix} \tag{12.40}$$

and define a parameterization p(u,v), $0 \le u,v \le 1$, by

$$p(u,v) = \mathbf{U}\mathbf{M}_b\mathbf{B}_b\mathbf{M}_b{}^T\mathbf{V}^T, \tag{12.41}$$

where $\mathbf{M}_b$ is the matrix defined in (11.47).

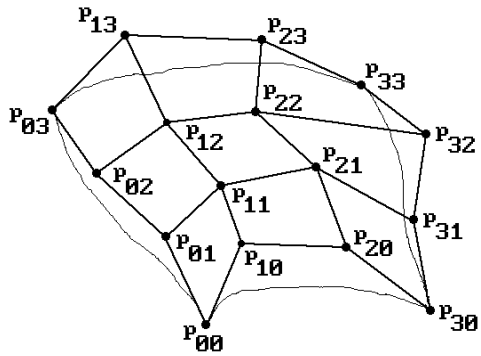


Figure 12.17. The cubic Bézier surface.

Definition. The parametric surface p(u,v) is called a *cubic Bézier surface*. The elements of the matrix $\mathbf{B}_b$, namely, the points $\mathbf{p}_{ij}$, are called the *Bézier coefficients* of this Bézier surface.

To understand this construction a little better, let

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \mathbf{B}_b \begin{pmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{pmatrix}.$$

It is easy to see that

$$p(u,v) = \mathbf{U}\mathbf{M}_h\mathbf{B}\mathbf{M}_h{}^T\mathbf{V}^T, \tag{12.42}$$

where $\mathbf{M}_h$ is the Hermite matrix defined in (11.10). In other words, $\mathbf{B}$ is the geometric matrix for the cubic Bézier surface defined by (12.40). Furthermore, a straightforward computation shows that p(u,0) is the Bézier curve based on the points $\mathbf{p}_{00}$, $\mathbf{p}_{10}$, $\mathbf{p}_{20}$, and $\mathbf{p}_{30}$. Similarly, the other boundary curves of the surface are Bézier curves on the corresponding boundary points $\mathbf{p}_{ij}$. One can also show that

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3}\binom{3}{i}u^i(1-u)^{3-i}\binom{3}{j}v^j(1-v)^{3-j}\mathbf{p}_{ij}.$$

The cubic Bézier surface can be generalized. Given points $\mathbf{p}_{ij}$, $0 \le i \le m$, $0 \le j \le n$, define a function p(u,v), $0 \le u,v \le 1$ by

$$\begin{aligned} p(u,v) &= \sum_{i=0}^{m}\sum_{j=0}^{n} B_{i,m}(u)B_{j,n}(v)\mathbf{p}_{ij} \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n}\binom{m}{i}u^i(1-u)^{m-i}\binom{n}{j}v^j(1-v)^{n-j}\mathbf{p}_{ij}. \end{aligned} \tag{12.43}$$

Definition. The parametric surface p(u,v) defined by equation (12.43) is called the *Bézier surface* defined by the *control points* $\mathbf{p}_{ij}$.

The points of a Bézier surface are efficiently computed by the two-dimensional analog of the de Casteljau algorithm (see [PieT95]). Partial derivatives are easily determined since we already know the derivatives for Bézier curves (see equation (11.57)):

$$\begin{aligned}\frac{\partial p}{\partial u}(u,v) &= \sum_{j=0}^{n} B_{j,n}(v)\left(\frac{\partial}{\partial u}\sum_{i=0}^{m} B_{i,m}(u)P_{ij}\right)\\ &= m\sum_{j=0}^{n}\sum_{i=0}^{m-1} B_{i,m-1}(u)B_{j,n}(v)(\mathbf{p}_{i+1,j}-\mathbf{p}_{ij}).\end{aligned}$$

Similarly,

$$\frac{\partial p}{\partial v}(u,v) = n\sum_{i=0}^{m}\sum_{j=0}^{n-1} B_{i,m}(u)B_{j,n-1}(v)(\mathbf{p}_{i,j+1}-\mathbf{p}_{ij}).$$

Formulas for higher derivatives are obtained just like in the case of curves.

Here are some properties of a Bézier surface, many of which are similar to those of Bézier curves:

(1) The boundary curves of a Bézier surface are Bézier curves.
(2) Only the corner vertices are interpolated but the shape of the surface closely follows the control points $\mathbf{p}_{ij}$.
(3) The vectors $\mathbf{p}_{00}\mathbf{p}_{10}$ and $\mathbf{p}_{00}\mathbf{p}_{01}$ generate the tangent plane at $\mathbf{p}_{00}$, with similar facts at the other corner points.
(4) The Bézier patch lies in the convex hull of its control points.

12.10.1 Theorem. The Bézier surface defined by (12.42) is affinely invariant.

Proof. This follows from the Theorem 11.2.2.3.

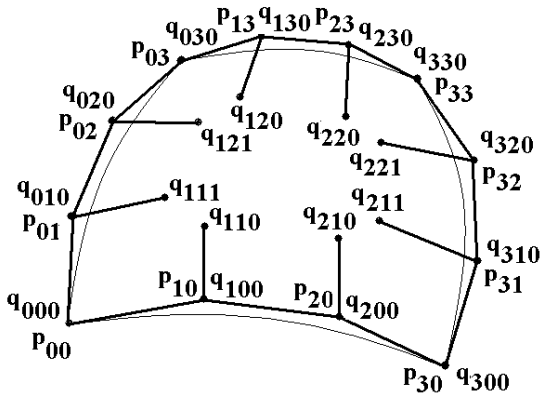
Bézier surfaces do not satisfy any known variation diminishing property (see [PraG92]).

Just like in the case of Bézier curves, one drawback with Bézier surfaces is that the degree of the surface increases as the number of control points increases. One can counter this problem by restricting oneself to cubic patches. See Section 12.15 for ways to ensure that patches meet smoothly.

12.11 Gregory Patches

Gregory patches are an extension of the Coons patch and Bézier surface. Chiyokura ([Chiy88]) has an extensive discussion of the surface and its uses. See also [HosL93] or [Salo99]. Rather than using 16 control points for a patch like an ordinary

Figure 12.18. The Gregory patch.



cubic Bézier patch, a Gregory patch uses 20 by splitting the interior four points in two.

Let $\mathbf{q}_{ijk} \in \mathbf{R}^3$, $0 \le i, j \le 3$, $k \in \{0,1\}$, and assume that $\mathbf{q}_{ij0} = \mathbf{q}_{ij1}$, $(i,j) \ne (1,1), (2,1), (1,2), (2,2)$. Define points $\mathbf{p}_{ij}(u,v)$ by

$$\mathbf{p}_{ij}(u,v) = \mathbf{q}_{ij0} = \mathbf{q}_{ij1}, \quad (i,j) \ne (1,1), (2,1), (1,2), (2,2),$$

$$\mathbf{p}_{11}(u,v) = \frac{u\mathbf{q}_{110} + v\mathbf{q}_{111}}{u+v}, \qquad \mathbf{p}_{21}(u,v) = \frac{(1-u)\mathbf{q}_{210} + v\mathbf{q}_{211}}{1-u+v},$$

$$\mathbf{p}_{12}(u,v) = \frac{u\mathbf{q}_{120} + (1-v)\mathbf{q}_{121}}{u+1-v}, \quad \mathbf{p}_{22}(u,v) = \frac{(1-u)\mathbf{q}_{220} + (1-v)\mathbf{q}_{221}}{1-u+1-v}.$$

See Figure 12.18. Note the similarity between the the last four points and those used in matrix (12.30) for the Gregory square.

Definition. The parametric surface p(u,v) defined by

$$p(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_{i,3}(u)B_{j,3}(v)\mathbf{p}_{ij}(u,v) \tag{12.44}$$

for $0 \le u,v \le 1$ is called a *Gregory patch* and the points $\mathbf{q}_{ijk}$ are called its *control points*.

The Gregory patch has a number of useful properties:

(1) If $\mathbf{q}_{ij0} = \mathbf{q}_{ij1}$, $i,j \in \{1,2\}$, then it reduces to the ordinary cubic Bézier surface as defined by equation (12.43).
(2) Using equation (12.44) makes evaluating p(u,v) and its derivatives just as easy as the corresponding task for Bézier surfaces. The only extra work is that one has to evaluate the interior points $\mathbf{p}_{ij}(u,v)$, $i,j \in \{1,2\}$, first.
(3) The surface lies in the convex hull of its twenty control points.
(4) It can be used to interpolate not only arbitrary Bézier boundary curves but arbitrary "normal" derivatives along the boundary by choosing the interior control points appropriately because

$$p_v(u,0) = 3\sum_{i=0}^{3} B_{i,3}(u)(\mathbf{q}_{i10} - \mathbf{q}_{i00}), \quad p_v(u,1) = 3\sum_{i=0}^{3} B_{i,3}(u)(\mathbf{q}_{i30} - \mathbf{q}_{i20}), \tag{12.45a}$$

$$p_u(0,v) = 3\sum_{i=0}^{3} B_{i,3}(u)(\mathbf{q}_{1i1} - \mathbf{q}_{0i0}), \quad p_u(1,v) = 3\sum_{i=0}^{3} B_{i,3}(u)(\mathbf{q}_{3i0} - \mathbf{q}_{2i1}). \tag{12.45b}$$

(5) It satisfies the compatibility condition for twist vectors and those do not have to be specified.

Because of property (4), it is easy to use Gregory patches to define composite surfaces in which all the patches meet smoothly along common edges. They are also good for blending since one can define nonrectangular Gregory patches. See [Chiy88] for how one works with these patches in practice and how the control points are used.

12.12 B-spline Surfaces

12.12.1 The Basic B-spline Surface

Let $N_{i,k}(u)$ and $N_{j,h}(v)$ be the functions defined by equations (11.69) with respect to given nondecreasing knot vectors $(u_0,u_1, \ldots, u_{m+k})$ and $(v_0,v_1, \ldots ,v_{n+h})$, respectively. Let $\mathbf{p}_{ij}$ be a given set of points. Define a function p(u,v) by

$$p(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,k}(u)N_{j,h}(v)\mathbf{p}_{ij}. \tag{12.46}$$

Definition. The parametric surface p(u,v) defined by equation (12.46) is called a *B-spline surface of order* (k,h) and *degree* (k – 1,h – 1) with *control points* $\mathbf{p}_{ij}$ and *u-knots* u_i and *v-knots* v_j. The *domain* of the surface is defined to be the rectangle $[u_{k-1},u_{m+1}] \times [v_{h-1},v_{n+1}]$. If k = h = 3, then the surface is called a *bicubic* B-spline surface.

The B-spline tensor product surface defined by equation (12.46) satisfies some important properties that follow from those of the corresponding curves. For example,

(1) (Local control) If a point $\mathbf{p}_{ij}$ is moved, then the only change to the function p(u,v) occurs in the rectangle $[u_i,u_{i+k}) \times [v_j,v_{j+h})$.
(2) At a u-knot of multiplicity r, the partial derivatives $\partial^i p/\partial u^i$ exist and are continuous for $0 \le i \le k - 1 - r$. At a v-knot of multiplicity s, the partial derivatives $\partial^j p/\partial v^j$ exist are continuous for $0 \le j \le h - 1 - s$.
(3) (Local convex hull property) The surfaces satisfy the convex hull property, that is, they lie in the convex hulls of their control points. In fact, like in the case of B-spline curves, a stronger property holds: if $(u,v) \in [u_i,u_{i+1}) \times [v_j,v_{j+1})$, then p(u,v) lies in the convex hull of the points $\mathbf{p}_{st}$, $i - k + 1 \le s \le i$, $j - h + 1 \le t \le j$.
(4) If the B-splines $N_{i,k}(u)$ and $N_{j,h}$ (v) have clamped knot vectors, then p(u,v) interpolates the four corner points, that is,

$p(u_{k-1}, v_{h-1}) = \mathbf{p}_{00}$, $p(u_{m+1}, v_{h-1}) = \mathbf{p}_{m0}$, $p(u_{k-1}, v_{n+1}) = \mathbf{p}_{0n}$, and $p(u_{m+1}, v_{n+1}) = \mathbf{p}_{mn}$.

(5) If $m = k - 1$, $n = h - 1$, $(u_i) = (0, \ldots, 0, 1, \ldots 1)$, and $(v_j) = (0, \ldots, 0, 1, \ldots, 1)$, then $p(u,v)$ defines a Bézier surface.

In addition, we have

12.12.1 Theorem. B-spline surfaces are affinely invariant.

Proof. This follows from Theorem 11.2.2.3.

B-spline surfaces $p(u,v)$ do not satisfy any known variation diminishing property (see [PraG92]). Section 12.12.5 will discuss algorithms for evaluating $p(u,v)$ and its derivatives.

12.12.2 Polynomial Surfaces and Multiaffine Maps

The multiaffine approach to polynomial curves described in Section 11.5.2 can be extended to polynomial surfaces. What we need to do is extend the notion of polar form or blossom for a polynomial function of one parameter to a polynomial function

$$p : \mathbf{R}^2 \to \mathbf{R}^m \tag{12.47}$$

of two parameters. There are two ways that one can define a blossom P for such p. One leads to tensor product surfaces, the other, to surfaces based on triangular patches. We only sketch the approaches. For much more detail see [Gall00] or [Fari97]. A reader who understands the material in Section 11.5.2 should find what we do here straightforward.

The Tensor Product Surface Blossom. Here we treat each variable u and v for a point (u,v) in $\mathbf{R}^2$ separately and construct a blossom for each in the same manner as in Section 11.5.2, that is, if the degree of p in u and v is d1 and d2, respectively, then the blossom has the form

$$P : \mathbf{R}^{d1} \times R^{d2} \to \mathbf{R}^m. \tag{12.48}$$

The function $P(u_1, u_2, \ldots, u_{d1}, v_1, v_2, \ldots, v_{d2})$, called the *tensor product polar form* or *tensor product blossom* for $p(u,v)$, is only symmetric and multiaffine in the variables u_i and v_j separately. It follows that keeping one set of variables fixed and thinking of P as a function of the other set means that all the algorithms and properties of the blossoms in Section 11.5.2 are valid here. The efficient evaluation of tensor product Bézier and B-spline surfaces is based on this notion of blossom.

12.12.2.1 Example. To find the tensor product blossom P of $p(u,v) = (u + v - 3, uv, u^2 + v^2)$.

Solution. For fixed v, the blossom with respect to u is

$$((1/2)(u_1+u_2)+v-3,(1/2)v(u_1+u_2),u_1u_2+v^2).$$

We now think of u_1 and u_2 as fixed and find the blossoms of the polynomials in v to get

$$P(u_1,u_2,v_1,v_2)=((1/2)(u_1+u_2)+(1/2)(v_1+v_2)-3,(1/4)(u_1+u_2)(v_1+v_2),u_1u_2+v_1v_2).$$

We would have gotten the same function P if we had first found the blossom for v and then u.

The Triangular Surface Blossom. Here we look for a more direct generalization of a blossom as defined in Section 11.5.2 and treat a point (u,v) in $\mathbf{R}^2$ as a single entity. In other words, the blossom of the polynomial function p in equation (12.47) should be a symmetric multilinear function of the form

$$P:(\mathbf{R}^2)^d \to \mathbf{R}^m, \tag{12.49}$$

where d is some appropriate "degree" of p, and it should satisfy P((u,v), . . . ,(u,v)) = p(u,v).

12.12.2.2 Theorem. If d is the total degree of p, then a unique such function P exists and is called the *triangular polar form* or *triangular blossom* of p. If p(u,v) is a monomial cu^hv^k, h + k = m ≤ d, then

$$P((u_1,v_1),\ldots,(u_d,v_d))=c\frac{h!k!(d-(h+k))!}{d!}\sum_{I\cup J\subseteq\{1,\ldots,d\},I\cap J=\phi,|I|=h,|J|=k}\left(\prod_{i\in I}u_i\right)\left(\prod_{j\in J}v_j\right). \tag{12.50}$$

The blossom for an arbitrary polynomial p(u,v) is obtained by adding up the blossoms of all the monomial terms in p(u,v) using equation (12.50).

Proof. See [Gall00].

12.12.2.3 Example. To find the triangular blossom P of $p(u,v)=(u+v-3,uv,u^2+v^2)$.

Solution. The polynomial has total degree two and so using equation (12.50) on all the monomials we get

$$P((u_1,v_1),(u_2,v_2))=((1/2)(u_1+u_2)+(1/2)(v_1+v_2)-3,(1/2)(u_1v_2+u_2v_1),u_1u_2+v_1v_2).$$

The justification for the adjective "triangular" in the name of the blossom P is that this approach leads to surfaces defined on triangular patches because one wants to use barycentric coordinates to describe points in the plane $\mathbf{R}^2$ (as we did for points in $\mathbf{R}$) and for that one needs a triangle. The basic domain for p, which was the unit interval [0,1] for curves, is now the triangle with vertices (0,0), (1,0), and (0,1).

With this true generalization of a blossom, everything we did in Section 11.5.2 carries over to here (as long as we translate results correctly). We need to remember, however, that barycentric coordinates are now triples of real numbers, not pairs, so that, as a consequence, arrays of control points are replaced by triangular arrays of control points as we shall see shortly. To simplify the notation, define the triangular set of integer triples I_d by

$$I_d = \{(i,j,k) \in Z^3 \mid 0 \le i,j,k \text{ and } i+j+k = d\}.$$

Definition. If $(i,j,k) \in I_d$, then the *Bernstein polynomial* $B^d_{i,j,k}(u,v,w)$, $u + v + w = 1$, is defined by

$$B^d_{i,j,k}(u,v,w) = \frac{d!}{i!\,j!\,k!} u^i v^j w^k.$$

Like their cousins in Section 11.4, these Bernstein polynomials also satisfy a recurrence relation (see Exercise 11.4.1).

12.12.2.4 Theorem. The Bernstein polynomials satisfy the recurrence relation

$$B^d_{i,j,k}(u,v,w) = uB^d_{i-1,j,k}(u,v,w) + vB^d_{i,j-1,k}(u,v,w) + wB^d_{i,j,k-1}(u,v,w), \ (i,j,k) \in I_d.$$

Proof. This follows easily from the binomial coefficient identity

$$\frac{d!}{i!\,j!\,k!} = \frac{(d-1)!}{(i-1)!\,j!\,k!} + \frac{(d-1)!}{i!\,(j-1)!\,k!} + \frac{(d-1)!}{i!\,j!\,(k-1)!}.$$

The next theorem states the main facts about triangular blossoms.

12.12.2.5 Theorem. Let $\mathbf{r}$, $\mathbf{s}$, and $\mathbf{t}$ be three linearly independent points in $\mathbf{R}^3$. Let

$$\{\mathbf{b}_{i,j,k} \mid (i,j,k) \in I_d\}$$

be any set of $(d + 1)(d + 2)/2$ points in $\mathbf{R}^3$ for some integer $d \ge 1$.

(1) There is a unique polynomial surface $p(u,v)$ of total degree d whose triangular blossom P satisfies

$$P(\underbrace{\mathbf{r},\ldots,\mathbf{r}}_{i},\underbrace{\mathbf{s},\ldots,\mathbf{s}}_{j},\underbrace{\mathbf{t},\ldots,\mathbf{t}}_{k}) = \mathbf{b}_{i,j,k}$$

for all $(i,j,k) \in I_d$.

(2) If $(u,v) = a\mathbf{r} + b\mathbf{s} + c\mathbf{t}$, $a + b + c = 1$, then

$$p(u,v) = P((u,v),\ldots,(u,v)) = \sum_{(i,j,k)\in I_d} B^d_{i,j,k}(a,b,c)\mathbf{b}_{i,j,k}. \tag{12.51}$$

Furthermore, if we define points $\mathbf{b}^m_{i,j,k}$, $1 \le m \le d$, by

$$\mathbf{b}^0_{i,j,k} = \mathbf{b}_{i,j,k}.$$
$$\mathbf{b}^m_{i,j,k} = a\mathbf{b}^{m-1}_{i+1,j,k} + b\mathbf{b}^{m-1}_{i,j+1,k} + c\mathbf{b}^{m-1}_{i,j,k+1},\ \ i+j+k = d-m,$$

then

$$\mathbf{b}^m_{i,j,k} = P(\underbrace{(u,v),\ldots,(u,v)}_{m},\underbrace{\mathbf{r},\ldots,\mathbf{r}}_{i},\underbrace{\mathbf{s},\ldots,\mathbf{s}}_{j},\underbrace{\mathbf{t},\ldots,\mathbf{t}}_{k}),$$

and $p(u,v) = \mathbf{b}^d_{000}$.

Proof. See [Gall00]. The second part of (2) is proved by a simple induction and leads to Algorithm 12.12.2.1 for computing p(u,v) called the *de Casteljau algorithm*.

Definition. Using the notation in Theorem 12.12.2.5 a set of points like the set of $\mathbf{b}_{i,j,k}$ is called a *triangular* or *Bézier control net* and the parametric surface p(u,v) is called a *triangular Bézier surface* of degree d whenever $\mathbf{r} = (1,0)$, $\mathbf{s} = (0,1)$, and $\mathbf{t} = (0,0)$.

The definitions and facts about triangular blossoms, especially the notation, may look formidable in the abstract but, like in the case of blossoms for curves, "the beast has a bark that is louder than its bite." The examples and applications in the next section will show that actually working with triangular blossoms is not that bad because it involves fairly simple triangle geometry.

Inputs: A triangular array of points $\mathbf{b}_{i,j,k}$, $0 \le i, j, k, i+j+k = d$
A tuple (a,b,c) with $a + b + c = 1$
Output: $\mathbf{b}^d_{0,0,0}$

Step 1: Set $\mathbf{b}^0_{i,j,k} = \mathbf{b}_{i,j,k}$.
Step 2: For $m = 1, 2,\ldots,d$ and $i + j + k = d - m$

$$\mathbf{b}^m_{i,j,k} = a\mathbf{b}^{m-1}_{i+1,j,k} + b\mathbf{b}^{m-1}_{i,j+1,k} + c\mathbf{b}^{m-1}_{i,j,k+1}$$

Algorithm 12.12.2.1. The de Casteljau algorithm for triangular patches.

12.12.3 Triangular Bézier Surfaces

In the last section we laid the groundwork for triangular blossoms. This section will show how one works with them. Using them for surface parameterizations means that such parameterizations are based on triangular domains. Triangles are in a sense more natural for surfaces because a triangular grid of points on a surface gives a truly linear approximation of a surface, whereas a rectangular grid may very well have rectangles whose vertices do not lie in a plane so that they would not be planar.

First of all, a triangular control net $\{\mathbf{b}_{i,j,k}\}$ is usually displayed in a triangular form (even though the points themselves can be arbitrary points of $\mathbf{R}^3$). For example, see the right side of Figure 12.19 for the case when d = 3. (That representation is not universal however because some authors show a rotated version of this triangle.) If we let $\mathbf{r} = (1,0)$, $\mathbf{s} = (0,1)$, and $\mathbf{t} = (0,0)$ in Theorem 12.12.2.5, then according to that theorem the points $\mathbf{b}_{i,j,k}$ define a surface p(u,v). Again looking at Figure 12.19 note that when we represent all points in the triangle **rst** by means of barycentric coordinates there is a natural correspondence between the point $\mathbf{b}_{i,j,k}$ and the point in the triangle **rst** with barycentric coordinates (i/d,j/d,k/d). Note also how the ith to the right slanting column of the $\mathbf{b}_{i,j,k}$ array all have the same index i. Such a diagram generalizes in the obvious way to triangular $\mathbf{b}_{i,j,k}$ arrays for other values of d. With this representation it is now easy to explain how the de Casteljau algorithm works in this situation. Consider the case d = 3 again. See Figure 12.20. To simplify the notation we have dropped the superscripts, so that the points $\mathbf{b}^m_{i,j,k}$ can be recognized by the fact that their indices add up to d − m. The start points $\mathbf{b}^0_{i,j,k} = \mathbf{b}_{i,j,k}$ are the points marked with solid circles along the outside of the region along with the one center point $\mathbf{b}_{1,1,1}$. The points $\mathbf{b}^1_{i,j,k}$ are the points marked as circles. For example,

$$\mathbf{b}^1_{0,0,2} = a\mathbf{b}_{1,0,2} + b\mathbf{b}_{0,1,2} + c\mathbf{b}_{0,0,3}.$$

The points $\mathbf{b}^2_{i,j,k}$ are the points marked with a cross. For example,

$$\mathbf{b}^2_{0,0,1} = a\mathbf{b}_{1,0,1} + b\mathbf{b}_{0,1,1} + c\mathbf{b}_{0,0,2}.$$

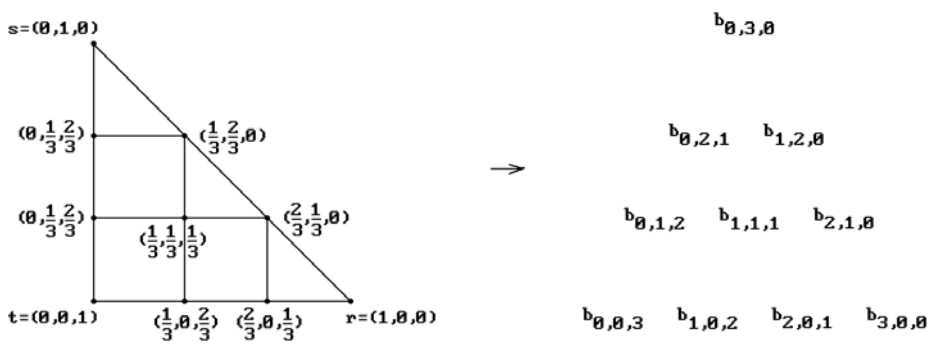


Figure 12.19. The standard triangular net versus an arbitrary one for d = 3.

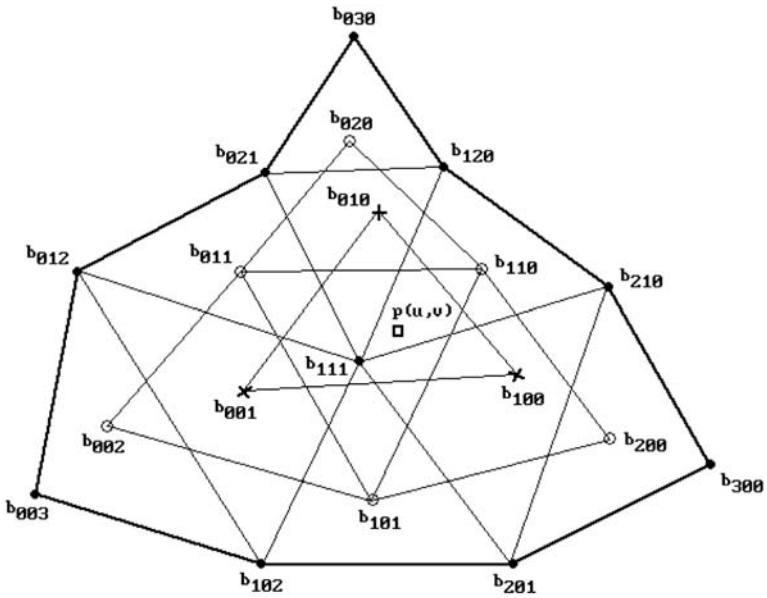


Figure 12.20. Stages of the de Casteljau algorithm.

Finally, the point

$$p(u,v) = \mathbf{b}^3_{0,0,0} = a\mathbf{b}_{1,0,0} + b\mathbf{b}_{0,1,0} + c\mathbf{b}_{0,0,1}.$$

is marked with a square.

Let us work through a concrete example.

12.12.3.1 Example. Let a triangular Bézier surface p(u,v) of degree two be defined by the triangular net

$$\begin{aligned}
&\mathbf{b}_{0,2,0} = (2,5,0)\\
&\mathbf{b}_{0,1,1} = (2,4,1) \quad \mathbf{b}_{1,1,0} = (3,4,3)\\
&\mathbf{b}_{0,0,2} = (2,3,0) \quad \mathbf{b}_{1,0,0} = (3,3,1) \quad \mathbf{b}_{2,0,0} = (4,3,0)
\end{aligned}$$

The problem is to compute p(1/6,1/3).

Solution. The barycentric coordinates of (1/6,1/3) with respect to the vertices (1,0), (0,1), and (0,0) of our triangular domain are (1/6,1/3,1/2). Therefore,

$$\begin{aligned}
\mathbf{b}^1_{0,0,1} &= (1/6)\mathbf{b}_{1,0,1} + (1/3)\mathbf{b}_{0,1,1} + (1/2)\mathbf{b}_{0,0,2}\\
&= (1/6)(3,3,1) + (1/3)(2,4,1) + (1/2)(2,3,0) = (13/6, 10/3, 1/2)\\
\mathbf{b}^1_{1,0,0} &= (1/6)\mathbf{b}_{2,0,0} + (1/3)\mathbf{b}_{1,1,0} + (1/2)\mathbf{b}_{1,0,1}\\
&= (1/6)(4,3,0) + (1/3)(3,4,3) + (1/2)(3,3,1) = (19/6, 10/3, 3/2)
\end{aligned}$$

$$\begin{aligned}
\mathbf{b}^1_{0,1,0} &= (1/6)\mathbf{b}_{1,1,0} + (1/3)\mathbf{b}_{0,2,0} + (1/2)\mathbf{b}_{0,1,1} \\
&= (1/6)(3,4,3) + (1/3)(2,5,0) + (1/2)(2,4,1) = (13/6, 13/3, 1) \\
\mathbf{b}^2_{0,0,0} &= (1/6)\mathbf{b}_{1,0,0} + (1/3)\mathbf{b}_{0,1,0} + (1/2)\mathbf{b}_{0,0,1} \\
&= (1/6)(19/6, 10/3, 3/2) + (1/3)(13/6, 13/3, 1) + (1/2)(13/6, 10/3, 1/2) = (7/3, 11/3, 5/6),
\end{aligned}$$

and $p(1/6,1/2) = \mathbf{b}^2_{0,0,0}$.

Next, let **S** be a triangular Bézier surface with parameterization p(u,v) of degree d and domain **D**. Restricting p(u,v) to the boundary of the triangle **D** maps those points to the boundary of the surface **S**. In terms of barycentric coordinates (a,b,c) the boundary of **D** corresponds to the points with a = 0, b = 0, or c = 0. From this it is easy to see that

(1) The surface interpolates the three corner points $\mathbf{b}_{d,0,0}$, $\mathbf{b}_{0,d,0}$, and $\mathbf{b}_{0,0,d}$ of its control net.
(2) Restricting p(u,v) to an edge of the boundary of **D** is just a Bézier curve with the points of the triangular net along that edge as its control points.
(3) The tangent plane to **S** at the three corner points of its control net is defined by the three corner control points. For example, the tangent plane at $\mathbf{b}_{0,d,0}$ is defined by $\mathbf{b}_{0,d,0}$, $\mathbf{b}_{1,d-1,0}$, and $\mathbf{b}_{0,d-1,1}$.

The partial derivatives of a triangular Bézier surface are easily computed by a de Casteljau–type algorithm. One uses the Bernstein representation, equation (12.51), for the parameterization and an argument similar to the one for computing derivatives of Bézier curves.

12.12.3.2 Theorem. Let p(u,v) define a triangular Bézier surface of degree d and control net $\{\mathbf{b}_{i,j,k}\}$. Then

$$\begin{aligned}
\frac{\partial p}{\partial u}(u,v) &= d \sum_{(i,j,k)\in I_{d-1}} B^{d-1}_{i,j,k}(u,v,1-u-v)(\mathbf{b}_{i+1,j,k} - \mathbf{b}_{i,j,k+1}) \\
\frac{\partial p}{\partial v}(u,v) &= d \sum_{(i,j,k)\in I_{d-1}} B^{d-1}_{i,j,k}(u,v,1-u-v)(\mathbf{b}_{i,j+1,k} - \mathbf{b}_{i,j,k+1}) \\
\frac{\partial^2 p}{\partial u^2}(u,v) &= d(d-1) \sum_{(i,j,k)\in I_{d-2}} B^{d-2}_{i,j,k}(u,v,1-u-v)(\mathbf{b}_{i+2,j,k} - 2\mathbf{b}_{i+1,j,k+1} + \mathbf{b}_{i,j,k+2}) \\
\frac{\partial^2 p}{\partial v^2}(u,v) &= d(d-1) \sum_{(i,j,k)\in I_{d-2}} B^{d-2}_{i,j,k}(u,v,1-u-v)(\mathbf{b}_{i,j+2,k} - 2\mathbf{b}_{i+1,j+1,k} + \mathbf{b}_{i+2,j,k}) \\
\frac{\partial^2 p}{\partial u \partial v}(u,v) &= d(d-1) \sum_{(i,j,k)\in I_{d-2}} B^{d-2}_{i,j,k}(u,v,1-u-v)(\mathbf{b}_{i+1,j+1,k} - \mathbf{b}_{i+1,j,k+1} - \mathbf{b}_{i,j+1,k+1} + \mathbf{b}_{i,j,k+2}).
\end{aligned}$$

Proof. The formulas follow from equation (12.51).

12.12.4 Rational B-spline Surfaces

Rational B-spline surfaces are the surface analogs of rational B-spline curves and their use is motivated by similar reasons, namely, that, although B-spline surfaces are very general, they do not include the quadric surfaces and can only approximate them. By using rational B-spline curves and surfaces a modeling system needs to support only one uniform representation for its geometric objects. Some general references for rational Bézier and B-spline surfaces are [PieT95], [Pieg91], [Fari95], [Roge01], or [RogA90].

Proceeding just like we did for curves in Section 11.5.3, consider the tensor product surfaces defined by equation (12.31). These were surfaces in affine space. Using homogeneous coordinates, their projective space analogs would have the form

$$P(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)\mathbf{P}_{ij}, \tag{12.52}$$

where the $\mathbf{P}_{ij}$ are points described with homogeneous coordinates. Expressing the points $\mathbf{P}_{ij}$ in the form $(w_{ij}x_{ij}, w_{ij}y_{ij}, w_{ij}z_{ij}, w_{ij})$, the projective space surface defined by P(u,v) will project to the surface

$$p(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)w_{ij}\mathbf{p}_{ij}}{\sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)w_{ij}}. \tag{12.53}$$

where $\mathbf{p}_{ij} = (x_{ij}, y_{ij}, z_{ij})$.

Definition. The parametric surface p(u,v) defined by equation (12.53) is called a *rational tensor product surface*. It is called a *rational Bézier surface* if its domain is [0,1] × [0,1] and $a_i(u) = B_{i,m}(u)$ and $b_j(v) = B_{j,n}(v)$. (The $B_{s,t}(u)$ are the functions defined by equation (11.50).) The surface p(u,v) is called a *rational B-spline surface* of order (k,h) and degree (k − 1,h − 1) if the $a_i(u)$ and $b_j(v)$ are B-splines of order k and h, respectively. The knots of $a_i(u)$ are called the *u-knots* of the surface and those of $b_j(v)$, the *v-knots*. In both the Bézier and B-spline case, the points $\mathbf{p}_{ij}$ are called the *control points* of the surface and the numbers w_{ij} are called its *weights*.

Definition. The functions

$$R_{ij}(u,v) = \frac{a_i(u)b_j(v)w_{ij}}{\sum_{s=0}^{m}\sum_{t=0}^{n} a_s(u)b_t(v)w_{st}}. \tag{12.54}$$

are called *rational basis functions* for the surface defined by equation (12.53).

Using the rational basis functions, equation (12.53) can be rewritten as

$$p(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} R_{ij}(u,v)\mathbf{p}_{ij}. \tag{12.55}$$

The most commonly used type of rational B-spline surface is the following:

Definition. If the B-splines $N_{i,k}(u)$ and $N_{j,h}(v)$ (defined by equations (11.69)) are defined with respect to knot vectors

$$U = (\underbrace{a,\ldots,a}_{k},u_k,u_{k+1},\ldots,u_m,\underbrace{b,\ldots,b}_{k}) \quad \text{and} \quad V = (\underbrace{c,\ldots,c}_{h},v_h,v_{h+1},\ldots,v_n,\underbrace{d,\ldots,d}_{h}), \tag{12.56a}$$

respectively, then the rational B-spline surface

$$p(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,k}(u)N_{j,h}(v)w_{ij}\mathbf{p}_{ij}}{\sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,k}(u)N_{j,h}(v)w_{ij}}. \tag{12.56b}$$

has domain $[a,b] \times [c,d]$ and is called a ***n**onuniform **r**ational **B-s**pline (NURBS) surface*.

Note that if the weights of a NURBS surface are all 1, then formula (12.56b) reduces to (12.46) and we just have an ordinary B-spline surface. This follows from Theorem 11.5.1.5. For NURBS surfaces one usually has k = h and the domain is assumed to be $[0,1] \times [0,1]$. Algorithms for evaluating NURBS surfaces will be discussed in the next section.

Compare the properties of NURBS surfaces listed in the next theorem to the corresponding ones in Theorem 11.5.3.2 for NURBS curves.

12.12.4.1 Theorem. Let p(u,v) be a NURBS surface of order (k,h) with domain $[0,1] \times [0,1]$, u-knots u_i, v-knots v_j, control points $\mathbf{p}_{ij}$, and weights w_{ij}.

(1) The rational basis functions $R_{ij}(u)$ for p(u,v) satisfy

$$\sum_{i=0}^{m}\sum_{j=0}^{n} R_{ij}(u,v) = 1$$

and $R_{ij}(u,v) \geq 0$ if all the weights are nonnegative.

(2) The surface p(u,v) interpolates the four corner points. More precisely, $p(0,0) = \mathbf{p}_{00}$, $p(1,0) = \mathbf{p}_{m,0}$, $p(0,1) = \mathbf{p}_{0n}$, and $p(1,1) = \mathbf{p}_{mn}$.

(3) (Local control) Changing the control point $\mathbf{p}_{ij}$ or weight w_{ij} only changes the formula for p(u,v) over the rectangle $(u_i,u_{i+k}) \times (v_j,v_{j+h})$.

(4) (Projective invariance) If the surface p(u,v) is transformed by a projective transformation, the formula for the new surface is gotten simply by transforming the homogeneous control points in equation (12.52) and then reprojecting to a formula in the form of equation (12.56b).

(5) (Local convex hull property) If $w_{ij} > 0$ for all i and j, then the surface p(u,v) satisfies a strengthened convex hull property. For each i and j, $p((u_i,u_{i+1}) \times (v_j,v_{j+1}))$ is contained in the convex hull of the control points $\mathbf{p}_{st}$, where $i - k + 1 \le s \le i$ and $j - h + 1 \le t \le j$.

Proof. See [PieT95].

It is not known whether NURBS surfaces satisfy any variation diminishing property.

12.12.4.2 Theorem. Every quadric surface can be expressed as a NURBS surface.

Proof. See [Fari97]. One can show that the only quadric surface that can be parameterized by polynomials are the elliptic or hyperbolic paraboloids and parabolic cylinders. Furthermore, every quadric is a central projection of one of these in $\mathbf{R}^4$.

Because one can think of a NURBS surface as a B-spline surface in $\mathbf{R}^4$ that is projected back to $\mathbf{R}^3$, all of the basic spline algorithms, such as knot insertion, etc., carry over easily to these surfaces.

Finally, we should note that just like in the case of curves, there are some differences between rational and ordinary spline surfaces. One can show that not every triangular patch on a quadric can be represented by a rational triangular quadratic patch. For this and characterizations of when a triangular patch lies on a paraboloid see [Fari89] or [Fari97].

12.12.5 Efficient B-spline and NURBS Surface Algorithms

B-spline and NURBS surfaces have become very popular like their curve cousins, so that it is good that efficient algorithms exist for evaluating them and their derivatives. A good reference for a great many detailed algorithms is [PieT95]. We shall only describe what one has to do to find points and first partials for the surfaces, which is all that is needed for rendering.

We begin with B-spline surfaces. The basic steps to evaluate a point on the B-spline surface p(u,v) are the following:

(1) Use Algorithm 11.5.4.1 to find the u-knot span that contains u, that is, find i so that $u \in [u_i,u_{i+1})$ or $u \in (u_i,u_{i+1}]$.
(2) Use Algorithm 11.5.4.1 to find the v-knot span that contains v, that is, find j so that $v \in [v_j,v_{j+1})$ or $v \in (v_j,v_{j+1}]$.
(3) Compute that part of the sum in equation (12.46) that involves the coefficients $N_{i-k+1,k}(u), N_{i-k+2,k}(u), \ldots, N_{i,k}(u)$ and $N_{j-h+1,h}(v), N_{j-h+2,h}(v), \ldots, N_{i,h}(v)$. These are the only coefficients that do not vanish at the given u and v, so that

$$p(u,v) = \sum_{s=i-k+1}^{i} N_{s,k}(u)\mathbf{p}_s(v) \tag{12.57a}$$

where

$$\mathbf{p}_s(v) = \sum_{t=j-h+1}^{j} N_{t,h}(v)\mathbf{p}_{st}. \tag{12.57b}$$

To carry out step (3) we use Algorithm 11.5.4.2 to compute each of the points $\mathbf{p}_s(v)$ in equation (12.57b) and then use it one more time to compute p(u,v) in equation (12.57a), which, for fixed v, is just another B-spline curve.

To compute the partial derivatives of a B-spline surface p(u,v), consider the case of $\partial p/\partial u$. Since

$$\frac{\partial p}{\partial u}(u,v) = \sum_{s=i-k+1}^{j} N_{s,k}'(u)\mathbf{p}_s(v), \tag{12.58}$$

computing the partial $\partial p/\partial u$ is no different than computing the ordinary derivative of the B-spline curve q(u) = p(u,v). This can again be done with Algorithm 11.5.4.2. The case of $\partial p/\partial v$ can be handled in a similar manner if we think of p(u,v) as a B-spline curve in v and interchange the summation in equation (12.46). The best way to deal with higher order of partial derivatives of p(u,v) is to have a separate algorithm (one could use a modified version of Algorithm 11.5.4.2) that computes all the needed derivatives of just the B-spline basis functions $N_{s,k}(u)$ and $N_{t,h}(v)$, puts these values into an array, and then computes the appropriate linear combination of these values as specified by the mathematical formula for the partial derivative.

A NURBS surface p(u,v) can be evaluated using the approach that was used to evaluate NURBS curves. If $\mathbf{p}_{ij} = (x_{ij},y_{ij},z_{ij})$, then $\mathbf{p}_{ij}$ can be represented in homogeneous coordinates by the point $\mathbf{P}_{ij} = (w_{ij}x_{ij},w_{ij}y_{ij},w_{ij}z_{ij},w_{ij})$. Let

$$P(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,k}(u)N_{j,h}(v)\mathbf{P}_{ij}. \tag{12.59}$$

If $P(u,v) = (P_1(u,v), P_2(u,v), P_3(u,v), P_4(u,v))$, then

$$p(u,v) = \left(\frac{P_1(u,v)}{P_4(u,v)}, \frac{P_2(u,v)}{P_4(u,v)}, \frac{P_3(u,v)}{P_4(u,v)}\right).$$

Therefore, compute P(u,v) thought of as a B-spline surface in $\mathbf{R}^4$ and divide the first three coordinates by $P_4(u,v)$ at the end to get p(u,v).

Partial derivatives of a NURBS surface are computed by formally differentiating the right-hand expression in equation (12.53) similar to what we did for NURBS curves in Section 11.5.4. The result then again consists of pieces that can be thought of as ordinary B-spline surfaces or their derivatives, so that we can apply the methods of evaluating those surfaces. For higher derivatives it is best to derive a recursive

formula like equation (11.107) for the NURBS curve derivatives. The reader is again referred to [PieT95].

Finally, for the situation where one only needs the value and first partial derivatives of a NURBS surface (which is all that is needed for rendering), Luken and Cheng ([LukC96]) analyze the complexity of three ways of computing those values: a two stage Cox-de Boor method, a knot insertion method based on Lee's approach that we used in Section 11.5.4, and a forward differencing method. Their conclusion is that the first of these methods is the best. Although forward differencing actually was the fastest method it has numerical stability problems.

12.12.6 B-spline Interpolation

Algorithms for finding surfaces that interpolate some given data depend on the structure of the data. If the data consists of a rectangular array of points, the algorithm for finding an interpolating B-spline surface is based on the corresponding algorithm for B-spline curves. We shall outline the steps for one form of bicubic B-spline surface interpolation.

The Bicubic B-spline Interpolation Problem. Given parameter values u_i, $i = 0, \ldots, m$, with $u_0 < u_1 < \ldots < u_m$, and v_j, $j = 0, \ldots, n$, with $v_0 < v_1 < \ldots < v_n$, and given points $\mathbf{p}_{ij}$, find a bicubic B-spline surface $p(u,v)$ with the u_i and v_j as the n-knots and v-knots, respectively, and control points $\mathbf{q}_{st}$, $s = -1, \ldots, m + 1$, $t = -1, \ldots, n + 1$, so that $p(u_i,v_j) = \mathbf{p}_{ij}$.

To motivate the solution to the interpolation problem note that an interpolating B-spline surface of the form shown in equation (12.46) would satisfy

$$\mathbf{p}_{ij} = p(u_i, v_j) = \sum_{s=0}^{m}\sum_{t=0}^{n} N_{s,k}(u_i)N_{t,h}(v_j)\mathbf{q}_{st} = \sum_{s=0}^{m} N_{s,k}(u_i)\mathbf{r}_{sj}, \tag{12.60a}$$

where

$$\mathbf{r}_{sj} = \sum_{t=0}^{n} N_{t,h}(v_j)\mathbf{q}_{st}. \tag{12.60b}$$

Equation (12.60a) shows that for fixed j the $\mathbf{r}_{sj}$ are the control points of a spline that interpolates the column of points $\mathbf{p}_{ij}$, $i = 0, \ldots, m$, and equation (12.60b) shows that for fixed i the $\mathbf{q}_{st}$ are the control points of a spline that interpolates the row of points $\mathbf{r}_{sj}$, $j = 0, \ldots, n$. This means that our interpolation problem can be solved in two stages using results from Section 11.5.5.

(1) First, solve the curve interpolation problem for each column of the array of points $\mathbf{p}_{ij}$. This will give us cubic B-spline curves $p_j(v)$ with knots v_j and control points $\mathbf{r}_{sj}$, $i = -1, \ldots m + 1$. Note that the solutions are based on the **same** tridiagonal matrix shown in equation (11.111), which means that the LU-decomposition of that matrix which is used to solve that system of equations is done only once.

(2) Next, do a curve interpolation on the rows of the array of points $\mathbf{r}_{sj}$ to get the desired control points $\mathbf{q}_{st}$.

Of course, like in the curve case, one is not given the knots in practice. Unfortunately, things get more complicated here because we have to produce **one** set of knots u_i for **all** of the curves $p(u,v_j)$ with fixed v_j, $j = 0, \ldots n$, and similarly for the v_j. One typically uses some sort of averaging process, but that may not work very well if our data is not well spaced. See [PieT95] or [Fari97] for a much more thorough discussion. [PieT95] also discusses of interpolation of curve networks.

Data sets are not always rectangular. For example, one might have gotten rows of unequal number of data points from a sampling of slices of an object. One approach for this case is to use spline curves that interpolate the rows and then use a skinning surface (see Section 14.7) for these curves for our interpolating surface. One potential problem is that the number of column control points might get very large. An approach that alleviates this problem can be found in [PieT00].

12.13 Cyclide Surfaces

This section discusses one final class of surfaces that can be defined by equations, namely, the cyclides. See Figure 12.21. Interest in cyclides has waxed and waned over time. In 1982 R.R. Martin ([Mart82]) showed that they were useful in CAGD and since then interest in these surfaces has revived. They have proved especially useful for certain blending operations. We shall describe a few such applications in Section 15.6. In this section we shall discuss a few of their properties relevant to CAGD. We can only present a brief overview. More details can be found in [ChDH89], [Prat90], and [Boeh90]. Another good reference is [KraM00]. Throughout this section, the term "cyclide" will mean "Dupin cyclide." Only at the end will we make a few comments about a more general related class of surfaces also called cyclides.

Cyclides are defined in Section 9.13 in [AgoM04] by means of geometric constructions that make it easier to deduce some of their geometric properties, but for computation purposes it is useful to have both a parametric and an implicit definition. We give such definitions for central cyclides whose spine curves are an ellipse and hyperbola and which are in standard position as shown in Figure 12.22. More precisely, we assume that the ellipse $\mathbf{E}$ and hyperbola $\mathbf{H}$ with branches $\mathbf{H}_1$ and $\mathbf{H}_2$ are defined by equations

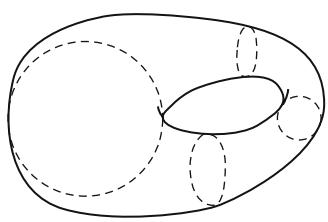


Figure 12.21. A central ring cyclide.

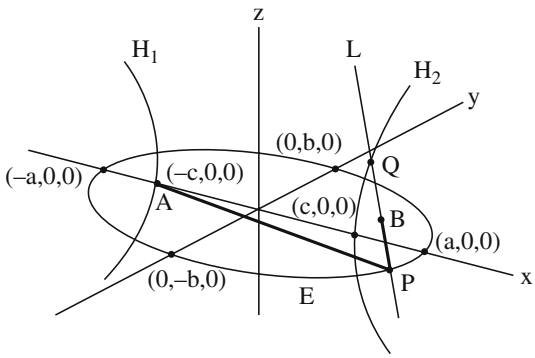


Figure 12.22. A central cyclide in standard position.

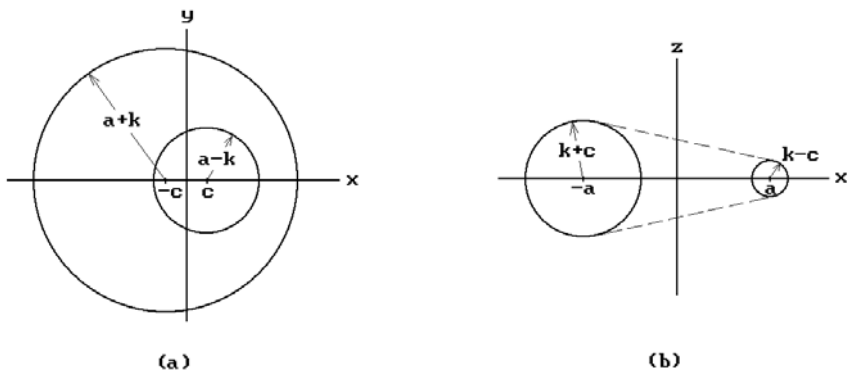


Figure 12.23. Projections of the central cyclide in Figure 12.22.

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}=1, \quad z=0, \quad 0<b<a, \tag{12.61a}$$

and

$$\frac{x^2}{c^2}-\frac{z^2}{b^2}=1, \quad y=0, \quad c^2=a^2-b^2, \tag{12.61b}$$

respectively. In Figure 12.22 the cyclide is then the set of points traced out by the end **B** of a taut string of fixed length $L = |\mathbf{AP}| + |\mathbf{PB}|$ that is tied to the focus **A** of the ellipse and touches the ellipse at its points **P**. The line **L** through **P** and **B** would meet the hyperbola in a point **Q**. Let us write the length L in the form L = a + k. Then the intersection of the cyclide with the two planes z = 0 and y = 0 is shown in Figure 12.23. In both cases we get two circles with radii that are simple functions of a, c, and k. Another way of thinking of the cyclide is as the envelope of spheres with centers on the two conics. Looking again at Figure 12.22, the spheres in question would have

radius **PB** if the sphere is centered on a point **P** of the ellipse and radius **QB** if it is centered on a point **Q** of the hyperbola.

12.13.1 Theorem. The central cyclide with the curves defined in (12.61) as its focal curves has a parameterization

$$p(\theta,\psi)=\left(\frac{k(c-a\cos\theta\cos\psi)+b^2\cos\theta}{a-c\cos\theta\cos\psi},\frac{b\sin\theta(a-k\cos\psi)}{a-c\cos\theta\cos\psi},\frac{b\sin\psi(c\cos\theta-k)}{a-c\cos\theta\cos\psi}\right), \quad (12.62)$$

where $0 \le \theta, \psi < 2\pi$, and is defined by the equation

$$\left(x^2+y^2+z^2-k^2+b^2\right)^2=4(ax-ck)^2+4b^2y^2. \quad (12.63)$$

Proof. See [Fors12]. We shall only sketch a derivation of the parameterization in (12.62), which can also be found in [Gray98] or [Boeh90]. Let us parameterize the points **P** on the ellipse **E** and points **Q** on the hyperbola **H** by

$$\begin{aligned}\theta &\to P(a\cos\theta, b\sin\theta), \quad \text{and}\\ \psi &\to Q(c\sec\psi, b\tan\psi).\end{aligned}$$

One shows that

$$|\mathbf{AP}|=|a+c\cos\theta| \quad \text{and} \quad |\mathbf{PQ}|=|a\sec\psi-c\cos\theta|.$$

If we write the length $L = |\mathbf{AP}| + |\mathbf{PB}|$ in the form $L = a + k$, then we can show that $|\mathbf{PB}| = |\alpha|$ and $|\mathbf{BQ}| = |\beta|$, where

$$\alpha=k-c\cos\theta \quad \text{and} \quad \beta=a\sec\psi-k.$$

In fact, the signed quantities α and β are such that **B** can be expressed in barycentric form as

$$\mathbf{B}=\frac{1}{\alpha+\beta}(\alpha\mathbf{P}+\beta\mathbf{Q}).$$

If we express all the variables in this representation of **B** in terms of θ and ψ, we will get the formula $p(\theta,\psi)$ in equation (12.62).

Theorem 12.13.1 makes it easy to work with a ring central cyclide. Its intersection with the planes $z = 0$ and $y = 0$ determine it completely. Therefore, one can use these cross-sections to visualize the surface and manipulating it is simply a matter of changing the values of a, c, and k. Also, equation (12.63) shows that the central cyclide is a fourth-degree surface. Parabolic cyclides are surfaces of degree three.

Here are a few more details about cyclides. There is a natural two-level geometric classification of cyclides (see [Boeh90] and [ChDH89]). At the top level there is the division of cyclides into central, parabolic, revolute, or degenerate cyclides depend-

ing on whether their spines are central conics, parabolas, straight lines and circles, or degenerate conics. Each of these types of cyclides can be further subdivided into three subtypes. We describe that subdivision in the case of central cyclides. Assume that the coordinate system is again chosen so that the cyclide is in standard position as shown in Figure 12.22.

Definition. The central cyclide is called a *ring cyclide* if $0 \le c < k \le a$. It is called a *horned cyclide* if $0 < k \le c < a$. It is called a *spindle cyclide* if $0 \le c \le a < k$.

Visually, a horned cyclide looks like two crescent-shaped solids that meet at their pinched points. A spindle cyclide has two pinch points at which the two parts of the entire surface meet with one part looking like a spindle inside the other. Ring cyclides, such as the one shown in Figure 12.21, are the easiest to draw because they do not have any of these degeneracies. Another way to describe these subtypes is in terms of two important values

$$s = \frac{ka}{c} \quad \text{and} \quad t = \frac{kc}{a}.$$

Let $\mathbf{L}_s$ be the line in the x-y plane that is parallel to the y-axis and passes through the point (s,0,0) on the x-axis. Let $\mathbf{L}_t$ be the line in the x-z plane that is parallel to the z-axis and passes through the point (t,0,0) on the x-axis. We will have a ring cyclide if $\mathbf{L}_s$ does **not** intersect the focal ellipse **E** and $\mathbf{L}_t$ does **not** intersect the focal hyperbola **H**. We will have a horned cyclide if $\mathbf{L}_s$ intersects the ellipse **E** and a spindle cyclide if $\mathbf{L}_t$ intersects the hyperbola **H**. The lines $\mathbf{L}_s$ and $\mathbf{L}_t$ are called the *characteristic lines* of the cyclide ([ChDH89]).

Some properties of cyclides that make them attractive to CAGD are (see [Prat90]):

(1) Each curvature line is a circle and cyclides are the only fourth-degree surfaces whose curvature lines are circles. The curvature lines split into two families similar to what happens in the case of a torus.
(2) The planes of each family of curvature lines meet in a line.
(3) One gets offset surfaces for a cyclide by changing the parameter k. Furthermore, the offset of a cyclide is a cyclide.
(4) For each curvature line, the angle between the principal normal of that curve and the surface normal is constant. Hence there is a right circular cone tangent to any circular curvature line of the cyclide.
(5) The inversion in a sphere of a cyclide with respect to a point not on it is again a cyclide.

One can show that the change of parameters

$$u = \tan\frac{\theta}{2}, \quad v = \tan\frac{\psi}{2}$$

produces a rational biquadratic parameterization of the cyclide. The lines of constant u and v correspond to lines of curvature, namely circles. If we call a region in a cyclide bounded by lines of curvature a *principal cyclide patch*, then we can use the rational

parameterization in u and v to define a Bézier parameterization for such a patch. See [Mart82], [MaPS86], [Prat90], [Boeh90], [DuMP93], [Prat95], and [KraM00]. This is very useful for representing cyclides in a CAGD program. One factor restricts the principal cyclide patch, namely, its four corners lie on a circle. This means that once one has picked two adjacent sides of a patch one has only one degree of freedom to pick the fourth corner since it must lie on the circle determined by the other three. One can also define triangular Bézier patches ([AlbD97]). Conditions for obtaining composite cyclide patches that join with G^1 continuity are discussed in [KraM00].

For cyclide intersections see [MaPS86] and [John93].

We already mentioned at the beginning of this section that cyclides are useful in blending. They are also useful in controlling the fairness of a surface. The advantage of cyclides is that they provide a manageable representation of a larger piece of a surface. Tensor product Bézier or B-spline patches deal with smaller pieces. This has led to a search for other general fourth-degree algebraic surfaces that might be useful in CAGD. One such class of surfaces are the *supercyclides* that are projective images of Dupin cyclides. These are special cases of so-called *double-Blutel* surfaces. For more on these generalizations see [Dege94], [Prat96], and [Prat97]. A unified theory of supercyclides is described in [Dege98].

12.14 Subdivision of Surfaces

Like in the case of curves, being able to subdivide surfaces is important in a variety of applications. Subdivision problems come in two flavors. In one case we have a parametric surface and in the other we have no parameterization but simply a polygonal surface defined by an arbitrary (not necessarily rectangular) grid of points. This section makes a few comments about the first case. The second is dealt with in Section 12.17.

At one level, subdividing a parametric surface simply amounts to subdividing its domain. On the other hand, if we are dealing with a surface defined by control points or control points and knots, then the more interesting question is how one can add new control points or knots. The tensor product surface case is quite straightforward and reduces to subdividing curves in the u- and v-direction and hence is basically a one-dimensional problem. The triangular surface case is more involved. Blossoms come in very handy here. One way to subdivide a triangle is shown in Figure 12.24. We add new vertices at the midpoints of the edges. The four new triangles give rise to four new triangular nets. The main issue is to do computations with respect to the smaller triangles as efficiently as possible by judicious use of the de Casteljau algorithm. We refer the interested reader to [Gall00]. There are other ways to subdivide triangles and [Fili86] suggests that the choice of subdivision should be made adaptively.

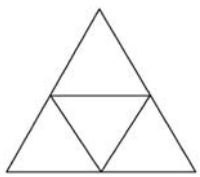


Figure 12.24. Subdividing triangular domains.

12.15 Composite Surfaces and Geometric Continuity

This section looks at conditions that ensure that a collection of surface patches that meet along boundary curves will define a globally smooth surface. For more information and references to work in this area see [FauP79], [Mort85], [Greg89], [HosL93], or [Fari97].

The idea here is basically the same as it was for curves. As a set, the union **S** of the sets that are traced out by the individual C^k parameterizations should be a smooth surface. Of course, since there are now two parameters, things are somewhat more involved computationally. Now C^k continuity would mean that the parameterization of the union **S** induced by the individual parameterizations be C^k. This requirement is again stronger than needed.

Definition. Two parameterized surface patches p(u,v) and q(u,v) meet along their boundary with *kth-order geometric continuity*, or *G^k continuity*, if there is a reparameterization r(u,v) equivalent to p(u,v) so that r(u,v) and q(u,v) meet with C^k continuity.

One can show that G^1 continuity simply means that the two patches have the same tangent planes at the points where they meet, so that G^1 continuity is sometimes referred to as *tangent plane continuity*. Since the analog of tangent lines for curves is tangent planes for surfaces, this is certainly a natural first condition for patches to meet in a way that is visually smooth. We mentioned earlier that one can get G^1 continuity with Gregory patches. Here we shall consider some simple conditions that will achieve this for tensor product and triangular patches. Section 15.2 discusses additional curvature-related criteria.

First of all, let p(u,v) and q(u,v) be two rectangular Bézier patches with domain $[a,b] \times [c,d]$ and $[a,b] \times [d,e]$ and control points $\mathbf{p}_{ij}$, $0 \le i \le m$, $0 \le j \le n$, and $\mathbf{p}_{ij}$, $0 \le i \le m$, $n \le j \le 2n$, respectively. The condition that they meet with G^1 continuity along the boundary curve defined by the points $\mathbf{p}_{in}$, $0 \le i \le n$, is that the three points $\mathbf{p}_{i,n-1}$, $\mathbf{p}_{in}$, $\mathbf{p}_{i,n+1}$ are collinear for $0 \le i \le n$. See Figure 12.25. The reason is that the normal for the tangent plane is the cross-product of the partials in the u- and v-direction. Since the boundary curve has the same control points for both patches, we only need the tangent vector in the v-direction to be parallel. For C^1 continuity, collinearity is

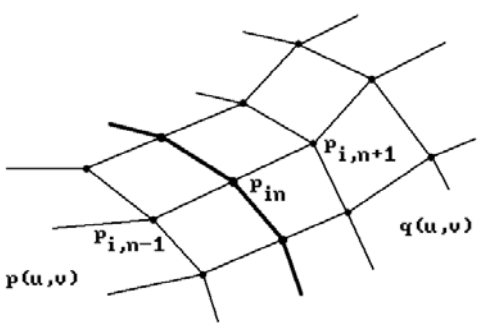


Figure 12.25. Collinearity condition for smoothly meeting Bézier patches.

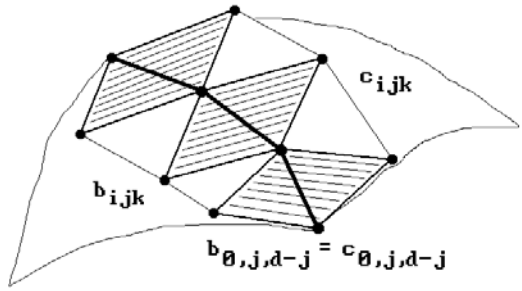


Figure 12.26. Ensuring that triangular Bézier patches meet smoothly.

not enough. The relative size of their domain comes into play and one has to add the condition that the points are in the **same** ratio (d–c):(e–d).

Next, consider triangular Bézier patches. Let p(u,v) and q(u,v) be two triangular Bézier patches of degree d defined by triangular nets $\{\mathbf{b}_{ijk}\}$ and $\{\mathbf{c}_{ijk}\}$, respectively, and assume that they meet along the edge u = 0 and $\mathbf{b}_{0,j,d-j} = \mathbf{c}_{0,j,d-j}$. A necessary condition for the patches to meet with G^1 continuity is that the triangles from the two patches that meet in a common edge are coplanar. For C^1 continuity those triangles for each patch must be the image of the **same** affine map. See Figure 12.26.

Finally, we look at bicubic patches as defined in Section 12.9. An interpolating bicubic B-spline surface can be thought of as a collection of such patches, but we want to go in the other direction. Given a rectangular grid of points $\mathbf{p}_{ij}$, $0 \le i \le m$, $0 \le j \le n$, we would like to find conditions on the geometric coefficients of the bicubic patches defined by the individual rectangles of the grid that will guarantee a globally smooth surface. We could transform a bicubic patch into a Bézier patch and use what we know about Bézier patches, but here we want to approach the problem directly.

We begin by analyzing the conditions under which two bicubic patches meet smoothly. Let p(u,v) be a patch defined by points $\mathbf{p}_{00}$, $\mathbf{p}_{10}$, $\mathbf{p}_{01}$, and $\mathbf{p}_{11}$ and q(u,v) a patch defined by points $\mathbf{q}_{00}$, $\mathbf{q}_{10}$, $\mathbf{q}_{01}$, and $\mathbf{q}_{11}$. Assume that $\mathbf{q}_{00} = \mathbf{p}_{10}$, $\mathbf{q}_{01} = \mathbf{p}_{11}$, and that the patches meet along the boundary curve

$$\gamma(v) = p(1,v) = q(0,v).$$

See Figure 12.27, where we use the abbreviations

$$\mathbf{p}_{ij}^{u} = p_u(i,j) \quad \text{and} \quad \mathbf{p}_{ij}^{uv} = p_{uv}(i,j)$$

and similar abbreviations for q. To get C^1 continuity where the patches meet it must be possible to find a change of coordinates of the function q(u,v) in the v direction, so that the new function has the same derivatives along γ as the function p(u,v). This means that $\partial q/\partial u$ and $\partial p/\partial u$ must be multiples of each other along γ. It follows that having

$$\mathbf{q}_{00}^{u} = a\mathbf{p}_{10}^{u}, \quad \mathbf{q}_{00}^{uv} = a\mathbf{p}_{10}^{uv}, \quad \mathbf{q}_{01}^{u} = a\mathbf{p}_{11}^{u}, \quad \text{and} \quad \mathbf{q}_{01}^{uv} = a\mathbf{p}_{11}^{uv},$$

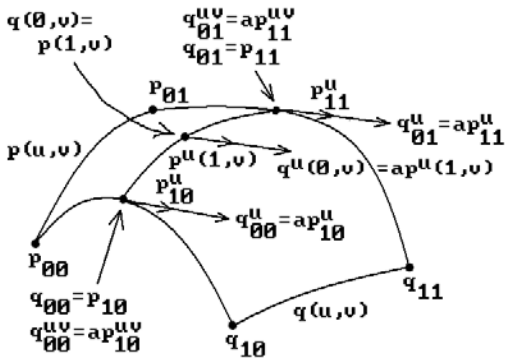


Figure 12.27. Making bicubic patches meet smoothly.

for some constant a, are sufficient conditions for achieving C^1 continuity. These conditions in fact imply that

$$\frac{\partial q}{\partial u}(0,v) = a\frac{\partial p}{\partial u}(1,v), \quad \text{for all v in } [0,1].$$

If our patches had met in a boundary curve in the u direction, say p(u,1) = q(u,0), the analogous conditions would be

$$\mathbf{q}^{v}_{00} = b\mathbf{p}^{v}_{01}, \quad \mathbf{q}^{uv}_{00} = b\mathbf{p}^{uv}_{01}, \quad \mathbf{q}^{v}_{10} = b\mathbf{p}^{v}_{11}, \quad \text{and} \quad \mathbf{q}^{uv}_{10} = b\mathbf{p}^{uv}_{11},$$

for some constant b. In the case of a network of patches, the condition for meeting with C^1 continuity along the boundary curves translates into the fact that we have lost quite a few degrees of freedom in defining the geometry of the surface. At each point of the network where four patches meet, instead of having 48 degrees of freedom (each of the four patches would have an arbitrary corner vertex, two tangent vectors, and one twist vector) we only have 14 (one vertex, two tangent vectors, one twist vector, and two constants). To put it another way, whereas the geometric matrices of a set of mn independent patches would normally be defined by 16mn vectors, given the aforementioned constraints, a C^1 continuous patch network can be represented by 4(m + 1)(n + 1) vectors and m + n scalars, namely, by a four-vector grid, $\mathbf{p}_{ij}$, $\mathbf{p}^{u}_{ij}$, $\mathbf{p}^{v}_{ij}$, and $\mathbf{p}^{uv}_{ij}$, and two sequences of scaling factors a_i and b_j. The geometric matrix of the ijth patch would then have the form

$$\begin{pmatrix} \mathbf{p}_{ij} & \mathbf{p}_{i,j+1} & \mathbf{p}^{v}_{ij} & b_{j+1}\mathbf{p}^{v}_{i,j+1} \\ \mathbf{p}_{i+1,j} & \mathbf{p}_{i+1,j+1} & \mathbf{p}^{v}_{i+1,j} & b_{j+1}\mathbf{p}^{v}_{i+1,j+1} \\ \mathbf{p}^{u}_{ij} & \mathbf{p}^{u}_{i,j+1} & \mathbf{p}^{uv}_{ij} & b_{j+1}\mathbf{p}^{uv}_{i,j+1} \\ a_{i+1}\mathbf{p}^{u}_{i+1,j} & a_{i+1}\mathbf{p}^{u}_{i+1,j+1} & a_{i+1}\mathbf{p}^{uv}_{i+1,j} & a_{i+1}b_{j+1}\mathbf{p}^{uv}_{i+1,j+1} \end{pmatrix}.$$

12.16 Fairing Surfaces

Fairing surfaces is, like in the case of curves, a question of achieving a pleasing shape. It is a much more complicated problem than fairing curves because the obvious approach to reduce the problem to a one-dimensional one would involve the shape of a surface along an infinite number of directions at every point. In any case, there are again two sides to the problem. First, one needs some tools to detect any imperfections in the shape and, second, one needs to describe ways to correct these imperfections.

Curvature is again key to the general detection process, but, just like the fact that a simple number, such as the slope for real-valued functions, cannot capture the idea of derivative for vector-valued functions, it is not easy trying to capture the idea of curvature at a point on a surface with real-valued functions. Some basic functions that have been used are

(1) Gauss curvature,
(2) mean curvature, and
(3) *absolute curvature* ($|\kappa_1| + |\kappa_2|$, where κ_1 and κ_2 are the principal normal curvatures).

Determining imperfections in the fairness of a surface boils down to making sure that plots of these curvature functions have appropriate shapes. In particular, places where the sign of the curvature function changes too often are potential problem spots. A problem with Gauss curvature is that it is zero for ruled surfaces and hence gives no useful information in those cases.

Other specialized tools have been used. Hagen and Hahmann ([HagH95]) use stability concepts based on infinitesimal bendings to control the shape of a surface. Séquin et al. ([SéCM95]) faired surfaces by minimizing functionals based on the arc-length derivative of normal curvature, although this turned out to be very expensive computationally. Other variational approaches are described by Sarraga ([Sarr98]). Net fairing methods are described in [SuLi89] in case a surface is defined by a network of curves.

Another method that has been used in designing pleasing car body shapes is based on *reflection lines* ([Klas80]). Reflection lines are the patterns that one sees on the polished surface of a car caused by parallel lines of light from light sources. See Figure 12.28. The criterion for a nice shape is that these patterns look "nice." Curvature plots

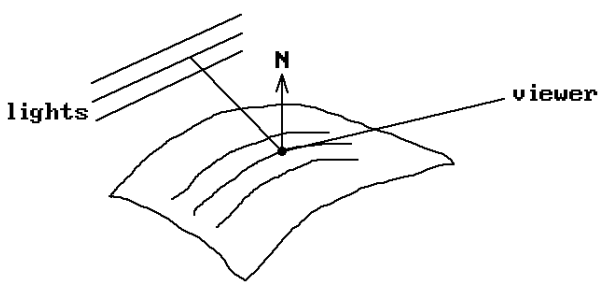


Figure 12.28. Detecting surface imperfections with reflection lines.

tend to detect local imperfections whereas reflection lines catch more global problems. A related approach is the method of *isophotes* ([Pösc84], [HutH96]) that analyzes lines of equal light intensity. There are, as one can see, a number of special curves on a surface that can be used to analyze it instead of using the surface curvature directly. Theisel and Farin ([TheF97]) show how one can determine the curvature of contour lines, lines of curvature, asymptotic curves, reflection lines, and isophotes without computing the curves directly. For more on surface interrogation algorithms see [Hage92] or [FolR93]. Rather than having the designer use surface analysis tools to fair a surface him/herself, Rando and Roulier ([RanR91]) describe a system that fairs parametric surfaces automatically. Moreton and Séquin ([MorS92]) describe a way to design smoothly shaped surfaces of any genus.

The problem of fairing discrete surfaces has also been studied. See [SuLi89]. A discussion on the fairing of subdivision surfaces can be found in [ZorS99]. One motivating observation is that obtaining a fair surface may be more important than obtaining a smooth one. If one has a polygonal mesh, one has more freedom to make changes. Discrete analogs of curvature are available. One can always get a smooth surface at the end if that is needed, an interpolatory B-spline surface in fact.

Finally, if problems with a surface's shape are detected, they need to be corrected. In the case of tensor product surfaces one can try some curve-smoothing methods on the curves in the two coordinate directions. For much more about surface fairing see [HosL93]. Section 15.2 has more about fairing and curvature.

12.17 Recursive Subdivision Surfaces

This section is about polygonal surfaces and ways to turn them into smoother looking objects. There are basically two approaches. One (see, for example, [Pete95]) is to define parametric patches for each facet and ensure that they all meet in a smooth manner. The other uses a recursive subdivision process that smooths the corners and edges to get a closer and closer polygonal approximation of a smooth surface. It is these recursive subdivision surfaces that we want to discuss in this section. There are two ways of looking at such surfaces. From one point of view, we can consider the original polygonal surface as having been a rough outline of some specific smooth one and that reconstruction of this smooth surface is our goal. One reason that one may not have had or even care about an actual parameterization for the smooth surface is that surfaces that have reasonable parameterizations with the usual rectangular or triangular domains are somewhat limited in their shape or topology unless one is willing to put up with what may be unpleasant singularities. Even a space as simple as a sphere cannot be parameterized without singularities unless one uses several patches to cover it. The other point of view is that recursive subdivision surfaces are a wholly new class of surfaces that are interesting in their own right independent of any associated smooth surface.

A number of polygonal surface subdivision algorithms are known. An excellent overview of the subject and its applications can be found in [ZorS99]. See also [CavM89] and [Sabi90]. One large class of such algorithms can be classified by whether they use a "corner cutting" or vertex insertion approach. In the latter case one can distinguish further based on whether they generate quadrilateral or

triangular meshes and whether they approximate or interpolate. An important question for all these algorithms is whether repeated application produces a sequence of surfaces that converges to a surface and how smooth this limit surface is. See [Reif95].

The first subdivision algorithm that we shall describe is the Doo-Sabin algorithm ([DooS78]). This algorithm is a corner-cutting algorithm and is a generalization of the Chaikin curve subdivision algorithm that we described in Section 11.14. The Doo-Sabin algorithm starts with a polygonal surface **S** that has been defined by a set of vertices together with a specification of its edges and faces. (Actually, the faces do not need to be planar for the algorithm and all we need is a polygon-type data structure and not a real polygon.) One then defines some new vertices for each face **F**, one for each vertex of **F**, and this new set of vertices will then become the vertex set of the subdivided surface **S′**. The faces (and edges) of **S′** are determined by the following three rules:

(1) If **F** was an n-sided face of **S**, then the n new vertices associated to **F** will become a face of **S′** and is called an *F-face*. See Figure 12.29(a).
(2) If **E** is an edge of **S** that belongs to faces **F** and **F′** of **S**, then the four vertices of **S′** created for the endpoints of **E** in **F** and **F′** define a face of **S′** called an *E-face*. See Figure 12.29(b). No face is associated to a boundary edge of **S**.
(3) If **V** is a nonboundary vertex of **S** that belonged to n faces of **S**, then the n vertices of **S′**, associated to the vertex **V** in those n faces, define a face of **S′** called an *V-face*. See Figure 12.29(c).

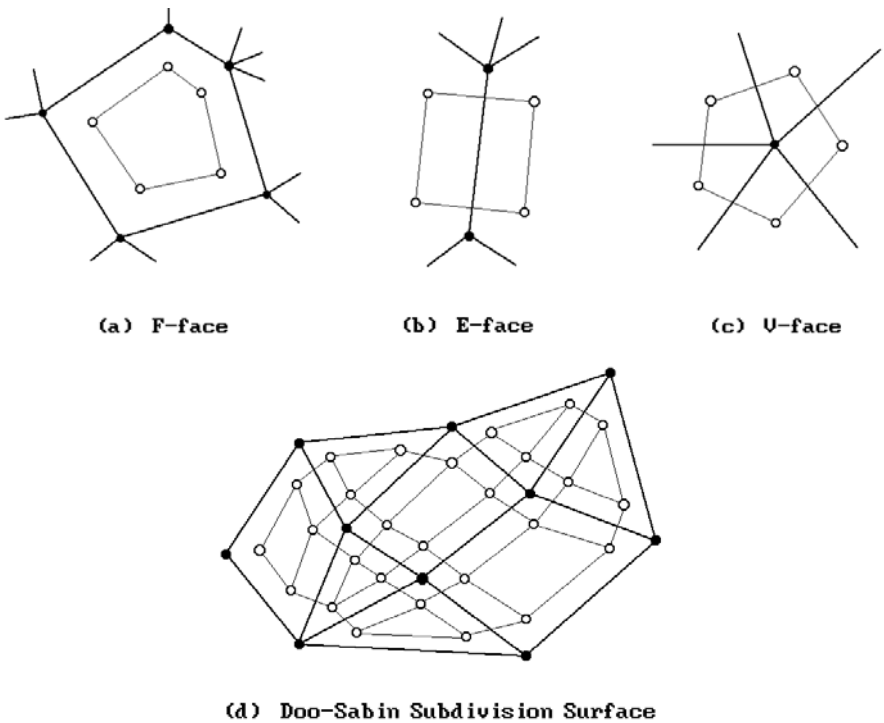


Figure 12.29. The new vertices of the Doo-Sabin algorithm.

Figure 12.29(d) shows the result of applying the algorithm to the surface **S** whose vertices are the solid circles and the edges are drawn as bold lines. The vertices and edges of the subdivided surface **S′** are the hollow circles and thin lines, respectively.

To finish the Doo-Sabin algorithm, all that is left to do is to explain how the new vertices are defined in each face of the original surface **S**. Various schemes exist. Let $\mathbf{v}_i$ be the new vertex created for vertex $\mathbf{w}_i$ in an n-sided face **F** of **S**. The original idea was to let $\mathbf{v}_i$ be the midpoint of the segment from $\mathbf{w}_i$ to the centroid of **F**. Another expresses $\mathbf{v}_i$ in the form

$$\mathbf{v}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{w}_j$$

and sets

$$\alpha_{ij} = \frac{n+5}{4n} \qquad \text{for} \quad i = j,$$

$$\alpha_{ij} = \frac{3 + 2\cos\dfrac{2\pi(i-j)}{n}}{4n} \qquad \text{for} \quad i \neq j.$$

The α_{ij} are not as strange as they may look at first glance. They are closely related to the nth roots of unity. In fact, they are barycentric coordinates because

$$\sum_{j=1}^{n} \alpha_{ij} = 1.$$

This follows easily from the fact that the sum of the real parts of the nth roots of unity is 0, namely,

$$\sum_{k=1}^{n} \cos\left(\frac{2\pi k}{n}\right) = 0.$$

The following observations can be made about the subdivision surfaces:

(1) After the first subdivision all vertices will be incident to four edges.
(2) Each n-sided face gives rise to an n-sided face in the subdivision but the number of n-sided faces, $n \neq 4$, stays constant. As one keeps subdividing, such faces get smaller and smaller and converge to the centroid of the original face.
(3) We will get more and more four-sided faces and our subdivision will look more and more like a rectangular grid.
(4) The surfaces satisfy the convex hull and local control property. They are also affinely invariant.

Observation (3) means that we can, except for a fixed number of arbitrarily small regions, express the last subdivision surface as a collection of 3×3 rectangular grids. These 3×3 grids can be replaced by a quadratic Bézier surface, so that our subdivisions start looking more and more like quadratic Bézier surfaces. The parameterization of the limit surface is C^1 (and has well-defined tangent planes) except at a finite number of points.

An improved Doo-Sabin algorithm that gives one control over the boundary and also allows for interpolation is described in [Nasr87]. An even simpler subdivision construction than the one used in the Doo-Sabin algorithm, called the mid-edge subdivision, is analyzed in [PetR97]. One disadvantage of the Doo-Sabin algorithm is that, being based on quadratic B-splines, it shares a problem with those, namely, that the limit surface may bulge too much to the control points.

The next subdivision algorithm we would like to describe is the Catmull-Clark algorithm ([CatC78]), which is a vertex insertion algorithm that produces a quadrilateral mesh. Given a polygonal surface $\mathbf{S}$, we create a new vertex for each of its faces, edges, and vertices. The new vertex $\mathbf{v}_F$ for a face $\mathbf{F}$ of $\mathbf{S}$ is just its centroid, that is, if $\mathbf{v}_1, \mathbf{v}_2, \ldots,$ and $\mathbf{v}_k$ are the vertices of $\mathbf{F}$, then

$$\mathbf{v}_F = \frac{1}{k}\sum_{i=1}^{k} \mathbf{v}_i.$$

If $\mathbf{E}$ is an edge of $\mathbf{S}$ with vertices $\mathbf{v}$ and $\mathbf{w}$ which is adjacent to faces $\mathbf{F}$ and $\mathbf{G}$, then the new vertex $\mathbf{v}_E$ for $\mathbf{E}$ is the centroid of $\mathbf{v}$, $\mathbf{w}$, $\mathbf{v}_F$, and $\mathbf{v}_G$, that is,

$$\mathbf{v}_E = \frac{1}{4}(\mathbf{v} + \mathbf{w} + \mathbf{v}_F + \mathbf{v}_G).$$

Finally, there are several ways to define the new vertex $\mathbf{v}_v$ for a vertex $\mathbf{v}$ of $\mathbf{S}$. We follow the original definition given in [CatC78]. Let $\mathbf{F}_{av}$ denote the average of all the $\mathbf{v}_F$, where $\mathbf{F}$ is face of $\mathbf{S}$ to which $\mathbf{v}$ belongs. Let $\mathbf{E}_{av}$ denote the average of all the midpoints of all the edges $\mathbf{E}$ of $\mathbf{S}$ which contain $\mathbf{v}$ and assume that there are n such edges. Then

$$\mathbf{v}_v = \frac{1}{n}\mathbf{F}_{av} + \frac{2}{n}\mathbf{E}_{av} + \frac{n-3}{n}\mathbf{v}.$$

The subdivision $\mathbf{S}'$ of $\mathbf{S}$ is now defined as follows: We create an edge from each new $\mathbf{v}_F$ to all the new $\mathbf{v}_E$, where $\mathbf{E}$ is an edge of $\mathbf{F}$, and also an edge from each new $\mathbf{v}_v$ to each $\mathbf{v}_E$, where $\mathbf{v}$ belongs to $\mathbf{E}$. See Figure 12.30. In that figure the vertices $\mathbf{v}_F$ are marked by a cross, the vertices $\mathbf{v}_E$ are marked by hollow circles, and the vertices $\mathbf{v}_v$ are marked by hollow squares. After each subdivision in this algorithm all faces are four-sided and most vertices belong to four edges, but there may be a constant number of vertices that are incident to n edges, $n \neq 4$. In the limit one gets a surface that is essentially built from cubic B-spline surfaces and has a parameterization that is C^2 except at a finite number of points. Although tangent planes are defined everywhere, the singular points may exhibit curvature problems in that one may either get

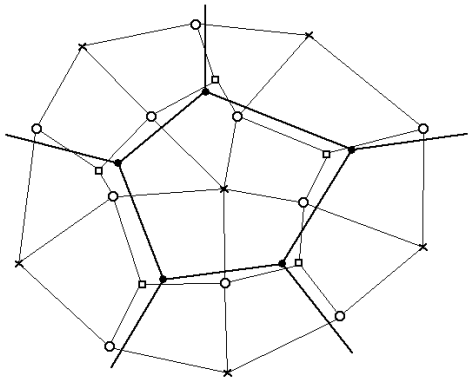


Figure 12.30. Catmull-Clark subdivision surface.

unbounded results when trying to estimate the curvature there or the curvature may be zero.

A final subdivision algorithm we would like to mention is the Loop algorithm ([Loop87]). This is another vertex insertion algorithm that produces a triangular mesh and a similar C^2 parameterization. Details of the algorithm can also be found in [Gall00].

The limit surfaces that one gets from subdivision algorithms like the ones discussed in this section are often called *recursive subdivision surfaces*. It is shown in [Stam98] and [ZorS99] that points and derivatives of Catmull-Clark and Loop limit surfaces can be computed directly without subdividing. The advantage of vertex insertion algorithms is that there is a natural way to associate the vertices of the original surface with vertices of the subdivided surface. For a more thorough discussion of the advantages and disadvantages of various subdivision schemes see [ZorS99]. For subdivision surfaces that interpolate a given set of possibly intersecting curves see [Nasr00].

12.18 Summary for Curves and Surfaces

In the last two chapters we have covered a great many topics on curves and surfaces. However, anyone wanting to develop a modeling system would quickly find that they have been presented with lots of choices. The intent of this section is to discuss some of the hard decisions that inevitably have to be made with regard to questions such as the following:

(1) What types of curves and surfaces should be supported?
(2) What sort of mathematical functions should be used for the parameterizations?
(3) Which algorithms should be used to compute values and derivatives for the chosen types?

Constructing curves and especially surfaces is not an easy task and so a user would want a modeling system to support as wide a range of curve and surface construc-

tions as possible. On the other hand, providing a user with many predefined types of curves and surfaces does not mean that all their underlying mathematical representations have to be implemented differently. For example, one would certainly allow users to define spheres and surfaces of revolution. The question for the modeling system implementer is whether to use special case formulas to parameterize these or describe them with, say, Bézier or B-spline surfaces. Having separate implementations usually means a gain in efficiency, but it also means that one has to maintain all these implementations. Some newer modeling systems are based on NURBS curves and surfaces and make these the only internal geometric types. These types are general enough to be able to represent all the curves and surfaces that one typically wants. The advantage of the NURBS approach is that one has to implement operations such as evaluation and differentiation only for these two types. (Actually, there would really only be one type of computation since the surface computations reduce to curve computations.) Sections 11.5.4 and 12.12.5 presented a few efficient algorithms for such computations. For a very detailed discussion of algorithms and constructions using NURBS see [PieT95]. One disadvantage with NURBS is, of course, that no matter how efficient they are, they will not be as simple or fast as special-case algorithms, although with regard to speed, computers have become powerful enough so that this is no longer such a great disadvantage. It should also be pointed out that there exist explicit matrix formulations for NURBS curves and surfaces that produce more efficient evaluation algorithms but are complex and have numerical stability problems. See [LiuW02]. We should note further that the Gregory patch and its generalizations achieved a lot of popularity for blending and surface design in general.

A related question that someone new to the subject might have is whether to use Bézier, spline, or B-spline curves and surfaces. Mathematically, these three types cover the same class of curves and surfaces. Only their representations are different but one can switch back and forth between them. The only way to make the question meaningful is to rephrase it as asking whether one wants to think of curves and surfaces in an interpolatory or interactive (control point) way. Because both interpolation and interactive manipulation are important, it would make sense for a modeling system to support both representations since each is more efficient in its own domain.

With respect to the question of computation, it is known that the Bézier form of a curve or surface with the de Casteljau algorithm is numerically more stable than the polynomial form (although Horner's method for evaluating polynomials makes the latter more efficient for computational purposes). It is important to note however that to obtain this stability one should not switch back and forth between the two forms and needs to do everything in the Bézier form.

Here are a few comments on deciding whether to use rectangular or triangular domains for parameterizing functions.

Advantages of triangular domains:

(1) Sometimes one needs to parameterize triangular surfaces patches. If one only uses rectangular patches, then one has to play tricks such as mapping two corners of a rectangle to the same point. A good example of this is the case of a surface of revolution where the curve one is revolving touches the axis one is revolving about. Collapsing rectangle vertices leads to degeneracies and would not be necessary if one allowed a triangular domain.

(2) Parts of complex geometric objects, such as rounded corners, may have a more natural description with triangular patches.

Disadvantages of triangular domains:

(1) Triangular domains are not convenient for everything, so that it would not be desirable to restrict all parameterizations to have such domains.
(2) Triangular patches involve a specialized data structure which would have to be integrated into an existing modeling system. This could be expensive.

For a discussion of how one can define surface patches over domains that are arbitrary convex polygons see [LooD89].

Finally, recursive subdivision surfaces have an advantage over regular B-spline surfaces when one can define a polygonal structure that fits the surface better. On the other hand, one pays a price because a recursive structure is usually much harder to work with than simple formulas.

12.19 A Little Bit of History

We would like to end our discussion of curves and surfaces with a brief history of their development as far as it relates to design and manufacturing. A good reference is [Sabi90]. See also [NowR83], [Hoch83], [Elsa83], and [Fari83].

The earliest significant instance involving a systematic design of surfaces was in the building of boats. For hundreds of years craftsmen designed boat hulls by using a sequence of frames that defined the lateral shape of the hull. Planking was then applied to these frames to produce the hull. If the frames were defined well, then the planks produced a smooth shape. It took great skill to end up with a shape that had the desired property. Eventually, pencil and paper drawings were used to help in this process. Designing a boat involved roughly two stages:

The Design Stage. Some initial specifications for the hull led to a preliminary line plan that consisted of a collection of planar curves representing an orthogonal set of cross-sections of the targeted shape. Figure 12.31(b) shows some sample views of the boat in Figure 12.31(a). The goal of this stage was to resolve any potential problems caused by perhaps conflicting specifications.

The Fairing Stage. The goal of this stage was to adjust the line plan obtained in the first stage to obtain a smoother looking surface while still meeting the basic initial requirements. The process involved checking the shape of the surface along some other control sections. The two strategies used here were to either modify some lines after the fact or to have defined a parameterized set of lines to begin with and to tweak these parameters.

Basically, one would draw a sequence of frames on a drawing board to full scale and horizontal and vertical sections were determined. This required a large drawing board that was situated on the floor of the loft above the room where the ship was being built, hence the term "lofting." When ships became too big for full-scale

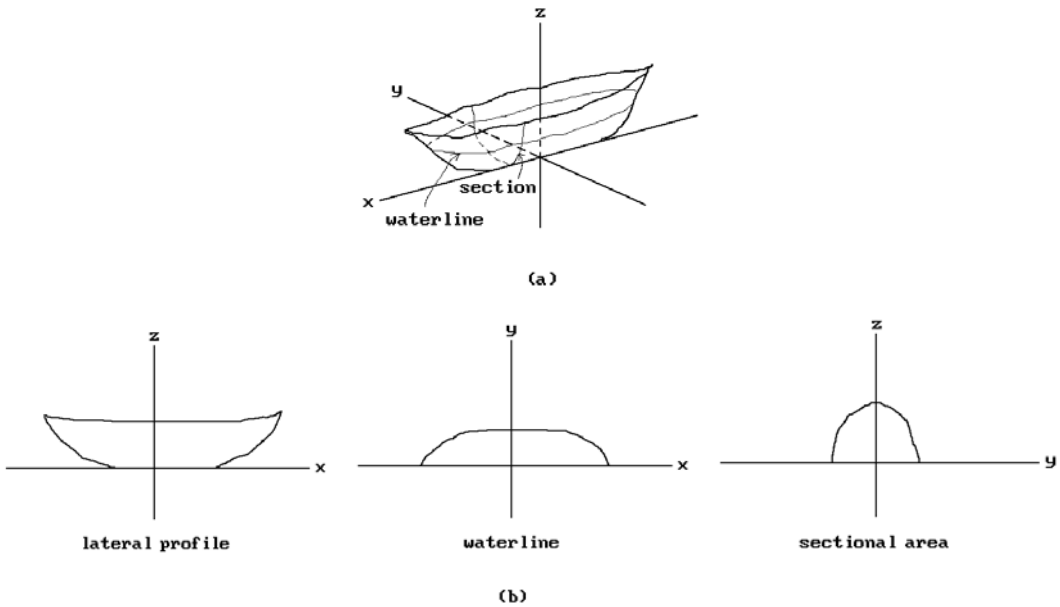


Figure 12.31. A simplified view of boat hull specification.

drawings, a smaller scale was used. Figure 12.31 shows a highly simplified view of what was specified. In the shown coordinate system one would define a number of orthogonal sectional views referred to as *sections* (x = constant), *buttocks* (y = constant), and *waterlines* (z = constant). From this data one could compute other control curves that would correspond to other sorts of sections. Their computation would need a skilled craftsman. If any of these had undesirable shape features, one would back up and change the initial form data and repeat the process until one was finally satisfied.

A similar process was used in the manufacture of other surfaces such as those needed in the aircraft and automobile industry. Using the approach just described meant that the process was based on curves and the fairing of curves. In the aircraft industry, "conic lofting" was popular, meaning that conic curves were used to describe sections of aircraft fuselages. In boat building, least squares fitting polynomials were found to be more useful. In the 1960s one started to use splines.

Another idea that started to take hold was the idea of fairing an entire surface patch because dealing with large meshes of curves and data points was very time-consuming and complicated. The papers of Ferguson ([Ferg64]) and Coons ([Coon67]) (an earlier 1964 version of this paper was well known) had a great influence and introduced the bicubic and Coons patch, respectively. From a single patch one moved on to a grid of patches that was defined automatically from the grid of boundary curves. Twist vectors were a major problem. Bézier ([Bézi71]) introduced Bézier curves and surfaces. The nice thing about these objects was that it was much easier to manipulate their shape. One simply moved the control points in their control polygon. A remaining problem was that the degree of the Bézier objects increased as the number of control points increased unless one resorted to piecewise Bézier objects. This

problem was solved when Gordon and Riesenfeld ([GorR74a]) suggested using B-splines as described by Schoenberg ([Scho67]). The term "NURBS" was coined at Boeing in 1985 ([BloK02]). See [Roge01] for other historical tidbits.

When the design and manufacturing process started to be computerized, one tried to imitate what was done by hand. Objects were defined by collections of curves. There were lots of possible curves, but mathematically one tried to define and manipulate them in terms of controlling features such as endpoints and tangents at the endpoints. Until the early 1980s most ship-fairing systems were based on fairing curves. Polynomial curves gave way to spline functions because, being piecewise polynomial, they avoided the global oscillation of large degree global polynomials and also allowed one to proceed in the same way that it was done with the physical splines by moving control points.

12.20 Exercises

Section 12.2

12.2.1 Consider the surface of revolution **S** obtained by revolving a space curve $g : [a,b] \to \mathbf{R}^3$ about a line **L** through a point **p** and direction vector **v**.

(a) Find a parameterization p(u,v) for **S**.
(b) Find a parameterization p(u,v) for **S** when g(u) = (0,0,u), [a,b] = [1,3], **p** = (1,2,0), and **v** = (2,–1,3). Also find ∂p/∂u, ∂p/∂v, and a normal vector at the point (0,0,1).

Section 12.3

12.3.1 Find an implicit equation f(x,y,z) = 0 for the surface **S** in Exercise 12.2.1(b). Use the gradient of f to find a normal to S at (0,0,1) and verify that it is parallel to the normal you got in Exercise 12.2.1(b).

Section 12.4

12.4.1 Consider the curves $f, g : [1,3] \to \mathbf{R}^3$, where $f(u) = (u + 1, u^2 - 4u + 5, 0)$ and $g(u) = (0, u, u^3)$. Let p(u,v) be the lofted surface defined by f and g. Sketch the surface and find the equation of the tangent plane at p(2,0.5).

Section 12.5

12.5.1 Consider the spiral $f : [0,\pi] \to \mathbf{R}^3$, f(t) = (3 cos t,3 sin t,t), and its Frenet frame (T(t),N(t),B(t)). Let p(u,v) be the sweep surface obtained by sweeping the segment [(0,0,–1),(0,0,1)] along f(t) rotating the segment in the N(t)-B(t) plane in a uniform counter-clockwise manner so that we have rotated through an angle of π when we reach f(π). Find the formula for p(u,v).

Section 12.7

12.7.1 Find the formula for the Coons surface p(u,v) defined by boundary curves p(u,0) = $(2u,u^2,0)$, p(u,1) = $(2u,-4u^2 + 8u + 1,0)$, p(0,v) = (0,v,0), and p(1,v) = $(-4v^2 + 4v + 2,4v + 1,0)$. Also find ∂p/∂u(0.5,0.5) and ∂p/∂v(0.5,0.5).

12.7.2 Prove Theorem 12.7.1.

Section 12.9

12.9.1 Prove that the matrix **Q** in equation (12.36a) is just the matrix **B** in equation (12.35).

12.9.2 Consider the bicubic patches p(u,v) with geometric matrices

(a) $$\mathbf{B} = \begin{pmatrix} (1,1,0) & (3,2,0) & (2,1,0) & (2,1,0) \\ (2,0,0) & (5,1,0) & (3,1,0) & (3,1,0) \\ (1,-1,0) & (2,-1,0) & (1,0,0) & (1,0,0) \\ (1,-1,0) & (2,-1,0) & (1,0,0) & (1,0,0) \end{pmatrix}$$

(b) $$\mathbf{B} = \begin{pmatrix} (1,0,0) & (0,2,3) & (-1,2,3) & (-1,2,3) \\ (3,0,0) & (3,2,3) & (0,2,3) & (0,2,3) \\ (2,0,0) & (3,0,0) & (1,0,0) & (1,0,0) \\ (2,0,0) & (3,0,0) & (1,0,0) & (1,0,0) \end{pmatrix}$$

(c) $$\mathbf{B} = \begin{pmatrix} (1,1,0) & (1,0,1) & \left(0,0,\frac{\pi}{2}\right) & \left(0,-\frac{\pi}{2},0\right) \\ (2,\sqrt{2},0) & (2,0,\sqrt{2}) & \left(0,0,\frac{\pi}{\sqrt{2}}\right) & \left(0,-\frac{\pi}{\sqrt{2}},0\right) \\ \left(1,\frac{1}{2},0\right) & \left(1,0,\frac{1}{2}\right) & \left(0,0,\frac{\pi}{4}\right) & \left(0,-\frac{\pi}{4},0\right) \\ \left(1,\frac{1}{2\sqrt{2}},0\right) & \left(1,0,\frac{1}{2\sqrt{2}}\right) & \left(0,0,\frac{\pi}{4\sqrt{2}}\right) & \left(0,-\frac{\pi}{4\sqrt{2}},0\right) \end{pmatrix}$$

Describe the surfaces defined by these patches geometrically like we did in Example 12.9.1.

12.9.3 (a) Let p(u,v) be the bicubic patch defined by Exercise 12.9.2(b). Consider the rectangle **A** = [1/4,3/4] × [1/3,2/3]. Let q(u,v) be the bicubic patch that is the subpatch p|**A** reparameterized to [0,1] × [0,1]. Find the geometric matrix for q(u,v).

(b) Generalize (a) to the case of an arbitrary subrectangle **A** = [a,b] × [c,d].

Section 12.12.3

12.12.3.1 The blossom for a triangular Bézier surface p(u,v) can be obtained from its control points $\mathbf{b}_{ijk}$ using the triangular de Casteljau algorithm similar to what was done in Exercise 11.5.2.1 for curves. To learn about it, see, for example, [Fari97].

12.21 Programming Projects

1. Display and manipulate surfaces (Sections 12.2–12.11, 12.12.1, 12.12.3, 12.12.4, 12.13)

   (a) Like in the case of curves, we shall simply suggest that the reader implement any one of the many types of surfaces in the listed sections. Because defining three-dimensional points is not easy to do interactively, allow the data to be read in from a file. B-spline and NURBS surfaces are of special interest like their curve counterparts.

   (b) Given a surface p(u,v), let the user input values for u and v and then display the partials $\partial p/\partial u$ and $\partial p/\partial v$ and the normal vector at p(u,v). A more interactive approach would be to let the user pick a point on the surface with the mouse and then display that data at the picked point. Finding the point that was picked amounts to sending a ray from the viewer through the picked point in the view plane and finding the nearest intersection of this ray with the surface patches in the world. An algorithm for finding the intersections is discussed in Section 13.4.1.

2. Interpolation (Sections 12.12.6)

   Implement some interpolating B-spline surfaces using data from files.

3. Recursive subdivision (Section 12.17)

   Allow a user to define a polygonal surface via data from a file and then show how it changes with recursive subdivision.

CHAPTER 13

Intersection Algorithms

Prerequisites: Basic calculus and vectors, Section 4.7 in [AgoM05] (Newton-Raphson method), Sections 10.4, 10.9, 10.10, and 10.15 in [AgoM05] for Sections 13.5.2 and 13.5.5

13.1 Overview

Set operations are common types of operations performed in geometric modeling. It is therefore important to have efficient algorithms for them, but mathematical solutions are only part of the problem. Set operation algorithms very often tend to involve relatively complex numerical computations and, even if they do not, they invariably are driven by the outcome of numeric tests, such as whether some quantity is zero or not. If there is ever a place where the shortcoming of a computer's finite precision arithmetic is apparent, then this is one of those places. Questions such as whether two lines or planes intersect, which are simple to answer mathematically, become complicated to answer in a consistent manner. Therefore, the **accuracy** and **robustness** of solutions are major criteria for evaluating algorithms. **Speed** is also important. It is possible to do infinite precision arithmetic on a computer, but the problem with this approach is speed. Next to infinite precision, interval arithmetic seems to be the best way to achieve accuracy. There are intersection algorithms for "interval" curves and surfaces, but they will not be discussed due to space and time limitations. The interested reader is referred to [HMSP96] and [HMPY97].

This chapter describes various approaches to finding intersections of objects. Actually, since finding intersections of linear objects reduces mathematically to solving basic linear algebra problems, we are mainly interested in the more complicated case of smooth object intersections. The only exception to this is Section 13.2, which describes a simple test for whether two convex linear polyhedra intersect. Sections 13.3–13.5 make up the bulk of the chapter. They describe algorithms for various types of intersections. Unfortunately, all we have time for is to describe the steps and the mathematics behind some of the algorithms and will have little to say on important implementation details for ensuring accuracy and robustness. The reader will have to

look up those sorts of details, insofar as they are available, in the references to specific algorithms. Some general references that give an overview of intersection algorithms and compare some of the methods used in these algorithms are [PraG86], [SedP86], [Luka89], [DoSY89], [Hoff89], [AzBB90], [BarK90], [SedN90], [Boen91], [Patr92], [Patr93], and [HosL93]. We conclude with a brief summary in Section 13.6.

A few general comments are in order before we get started. The problem of finding the intersection of two smooth objects is invariably reduced to the problem of finding the roots of an equation. The exact nature of this equation is determined by how the two objects $\mathbf{O}_1$ and $\mathbf{O}_2$ are presented, that is, whether they are defined parametrically or implicitly via equations. In the discussion below assume that $\mathbf{O}_1$ and $\mathbf{O}_2$ are m-dimensional objects in $\mathbf{R}^n$, that $\mathbf{x} \in \mathbf{R}^n$, and that $\mathbf{s}, \mathbf{t} \in \mathbf{R}^m$ are variables used to parameterize the objects. (For the curves and surfaces of interest to us, m will be 1 or 2, respectively, and n will be 2 or 3.)

Implicit/parametric Intersection. This is the easiest case. Suppose that $\mathbf{O}_1$ is defined by an equation

$$f(\mathbf{x}) = 0, \tag{13.1}$$

and that $\mathbf{O}_2$ is parameterized by a function $p(\mathbf{t}) = (p_1(\mathbf{t}), p_2(\mathbf{t}), \ldots, p_n(\mathbf{t}))$. The intersection $\mathbf{O}_1 \cap \mathbf{O}_2$ can be obtained by solving the equation

$$F(\mathbf{t}) = f(p_1(\mathbf{t}), p_2(\mathbf{t}), \ldots, p_n(\mathbf{t})) = 0, \tag{13.2}$$

which defines an implicit object in $\mathbf{R}^m$.

Implicit/implicit Intersection. Suppose that $\mathbf{O}_1$ and $\mathbf{O}_2$ are defined by equations

$$f(\mathbf{x}) = 0$$

and

$$g(\mathbf{x}) = 0,$$

respectively. One needs to solve

$$F(\mathbf{x}) = (f(\mathbf{x}), g(\mathbf{x})) = (0, 0) = \mathbf{0}.$$

Parametric/parametric Intersection. Suppose that $\mathbf{O}_1$ and $\mathbf{O}_2$ have parameterizations $p(\mathbf{s})$ and $q(\mathbf{t})$, respectively. We need to solve

$$F(\mathbf{s}, \mathbf{t}) = p(\mathbf{s}) - q(\mathbf{t}) = \mathbf{0}. \tag{13.3}$$

Because the implicit/parametric case is the easiest case, one often tries to convert to this case. This is one way how algebraic geometry enters the picture. Converting the parametric/parametric case means that we would use algebraic geometry to convert one of the parameterizations to an implicit representation. This is always pos-

sible for a rational parameterization. For the implicit/implicit case, we would use algebraic geometry to convert one of the implicit representations to a parametric one, although this is **not** always possible. See Sections 10.9 and 10.15 in [AgoM05].

Another comment we would like to make here is with regard to the Newton-Raphson method. This method, whose mathematics is described in more detail in Section 4.7 in [AgoM05], will be referred to over and over again because it gets used in many of the algorithms described in this chapter. Therefore, to ensure that the reader knows what is involved when it is mentioned, it is probably helpful to review the idea behind it in general terms before we get started. The Newton-Raphson method is an approach to solving an equation of the form

$$\mathrm{f}(\mathbf{x}) = \mathbf{0}. \tag{13.4}$$

One starts with an initial guess $\mathbf{x}_0$ and then generates a sequence of points $\mathbf{x}_i$ that, hopefully, converge to a solution $\mathbf{p}_0$ of equation (13.4). If $\mathbf{S}$ is the set defined by equation (13.4), then this process is often referred to as *relaxing* $\boldsymbol{x}_0$ *to* $\boldsymbol{p}_0$ *in* $\mathbf{S}$. Given $\mathbf{x}_i$, one gets the next point $\mathbf{x}_{i+1}$ by using the first few terms of the Taylor expansion for f about $\mathbf{x}_i$ as an approximation for f and solving for its zeros. Strictly speaking, the Newton-Raphson method uses the linear approximation

$$\mathrm{h}(\mathbf{x}) = \mathrm{f}(\mathbf{x}_i) + \mathrm{Df}(\mathbf{x}_i)(\mathbf{x} - \mathbf{x}_i),$$

which uses the first two terms. The solution to the linear system $\mathrm{h}(\mathbf{x}) = \mathbf{0}$ then becomes the next point $\mathbf{x}_{i+1}$. Using the Moore-Penrose inverse matrix to solve for $\mathbf{x}_{i+1}$ we get

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathrm{f}(\mathbf{x}_i)\left(\mathrm{f}'^{\mathrm{T}}(\mathbf{x}_i)\left(\mathrm{f}'(\mathbf{x}_i)\mathrm{f}'^{\mathrm{T}}(\mathbf{x}_i)\right)^{-1}\right). \tag{13.5}$$

where $\mathrm{f}'(\mathbf{z})$ is the Jacobian matrix of f at $\mathbf{z}$. The solution $\mathbf{p}_0$ may only be one point of a curve segment that belongs to the entire solution set $\mathbf{S}$ of equation (13.4). In that case, one generates a sequence of solutions $\mathbf{p}_i$ that become the vertices of a polygonal curve that approximates a curve segment of $\mathbf{S}$. One gets $\mathbf{p}_{i+1}$ from $\mathbf{p}_i$ by starting with a guess $\mathbf{x}_0 = \mathbf{p}_i + \boldsymbol{\delta}$ for some appropriate small $\boldsymbol{\delta}$ and relaxing that point to $\mathbf{S}$ in the manner described above. If one is careful, then one can use this technique to generate an approximation to the entire solution set $\mathbf{S}$, but one needs a starting guess for every component of $\mathbf{S}$.

A major problem for algorithms that compute intersections is that one has to cope with potentially complicated intersections and situations where objects intersect in singular ways. For example, two surfaces may intersect in a surface, in isolated points, or in a curve that has cusps, self-intersections, and consists of several disconnected pieces. The nicest situation is where objects meet transversally, but even there problems arise in practice if points are too close to a singularity, such as where the objects are close to intersecting or almost tangent. As we indicated in our comments at the beginning of this section, there is a constant worry about numerical instability.

Another issue that comes up with surface intersections is the question of how the intersection should be represented. Some ways to represent the intersection curve are

(1) exactly via equations or parameterizations (for example, this is possible for quadric intersections),
(2) as a polygonal curve that matches the intersection to some tolerance,
(3) as a spline (usually of a low degree) that uses an appropriate number of points on the intersection as control points, or
(4) procedurally (for example, determine a fixed number of points on the intersection and compute the intersection to the desired accuracy on demand by a marching method).

The polygonal curve approach seems to be the most popular.

13.2 Convex Set Intersections

This section describes a simple algorithm for testing whether two convex linear polyhedra intersect. The result is based on the fact, proved in Section 17.5, that disjoint convex sets can be separated by a hyperplane and that there is a straightforward algorithm for determining whether such a hyperplane exists or not.

See [Edel87] for a linear programming approach to the intersection and separating hyperplane problem. The solutions there result in much more complex programs and may not be that much faster if m is relatively small.

Algorithm 13.2.1 is an abstract program for a function AreDisjoint that determines whether two convex hulls are disjoint. First, a function SpaceProjectionType is called that implements Theorem 17.5.1 directly and determines if the sets are linearly separable. If the sets are separated by a plane **P**, then this still allows the possibility that they intersect in that plane. Therefore, Theorem 17.5.1 has to be applied again to the intersections of the sets with this plane. This is accomplished by the function PlanarProjectionType. If two sets are separated in a plane this allows one final possibility that they may intersect on the separating line and so we check for that with a final call to function LinearProjectionType.

13.2.1 Theorem. Algorithm 13.2.1 for whether the convex hulls of two sets **X** and **Y** with s and t points, respectively, are disjoint is an $O(m^4)$ algorithm, where $m = s + t$.

Proof. The algorithm is implemented by the function AreDisjoint. The work done in AreDisjoint is dominated by the call to SpaceProjectionType. The loop in this function is executed $m(m - 1)(m - 2)$ times and involves $O(m)$ operations.

Finally, it should be noted that it is possible to make the algorithm more efficient by decreasing the size of the loop in the SpaceProjectionType procedure. For example, for each point picked in the outer i loop of the procedure one only needs to look at vertices that lie in the "silhouette" of the polygons as seen from that point. Of course, any gain in efficiency of the algorithms is at the expense of much more complicated code.

```
P(p,n) denotes the plane through p with normal vector n.
Given a vector n, n(X) denote the set {n•x | xε X}

boolean function AreDisjoint (point set X, Y)
{ The function returns true if conv(X) and conv(Y) are disjoint and false otherwise.
  We assume that the affine hull of the points in X and Y has dimension 3. }
begin
    integer   itp;
    point     p, q;   vector n1, n2;
    point set X1, Y1;
    real      c;

    itp := SpaceProjectionType (X,Y,p,n1);
    case  itp  of
        0  : return (false);                    { Sets intersect }
        1  : return (true);                     { Sets are disjoint }
        2  : begin
                 X1 := X ∩ P(p,n1);   Y1 := Y ∩ P(p,n1);
                 itp := PlanarProjectionType (X1,Y1,p,n1,q,n2);
                 case  itp  of
                     0  : return (false);       { Sets intersect }
                     1  : return (true);        { Sets are disjoint }
                     2  : begin
                              X1 := X1 ∩ P(q,n2);   Y1 := Y1∩ P(q,n2);
                              itp := LinearProjectionType (X1,Y1,q,n1,n2);
                              return (itp = 0);
                          end
                 end
             end
    end
end;

integer function SpaceProjectionType (point set X, Y; ref point p; ref vector n)
{  Inputs:    subsets  X = (p1,p2,…,ps)  and  Y = (q1,q2,…,qt)  of R^3
   Output:    point p and normal vector for a plane P
              function returns an integer:
                  0  - convex hulls of X and Y intersect
                  1  - convex hulls of X and Y are disjoint
                  2  - conv(X) ∩ conv(Y)  lies in P(p,n)          }
begin
    point set S;
    integer   m, itp, i, j, k;

    S   := X ∪ Y;          { S = (p1,p2,…,ps,q1,q2,…,qt) = (s1,s2,…,sm) }
    m  := s + t;    itp := 0;
```

Algorithm 13.2.1. Algorithm for when convex sets are disjoint.

```
    for  i:=1 to m  do
        for  j:=i+1 to m  do
            for k:=j+1 to m  do
                if  si,sj,sk  are linearly independent points  then
                    begin
                        n := (sj − si) × (sk − si); { normal to plane P }
                        if   n(X) is disjoint from n(Y)
                            then
                                begin
                                    itp := 1;      Break;
                                end
                            else  if n(X) meets n(Y) is a single number
                                then
                                    begin
                                        itp := 2;       p := si; Break;
                                    end
                    end;
    return (itp);
end;

integer function PlanarProjectionType (point set X, Y;point p;vector n1;
                                        ref point q;ref vector n2)
{ Inputs:  X = (p1,p2,...,ps) ,  Y = (q1,q2,...,qt)  ⊆  plane P = P(p,n1)
           Convex hull of points in X and Y is assumed to have dimension 2.
  Output:  point q and normal vector n2 for a plane P
           function returns an integer:
             0  - convex hulls of X and Y intersect
             1  - convex hulls of X and Y are disjoint
             2  - conv(X) ∩ conv(Y)  lies on line  L = P(q,n1) ∩ P(q,n2)  }
begin
    point set S;
    integer   m, itp, i, j;

    S   := X ∪ Y;          { S = (p1,p2,...,ps,q1,q2,...,qt) = (s1,s2,...,sm)  }
    m   := s + t;
    itp := 0;
    for  i:=1 to m  do
        for  j:=i+1 to m  do
            if   si,sj are distinct points   then
                begin
                    n2 := (sj − si) × n1;        { normal to the line L in the plane P }
                    if   n2(X) is disjoint from n2(Y)
                        then
                            begin
                                itp := 1;   Break;
                            end
```

Algorithm 13.2.1. *Continued*

```
                    else if  n2(X) meets n2(Y) is a single number
                        then
                            begin
                                itp := 2;    q := si;        Break;
                            end
                end;
    return (itp);
end;

integer function LinearProjectionType (point set X, Y; point p; vector n1, n2)
{ Inputs:    X = (p1,p2,...,ps) ,  Y = (q1,q2,...,qt) ⊆ line  L = P(p,n1) ∩ P(p,n2)
 Output:     function returns an integer:
                 0  - convex hulls of X and Y intersect
                 1  - convex hulls of X and Y are disjoint      }
begin
    vector v;
    real    a, b, c, d;

    v := a direction vector for line L;
    a := min( v(X) );    b := max( v(X) );    c := min( v(Y) );    d := max( v(Y) );
    if   [a,b] ∩ [c,d] = ϕ  then  return (0)
                            else  return (1);
end;
```

Algorithm 13.2.1. *Continued*

13.3 Curve Intersections

13.3.1 Ray-Curve Intersection

We assume that we are dealing with planar curves. Intersecting a ray with a polygonal curve reduces to intersecting a ray with a segment. This is a problem that the reader was asked to solve in Exercise 6.5.2. That leaves the interesting case of smooth curves.

A Newton-Raphson Method. Suppose that one wants to find the intersection of a ray **X** from a point **p** in a direction **v** with a smooth curve parameterized by a function p(u). Let **L** be the **line** through **p** with unit direction vector **v**. One can use a rigid motion to transform this situation into one where **p** is the origin and **v** is $\mathbf{e}_2$, so that **L** is the y-axis. To simplify our discussion, assume that this has been done. Note that there is typically no real cost associated to this assumption since modeling systems usually define objects relative to associated coordinate systems. This means that to

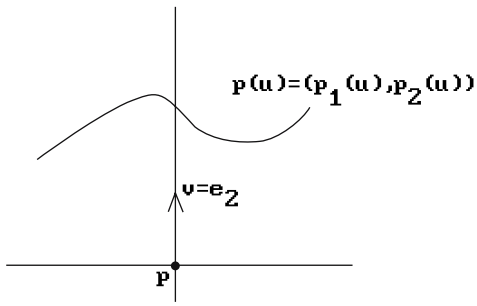


Figure 13.1. A ray-curve intersection problem.

find the world coordinates of their points involves a transformation in any case. Adding a rigid motion costs only one matrix multiplication per **object** (not one per point).

If $p(u) = (p_1(u),p_2(u))$, then finding the point where the y-axis pierces the curve is equivalent to solving the equation

$$p_1(u) = 0$$

along with the constraint $p_2(u) \geq 0$. See Figure 13.1. The equation can be solved using a Newton-Raphson method. To apply this method we need a starting guess at a solution. One way to get one is to subdivide the domain of p(u) into small segments with endpoints u_i and then approximate the curve by the polygonal curve defined by the points $p(u_i)$. If we find a segment pierced by the ray, then any one of its endpoints can be used as an initial guess for the Newton-Raphson iteration. Two situations that cause problems with this approach are zero derivatives and places where the iteration generates parameter points that leave the interval which is the domain of the function p(u).

Recursive Subdivision Methods. The first approach of this type, described in [LanR80], can be used to find ray-curve or line-curve intersections in the case of curves like Bézier and B-spline curves, which satisfy the variation diminishing property, namely, that a hyperplane meets the curve in no more points than it meets the control polygon of the curve. Because of that property one can use the following algorithm to find the intersection of a line and the curve p(u):

(1) If the line does not intersect the control polygon, then the line will not intersect the curve and we quit.
(2) Otherwise, subdivide the curve into two parts, compute the control polygon for these two subcurves, and repeat step (1) for each of them. One continues this process until the curves are small enough so that we can use an endpoint as an approximation to the intersection.

A better algorithm replaces (1) by

(1) If the line does not intersect a bounding box containing the control polygon, then the line will not intersect the curve and we quit.

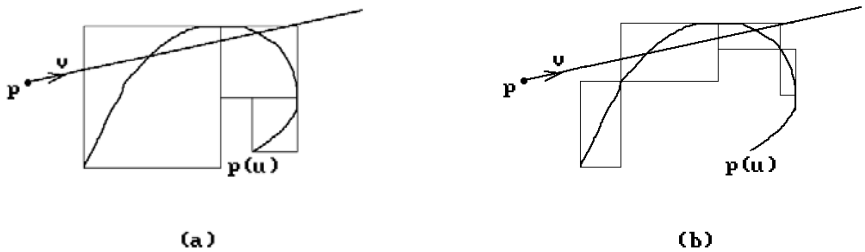


Figure 13.2. Bounding rectangles for curve segments without characteristic points.

The easiest way to get a bounding box is to use the minimum and maximum values of the coordinates of the control points. It is faster to compute the intersection of a line with such a bounding box than with the control polygon. The subdivision approach generalizes to nonplanar curves.

Another recursive subdivision algorithm is a "characteristic point"–based subdivision algorithm described in [KopM83]. A characteristic point is basically one where the first or second derivative of a component function vanishes. Such points were computed using interval analysis for accuracy. We describe the planar case. Given a curve, one determines all the points that have horizontal or vertical tangent lines. These points, along with the endpoints of the curve, divide the curve into segments. The rectangle that has diagonal defined by the two endpoints of a curve segment then contains that segment. See Figure 13.2. The collection of these rectangles becomes the bounding boxes for the curve. If our ray intersects any of these rectangles, then we divide the corresponding curve segment in half and compute the new bounding rectangles. We keep doing this until the remaining bounding rectangles are small enough. The only disadvantage of this algorithm is that one has to find the places on the curve with horizontal and vertical tangents. For low-degree curves it is shown in [SedP86] that the method is actually faster than the Bézier subdivision approach, although the latter is more stable.

13.3.2 Curve-Curve Intersections

Finding the intersection of two polygonal curves reduces to finding the intersection of a segment with a curve and finally to finding the intersection of two segments. The last problem has already been dealt with in Section 6.5.

To find the intersection of a parameterized and a polygonal curve reduces to finding the intersection of a segment with a smooth curve. We can use a straightforward modification of the ray-curve algorithm described in Section 13.3.1.

To intersect two smooth curves one could of course replace those curves by polygonal approximations and a use polygonal intersection algorithm. The main problem is determining whether the approximations are good enough so that no intersections are missed. In a variation of the subdivision approach described in [LanR80], Turner ([Turn88]) actually replaced the curve with two approximations that contained the curve between them. As an example, consider the planar Bézier curve shown in Figure 13.3. The convex hull of the four control points is bounded by two curves $\mathbf{C}_1$ and $\mathbf{C}_2$,

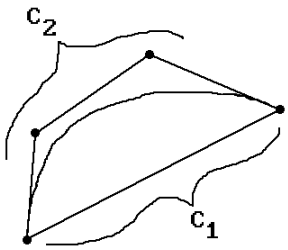


Figure 13.3. Bounding curves for a Bézier curve.

one on either side of the actual curve. Turner called the region between the two curves a *curve-bounding area*. This idea easily generalizes to any curves defined by control points. In general, one can define two bounding curves associated to a curve, which have the property that the region between them contains the original curve. If the curve-bounding areas of two curves do not intersect, then the curves will not intersect. If they intersect, then a subdivision process is applied to get a new set of two curves so that the region between them is a closer fit to the original curve. Turner describes steps that one can take to make sure that the intersection algorithm based on this idea provides correct answers in the presence of singularities.

The next three sections discuss more direct approaches to the smooth curve intersection problem. For an approach based on interval arithmetic see [HMSP96].

13.3.3 A Curve Newton-Raphson Method

To find the intersection of two smooth curves parameterized by functions p(u) and q(u) amounts to solving an equation of the form

$$f(u, v) = p(u) - q(v) = 0. \tag{13.6}$$

We describe an approach discussed in [HosL93]. Let $p(u) = (p_1(u),p_2(u))$ and $q(v) = (q_1(v),q_2(v))$.

(1) Pick a point $\mathbf{p}_0 = (x_0,y_0) = p(u_0)$ on the curve p(u).
(2) Find the intersection of the tangent line $\mathbf{L}_0$ to p(u) at $\mathbf{p}_0$ with the curve q(u). Since this tangent line is defined by $\mathbf{p}_0 + t\, p'(u_0)$ and has equation

$$(x - x_0)p_2'(u_0) - (y - y_0)p_1'(u_0) = 0,$$

the equation we need to solve is

$$(q_1(v) - x_0)p_2'(u_0) - (q_2(v) - y_0)p_1'(u_0) = 0,$$

or

$$q_1(v)p_2'(u_0) - q_2(v)p_1'(u_0) - (x_0p_2'(u_0) - y_0p_1'(u_0)) = 0. \tag{13.7}$$

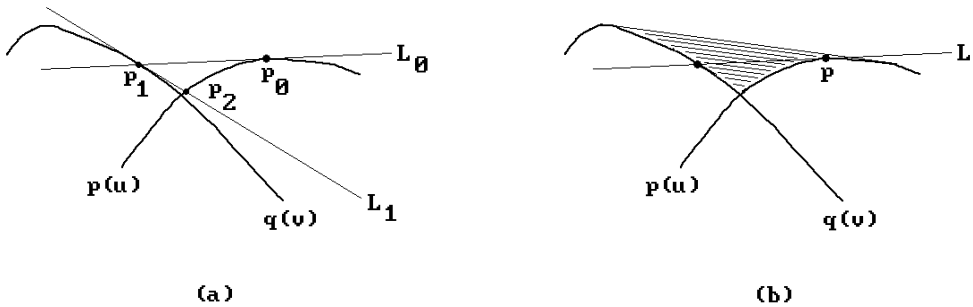


Figure 13.4. Curve intersections with the Newton-Raphson method.

Let v_0 be a solution. Now find the intersection of the tangent line $\mathbf{L}_1$ to q(v) at $\mathbf{p}_1 = q(v_0)$ with the curve p(u). Let this point be $\mathbf{p}_2 = p(u_1)$.

(3) Continue this process, getting sequences u_i, v_i, and $\mathbf{p}_i$. Stop when $|\mathbf{p}_i - \mathbf{p}_{i+1}|$ is sufficiently small.

See Figure 13.4(a). The method works as long as the tangents keep intersecting the curves. In Figure 13.4(b) this corresponds to starting at points along the boundary of the shaded region.

Numerical techniques for tracing curves work pretty well if one does not run into any singularities. The problem at singularities is that the approximation that one is using for the curve fails to be sufficiently accurate at such points.

13.3.4 Curve Recursive Subdivision Methods

We can extend the line-curve recursive subdivision (divide-and-conquer) approaches described earlier. The only difference is that we keep subdividing both curves and checking for intersections of bounding boxes. For curves that satisfy the variation diminishing property we get the algorithm from [LanR80]. The algorithm in [KopM83] applies to arbitrary curves.

The paper [SedN90] describes a better algorithm for planar Bézier curves. The authors refer to their approach as *Bézier clipping*. It is also a bounding box type approach.

Definition. A *fat line* is the region between two parallel lines.

Note that a fat line can be thought of as special case of the slabs defined in Section 6.2. The reader should compare the computations we will make for fat lines with those we made for slabs. Some other related concepts are the "fat arcs" in [SeWZ89], used as a criterion for curve flatness, and what could be called "fat planes", used in [Carl82] for an intersection algorithm for Bézier surfaces.

The first thing we want to do is associate a fat line to an arbitrary Bézier curve p(u). Let $\mathbf{p}_i = (x_i, y_i)$, i = 0,1, . . . n, be the control points of p(u). See Figure 13.5. Let $\mathbf{L}$ be the line through $\mathbf{p}_0$ and $\mathbf{p}_n$. Represent the line by an equation of the form

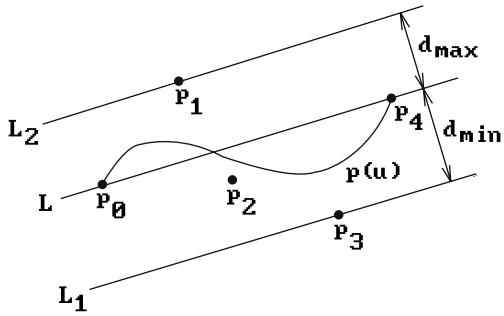


Figure 13.5. The fat line for a Bézier curve.

$$ax + by + c = 0, \quad a^2 + b^2 = 1, \tag{13.8}$$

and define

$$d(x,y) = ax + by + c. \tag{13.9}$$

It is easy to see that d(x,y) is just the signed distance from a point (x,y) to **L** because **n** = (a,b) is a unit normal vector for **L** and $d(x,y) = ((x,y) - \mathbf{p}_0) \bullet n$. Let $d_i = d(x_i,y_i)$ and set

$$d_{min} = \min\{d_0, d_1, \ldots, d_n\},$$
$$d_{max} = \max\{d_0, d_1, \ldots, d_n\}.$$

The convex hull property of Bézier curves implies that the curve is contained in the fat line defined by the two lines $\mathbf{L}_1$ and $\mathbf{L}_2$, which are parallel to **L** and a distance of d_{min} and d_{max} from **L**, respectively. Actually, as the example in Figure 13.5 indicates, we can do better. The curve is contained in a thinner fat line and the lines $\mathbf{L}_1$ and $\mathbf{L}_2$ can be moved closer to **L**. The thinnest fat line parallel to **L** is really determined by the minimum and maximum of d(u), the signed distance from p(u) to **L**. The polynomial d(u) can be explicitly computed in the quadratic and cubic curve case and one can use the following improved values for d_{min} and d_{max}:

Quadratic Curve. $d_{min} = \min\left\{0, \frac{d_1}{2}\right\}$ and $d_{max} = \max\left\{0, \frac{d_1}{2}\right\}$

Cubic Curve. Although precise values can be computed in this case also, the formulas are complicated and the extra computational effort is not justified. Instead the following approximations are suggested in [SedN90]:

If $d_1 d_2 > 0$, then let

$$d_{min} = \frac{3}{4}\min\{0, d_1, d_2\} \quad \text{and} \quad d_{max} = \frac{3}{4}\max\{0, d_1, d_2\}.$$

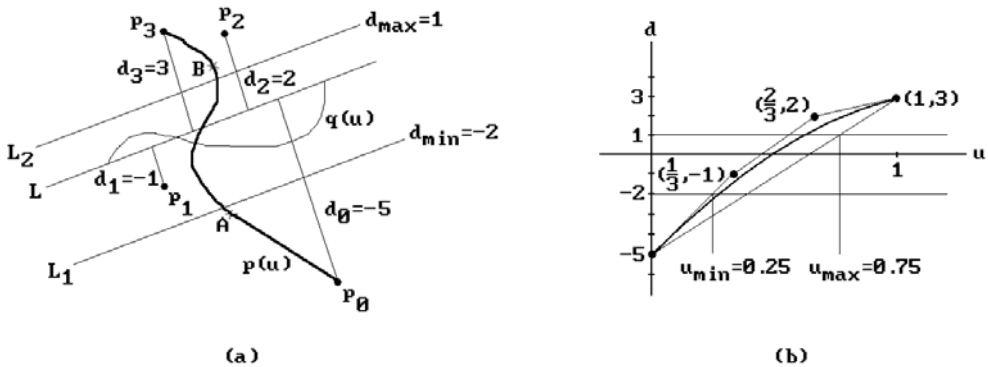


Figure 13.6. Sederberg-Nishita Bézier clipping.

If $d_1 d_2 \le 0$, then let

$$d_{min} = \frac{4}{9}\min\{0, d_1, d_2\} \quad \text{and} \quad d_{max} = \frac{4}{9}\max\{0, d_1, d_2\}.$$

After these preliminaries, we are ready to describe the Bézier clipping algorithm. Let p(u) and q(u) be the two Bézier curves with domain [0,1] whose intersection we want to determine. Let **X** be a fat line that contains the curve q(u). See Figure 13.6(a). In that example, **X** is the region between the parallel lines $\mathbf{L}_1$ and $\mathbf{L}_2$. We want to determine the values of u for which p(u) lies outside of **X**. Express p(u) in the form

$$p(u) = \sum_{i=0}^{n} B_{i,n}(u)\,\mathbf{p}_i, \tag{13.10}$$

where $B_{i,n}$ (u) are the Bernstein polynomials and $\mathbf{p}_i = (x_i, y_i)$ are the control points of p(u). Let equation (13.8) be the equation of the line through $\mathbf{p}_0$ and $\mathbf{p}_n$ and let d(u) be the signed distance from p(u) to **L**. It is easy to show that

$$d(u) = \sum_{i=0}^{n} B_{i,n}(u)\,d_i, \quad d_i = ax_i + by_i + c.$$

Finally, define the parametric curve

$$\begin{aligned} D(u) &= \sum_{i=0}^{n} B_{i,n}(u)\mathbf{D}_i \\ &= (u, d(u)), \end{aligned}$$

where $\mathbf{D}_i = (t_i, d_i)$ and $t_i = i/n$. The fact that the x-coordinate of D(u) is u follows from the identities

$$\sum_{i=0}^{n} B_{i,n}(u)(i/n) = u[(1-u)+u]^n = u.$$

Figure 13.6(b) shows the graph of D(u) for the example in Figure 13.6(a). Clearly, if

$$D(u) \le d_{min} \text{ or } D(u) \ge d_{max},$$

then p(u) will not lie on the curve q(u). Therefore one can trim away those parts of p(u). Actually, we shall only trim away parts at the ends of the curve. Let u_{min} and u_{max} be the largest and smallest u so that the convex hull of the $\mathbf{D}_i$ misses the fat line **Y** between $y = d_{min}$ and $y = d_{max}$ over the intervals $[0, u_{min}]$ and $[u_{max},1]$, respectively. For the example in Figure 13.6(b) it turned out that $u_{min} = 0.25$ and $u_{max} = 0.75$. We now subdivide the Bézier curve p(u) at u_{min} and u_{max} using the de Casteljau algorithm and only keep the part over $[u_{min},u_{max}]$. That part is then reparameterized to [0,1]. Note that we are being conservative here and we could have trimmed a little more. As Figure 13.6(b) shows, the curve D(u) meets the fat line **Y** over an interval smaller than $[u_{min},u_{max}]$.

After clipping p(u) against q(u), we repeat this process, this time clipping q(u) against the fat line associated to the clipped p(u). We alternate back and forth between clipping against p(u) and q(u) until we arrive at our intersection point. This is the basic idea behind the Bézier clipping algorithm for finding curve intersections, but there are some important details that need to be added.

The first observation is that we can use any fat line for the clipping. The choice above is usually effective, but another choice that is sometimes better is to use a fat line orthogonal to the line containing $\mathbf{p}_0$ and $\mathbf{p}_n$. An example of that is shown in Figure 13.7. The fat line between $\mathbf{L}_1$ and $\mathbf{L}_2$ is a better choice when clipping q(u) against p(u) than using one that is parallel to the line containing the endpoints **A** and **B** of the curve p(u). The extra time used in checking which choice is better seems to be worth it.

Another point that has not been addressed is whether or not the above algorithm actually converges. In fact, it may not if we only clip little if anything at each stage.

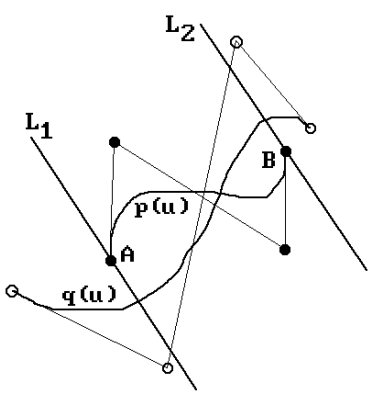


Figure 13.7. An alternative fat line.

Figure 13.8. Multiple-curve intersections.

Figure 13.8 shows an example of multiple intersections, which lead to such situations. Therefore, the suggested solution for the problem is the following recursive subdivision step: If Bézier clipping does not reduce the parameter range of each of the curves by at least 20%, then subdivide the curve with the largest remaining parameter interval and do a Bézier clip of the other curve with the two halves. Although this solves our convergence problem, if curves are almost tangent or two intersections are very close, then the algorithm reduces to a divide-and-conquer binary search type algorithm in the neighborhood of those points. There is a very efficient way to handle that situation, but we refer the reader to [SedN90] for the details. This finishes our discussion of the Sederberg-Nishita Bézier clipping algorithm. Timing comparisons in [SedN90] show that for curves of degree less than five, the implicitization algorithm in [SedP86] runs typically between 1 and 3 times faster than the Bézier clip algorithm, which in turn is between 2 and 10 times faster than the algorithms from [LanR80] and [KopM83]. For higher-degree curves, the Bézier clipping algorithm seems to win. [Rock90] also has a comparison of various divide-and-conquer methods for curves.

13.3.5 Curve Algebraic Methods

[SedP86] describes an algorithm for computing the intersection of two rational curves. By implicitizing one of the curves and substituting the parameterization of the other into the equation for the first one ends up having to solve a set of homogeneous linear equations. Another approach combining elimination theory and matrix computations is described in [ManD94] and [ManK97]. An algorithm for finding the intersection of two algebraic curves is described in [Sede89].

In general, depending on how smooth curves are presented, earlier comments imply that the intersection problem can be reduced to finding the roots of an equation of one or two variables. The interesting case is the two-variable case that involves solving a polynomial equation of the form

$$f(u, v) = 0,$$

like equation (13.6). This is the type of problem that algebraic geometry can help solve and is really part of the more general problem of computing implicitly defined objects about we have more to say in the next chapter.

Sometimes one wants not only the points in the plane that constitute the intersection of two parameterized curves but the parameter values at the points of intersection. If one used an implicitization approach to find the intersection, then the parameter variables would have disappeared. This means that after finding an

intersection point **q**, one is left with another problem, namely, that of finding the parameter u so that p(u) = **q**.

13.4 Special Surface Intersections

13.4.1 Ray-Surface Intersections

Problem 10.2.2.6 solved the ray-surface intersection problem for the faceted surface case. Now assume that **S** is a parameterized surface with parameterization p(u,v) and that **X** is a ray from a point **p** in a direction **v**. Let **L** be the **line** through **p** with direction vector **v**. Using a rigid motion we can transform this situation into one where **p** is the origin and **v** is $\mathbf{e}_3$, so that **L** is the z-axis. Like in the ray-curve case, there is little cost associated to assuming that this has been done.

If $p(u,v) = (p_1(u,v),p_2(u,v),p_3(u,v))$, then finding the point where the z-axis pierces **S** is equivalent to solving the equations

$$p_1(u,v) = 0$$
$$p_2(u,v) = 0$$

along with the condition that $p_3(u,v) \geq 0$. These equations can be solved using a Newton-Raphson method but one needs a starting guess at a solution. One way to get one is to subdivide the u-v domain into small rectangles with corners (u_i,v_j) and then approximate the surface by the grid of points $p(u_i,v_j)$. If we find a quadrilateral pierced by the ray, then any one of its corners can be used as an initial guess for the Newton-Raphson iteration.

Again, like in all Newton-Raphson method approaches, one has to be ready to deal with at least two problems. The first is that the partial derivatives of the functions $p_1(u,v)$ and $p_2(u,v)$ may not be linearly independent at some points causing the usual problems with a Newton-Raphson method. The other more complicated problem is caused by the fact that we have a bounded domain and the iteration may want to cross one of the boundary lines. When this happens one should check if the solution does not in fact lie on the boundary by applying a Newton-Raphson iteration **along** the boundary, which is a one-dimensional problem. There will be additional problems at the corners of the patch. If at any time the solution seems to want to move back inside the patch, revert to the two-dimensional search problem.

13.4.2 Curve-Surface Intersections

We only consider the case where both the curve and surface are smooth subsets of $\mathbf{R}^3$ and are defined by parameterizations. Assuming that the curve is parameterized by q(t) and the surface by p(u,v), we need to solve the equation

$$r(t,u,v) = q(t) - p(u,v) = \mathbf{0}. \tag{13.11}$$

We describe a Newton-Raphson approach. The idea is to start with a guess (t_0,u_0,v_0) for a solution to equation (13.11) and then define a sequence (t_i,u_i,v_i) that converges to an actual solution. Assume that we have already defined $\mathbf{r}_i = (t_i,u_i,v_i)$, $i \geq 0$. Using the linear function

$$h(t,u,v) = r(t_i,u_i,v_i) + Dr(t_i,u_i,v_i)(t-t_i,u-u_i,v-v_i)$$

as an approximation to the function r(t,u,v) in a neighborhood of (t_i,u_i,v_i), we get the next iterate $(t_{i+1},u_{i+1},v_{i+1})$ by solving $h(t,u,v) = \mathbf{0}$. Let $\mathbf{q}_i^t = q'(t_i)$, $\mathbf{p}_i^u = p_u(u_i,v_i)$, and $\mathbf{p}_i^v = p_v(u_i,v_i)$. It is easy to show that

$$h(t,u,v) = \mathbf{r}_i + \mathbf{q}_i^t(t-t_i) - \mathbf{p}_i^u(u-u_i) - \mathbf{p}_i^v(v-v_i),$$

so that we need to solve

$$\mathbf{r}_i + \mathbf{q}_i^t(t-t_i) - \mathbf{p}_i^u(u-u_i) - \mathbf{p}_i^v(v-v_i) = \mathbf{0}. \tag{13.12}$$

Since, $\mathbf{p}_i^u \times \mathbf{p}_i^u = \mathbf{0}$,

$$\mathbf{p}_i^u \times \mathbf{r}_i + \mathbf{p}_i^u \times \mathbf{q}_i^t(t-t_i) - \mathbf{p}_i^u \times \mathbf{p}_i^v(v-v_i) = \mathbf{0}.$$

But $\mathbf{q}^t{}_i$ is orthogonal to $\mathbf{p}_i^u \times \mathbf{q}_i^t$ and so $\mathbf{q}_i^t \bullet (\mathbf{p}_i^u \times \mathbf{q}_i^t) = 0$ and

$$\mathbf{q}_i^t \bullet \left(\mathbf{p}_i^u \times \mathbf{r}_i\right) - \mathbf{q}_i^t \bullet \left(\mathbf{p}_i^u \times \mathbf{p}_i^v\right)(v-v_i) = \mathbf{0}. \tag{13.13}$$

Finally, let $\mathbf{n}_i = \mathbf{p}_i^u \times \mathbf{p}_i^v$ be the normal to the surface at $r(t_i,u_i,v_i)$ and set $D_i = \mathbf{q}^t{}_i \bullet \mathbf{n}_i$. Then equation (13.13) can be solved for v. A similar argument can be applied to the variables u and t. What we finally get are equations

$$\begin{aligned} t_{i+1} &= t_i - \left(\mathbf{p}_i^u \bullet \left(\mathbf{p}_i^v \times \mathbf{r}_i\right)\right)/D_i \\ u_{i+1} &= u_i - \left(\mathbf{p}_i^v \bullet \left(\mathbf{q}_i^t \times \mathbf{r}_i\right)\right)/D_i \\ v_{i+1} &= v_i + \left(\mathbf{q}_i^t \bullet \left(\mathbf{p}_i^u \times \mathbf{r}_i\right)\right)/D_i\,. \end{aligned} \tag{13.14}$$

The sequence (t_i,u_i,v_i) defined by equations (13.14) will converge to a point in the intersection of our curve and surface provided that we do not run into the usual problems associated to Newton-Raphson methods. Getting all the intersection points hinges on being able to come up with enough initial guesses.

13.4.3 Surface Sections

We begin our look at surface-surface intersections with the special case of finding a section of a surface **S**. This includes the problem of finding contours, although we will say more about that in Section 14.6.

Definition. The intersection of a set in $\mathbf{R}^n$ and a hyperplane is called a *section* of that set. If the hyperplane is parallel to a coordinate plane, that is, if it is defined by an equation of the form $x_i = c$, where c is constant, then the section is called a *contour*.

We first consider the case where **S** is faceted. Assume that **S** has convex facets. To find the section of **S** with respect to some plane there is again no loss of generality by assuming that the plane is the x-y plane. The idea will be to find all the edges of **S** that cross the x-y plane and then to connect the intersection points appropriately. To find the crossing edges, we only need to look at the z-coordinates of the endpoints of the edges and see if they straddle 0. To facilitate connecting the edge intersections we use the fact that if the section crosses a face then it crosses two of its edges (unless it just touches the face in an edge or vertex). Therefore, we associate a pair of integers to each face which will either both be –1 or the index of an edge in the edge array. The resulting algorithm is shown in Algorithm 13.4.3.1.

The problem of finding sections of smooth surfaces is more complicated. We have actually run into this problem earlier in the book. In fact, some of the techniques described in Section 7.10 that provided a scan line visible surface determination algorithm for smooth surfaces would also apply to solve this problem. Here we describe another approach. See [PraG86] or [Sutc80].

The Grid or Lattice Evaluation Method. First of all, we may again transform the intersection problem by a rigid motion to the case where the plane is parallel or in fact equal to the x-y plane. The problem then is equivalent to finding a contour on a surface. If the surface is parameterized by a function

$$p(u,v) = (x(u,v), y(u,v), z(u,v)),$$

then the problem is to solve the equation $z(u,v) = 0$. The basic idea is to reduce the problem to a lower-dimensional one by adding some constraints. Typically, one evaluates $z(u,v)$ at a grid of points (u_i,v_j). Let $z_{ij} = z(u_i,v_j)$. If adjacent z_{ij}'s have opposite signs, then the contour will cross the corresponding edge between the grid points. One finds the crossing point by solving a one-dimensional problem of the form $z(u_i,v) = 0$ or $z(u,v_j) = 0$. The intersection points are then connected to get a polygonal approximation of the contour. See Figure 13.9(a). One major problem with this approach is that it can be difficult to determine how to connect intersection points on the grid

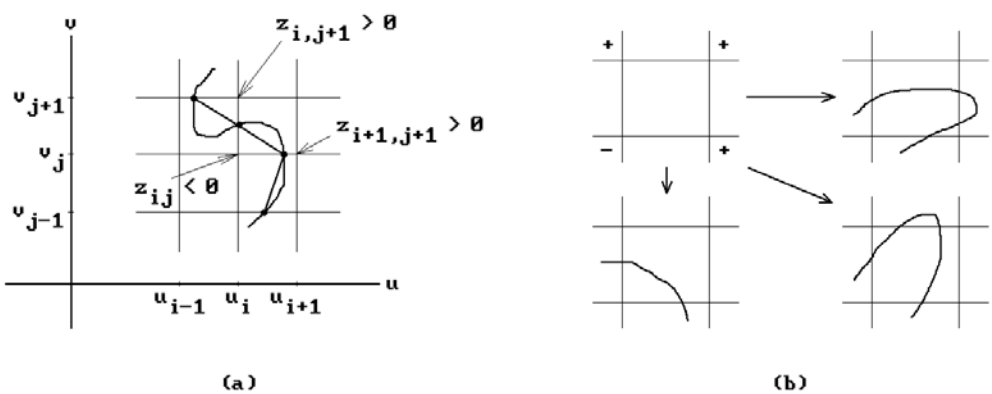


Figure 13.9. Sections using a lattice evaluation method and ambiguities.

```
Input:     A faceted surface S
Output:    A list of pseudofacets F that constitute the section of S by the x-y plane
           (A pseudofacet is a list of vertices that define a facet that may be listed
           in a random order)

{ Global variables }
integer list array indexed by the faces of S Exs;
edge list     E;
edge          e, e';
real          z0, z1;
vertex        v0, v1;
pseudofacet   fc;

begin
  for each  face f in S  do  initialize Exs[f] to the empty list;
  Initialize E to empty;
  for each  edge e in S  do
    begin
      z0 := z-coordinate of first vertex v0 of e;
      z1 := z-coordinate of second vertex v1 of e;
      if  0 ∈ [z0,z1]  then
        case  type of intersection  of
            z0 = 0, z1 ≠ 0 : for each  edge e' incident to v0  do
                                begin
                                  Add e' to E;
                                  for each  face f incident to e'  do  Add e' to Exs[f];
                                end;
            z1 = 0, z0 ≠ 0 : for each  edge e' incident to v1  do
                                begin
                                  Add e' to E;
                                  for each  face f incident to e'  do  Add e' to Exs[f];
                                end;
            z0 = z1 = 0 :   { horizontal edge case }
                            for each  face f  incident to e  do
                                for each  nonhorizontal edge e' of f adjacent to e  do
                                  begin
                                    Add e' to E;    Add e' to Exs[f];
                                  end;
            0 ∈ (z0,z1) :   begin  { the typical generic case }
                              Add e to E;
                              for each face f incident to e  do  Add e to Exs[f];
                            end
        end;

      Initialize F to empty;
      if  Empty (E)  then  Exit;
```

Algorithm 13.4.3.1. A faceted surface sectioning algorithm.

```
        for  e in E  do
           begin
              fc := PseudoFacetItDefines (e);  { Delete e and possibly other edges from E }
              if  NotNull (fc)  then  Add fc to F;
           end
end;

pseudofacet function PseudoFacetItDefines (edge e0)
begin
    pseudofacet  fc;
    face         f, lastf;
    edge         e;
    boolean      more;

    Delete e0 from E;
    Initialize fc to z, the intersection of e0 with x-y plane;
    for each  face f adjacent to e0  do  flag the e0 in Exs[f] as SEEN;
    (lastf,e) := face lastf is adjacent to e0 and edge e in Exs[lastf] is not yet SEEN;

    more := true;
    while  more  do
        begin
            Delete e from E;
            Add intersection z of e with x-y plane to end of fc;
            for each  face f adjacent to e  do  flag the e in Exs[f] as SEEN;

            if  some face f (≠ lastf) adjacent to e has edge (≠ e) in Exs[f] marked SEEN
                then  more := false
                else
                    begin
                        lastf := face adjacent to e;
                        e is edge in Exs[lastf];
                    end;
        end;

    Delete all references to edges in all Exs[] which were marked as SEEN;

    return (fc);
end;
```

Algorithm 13.4.3.1. *Continued*

lines. A second problem is that some loops may be missed entirely if the grid is not fine enough. This problem often occurs near critical points of z(u,v), such as saddle points, or other singularities because the data may not have a unique interpretation. In Figure 13.9(b) there are three possible intersection curves that give rise to the same labeling of grid vertices with the sign of their z_{ij} values.

[Faro87] describes an algebraic geometry approach to finding sections of bicubic patches. See also [ChaK87]. Another surface section algorithm is a special case of the general surface intersection algorithm described in [GraK97].

The problem of finding sections has a converse, namely to describe a surface given a collection of sections for it. This is **skinning** problem and will be discussed in Section 14.7.

13.5 Surface-Surface Intersections

General surface intersection algorithms are particularly important in geometric modeling, such as in the boundary evaluation algorithm of CSG and NC cutter path generation. Unfortunately, finding such intersections is a very difficult problem and no algorithm that is both efficient **and** correct is known at the moment. Efficient algorithms have problems near singularities or almost-singularities because computers only have finite precision. Accurate and robust algorithms are currently too slow. Some are based on algebraic geometry methods that do not apply to nonalgebraic surfaces. Others try to use interval arithmetic ([HMPY97]). Pratt and Geisow ([PraG86]) give a good survey of some older known intersection algorithms. [Fari92b] contains a list of references on the topic of intersection algorithms.

First of all, finding the intersection of two faceted surfaces reduces to finding the intersection of two facets in 3-space. This in turn reduces to finding the intersection of a convex polygon with a plane and then finding the intersection of a segment with a convex polygon. The mathematics behind doing this was discussed in Section 6.5. If only one of the surfaces is faceted and the other is smooth, one can reduce this problem to finding sections of the smooth surface and finding the intersection of two curves in these sections. This leaves the problem of finding the intersection of two smooth surfaces.

Like in the curve case, one approach to finding intersections of smooth surfaces that one might think of almost immediately is to approximate them by faceted surfaces and then use methods for intersecting those types of surfaces. The usual problem is deciding when an approximation is good enough. [Grif75] and [Grif78] used this approach to display parametric surfaces. [HaAG83] starts off with coarse faceted approximations whose intersecting facets are then subdivided further appropriately. [RosR86] used meshes of isoparametric curves. [Turn88] used a two-surface bounding volume that generalized his curve intersection approach described earlier in Section 13.3.2. After finding points close to the intersection, a Newton-Raphson method was used to actually find points on it.

Rather than solving the smooth surface intersection problem by reducing it to the faceted case it is better to deal with it directly. The special case of quadric surface intersections has been studied extensively. See, for example, [FaNO89] and [Pieg92]

and the references in those papers. The solutions to the general problem involve five basic approaches referred to as lattice evaluation methods, marching methods, homotopy methods, recursive subdivision methods, and methods based on algebraic geometry. When the actual complete surface intersection algorithms that have been proposed are classified as belonging to one of these approaches one must keep in mind that the algorithms usually consist of several stages and different methods can be used for the separate stages. Therefore, many algorithms are really hybrids that do not fit under any single roof as, for example, the ones described in [BarK90], [Kopa91], and [GraK97]. The next five sections will discuss the various approaches.

[MarM91] describes a parameterization for the intersection curve. [GarW89] represents the intersection curve algebraically, basically as an algebraic curve and a birational map between the plane and space curve. Bounding boxes are sometimes used in intersection algorithms and [FiMM86] gives some general bounds for surfaces.

Parallelism has also been used to speed up intersection algorithms. See [BurS93] and [ChBA94].

13.5.1 Surface Lattice Evaluation Methods

A common special case of the lattice evaluation method was already described in Section 13.4.3 in the context of finding contours of the graph of a function of two variables. For more general surface intersection algorithms the method is used at most as a preprocessing step to get some starting points in the intersection. Given two parametric surfaces $p(u,v)$ and $q(u,v)$, one considers the lattice of curves $p(u_i,v)$ and $p(u,v_j)$ defined over grid lines in the rectangular domain of $p(u,v)$ and finds their intersection with the surface $q(u,v)$. The intersections give us starting points to which a marching method can be applied to find the complete intersection of the two surfaces. One might have to refine the lattice at places to make sure that no part of the intersection is missed. We shall see an example of this in the discussion of Timmer's algorithm in the next section. In [BFJP87] polygonal approximations of the curves $p(u_i,v)$ and $p(u,v_j)$ were intersected with a polygonal approximation of $q(u,v)$ and a Newton-Raphson method was then used to find real points on the intersection curves.

13.5.2 Surface Marching Methods

Marching methods, also sometimes referred to as *tracing methods*, are the most widely used methods for computing intersections. They assume that one has found points on the intersection and one then "marches" out from those points along the intersection curve by using information about the local geometry. We have already seen examples of this approach in other contexts. In general, this type of approach has three phases: a **hunting phase**, a **tracing phase**, and a **sorting phase**. In the initial hunting phase one tries to find start values that one then uses in the tracing phase to generate sequences of points that lie on the intersection. One needs enough start values, so that no part of the intersection is missed. Finally, in the sorting phase one separates the sequences of points generated by the tracing phase into sequences that define the connected pieces and loops that are the subcurves of the entire intersection. We end up with a discrete approximation to the intersection.

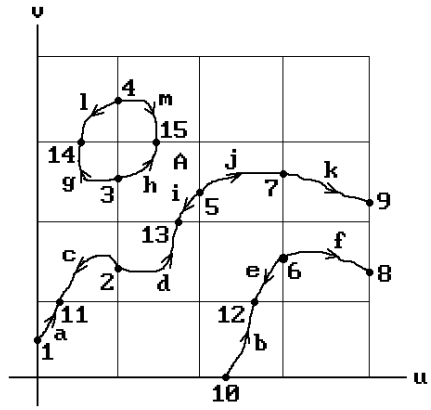


Figure 13.10. Hunting grids for Timmer's algorithm.

An early marching method is one due to Timmer. See [Timm77] or [Mort85]. Suppose that two surfaces $\mathbf{S}_1$ and $\mathbf{S}_2$ have parameterizations p(u,v) and q(u,v), respectively, each with a rectangular domain. To find their intersection **X**, we shall find the subset of the domain of p(u,v) which parameterizes this set. The three solid curves in Figure 13.10 are an example of a set of parameter values for one possible such **X**.

Timmer's Hunting Phase. Subdivide the domain of p into subrectangles. Restricting p to the associated grid lines defines a grid of curves $p(u_i,v)$ and $p(u,v_j)$ on $\mathbf{S}_1$. The intersections of these curves with $\mathbf{S}_2$ will provide the starting points that are used to trace the pieces of **X**. We run into the usual problem of making our grid fine enough, so that we will not miss any part of **X**. In Figure 13.10 the intersection of the grid lines with this set consists of points that have been numbered from 1 to 15. Let uv_i denote the parameter point numbered i. The numbering is based on an ordering of the grid lines and the intersections that lie on them. We started with the vertical grid lines ordered from left to right and ordered their intersections based on increasing v value and then moved on to the horizontal lines ordered from bottom to top and ordered their intersections based on increasing u value. Actually, we do not really have to compute the intersection points precisely initially. They only need to be accurate enough so that a Newton-Raphson method, say, can be used to converge to the actual values. Therefore, the initial guesses for the intersection points could be found by discretizing the curves $p(u_i,v)$ and $p(u,v_j)$ also. Alternatively, we can think of the problem as one of finding points on a curve that are closest to a surface and use any algorithm that is known to solve this problem. Once we have our intersection parameter values uv_i we start the next phase.

Timmer's Tracing Phase. We consider the part of the intersection over each subgrid separately. We analyze each subgrid one after another based on a left to right, bottom to top order. Figure 13.11 shows the subgrid labeled A in Figure 13.10. Since the points uv_i correspond to endpoints of curves in the subgrid, we now use them one at a time based on their ordering to trace the parts of the curves that lie in the subgrid. This not only gives an ordering to the pieces but also a direction. Figure 13.10 also showed the ordering, indicated with the labels a, b, . . . , k, and the direction in which curve segments were traced.

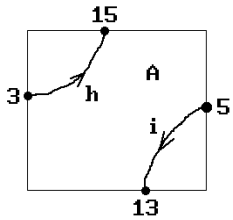


Figure 13.11. Curves in a hunting subgrid.

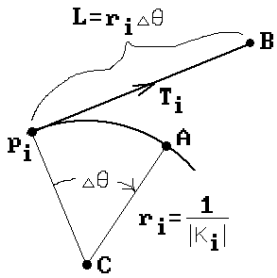


Figure 13.12. Determining the step size in Timmer's algorithm.

The actual tracing of the curves is carried out using a standard Newton-Raphson approach, but it is worth commenting on how two details were implemented.

First, tangent vectors for the curves were used for stepping. They are obtained from the normals to the surfaces provided that the surfaces are not tangent at the point of intersection. Specifically, if $\mathbf{x} = p(u_1,v_1) = q(u_2,v_2)$ is a point in the intersection $\mathbf{X}$, then

$$(p_u(u_1,v_1) \times p_v(u_1,v_1)) \times (q_u(u_2,v_2) \times q_v(u_2,v_2))$$

is a tangent vector to $\mathbf{X}$ at $\mathbf{x}$. Note however that we are making all of our computations relative to p(u,v), so that we have the parameter (u_1,v_1) but not (u_2,v_2). To get the parameters (u_2,v_2), another iterative procedure is used that, given a point $\mathbf{x}$, solves an equation of the type

$$\mathbf{x} - q(u,v) = 0$$

for (u,v).

Second, one needed to decide on a step size L. See Figure 13.12. Suppose that we are at point $\mathbf{p}_i$. Let $\mathbf{T}_i$ be the unit tangent vector at $\mathbf{p}_i$. Let κ_i be the curvature of the curve at $\mathbf{p}_i$. This value can be computed using the first and second partials of p(u,v) and q(u,v). Let r_i be the radius of curvature, namely, the radius of the osculating circle. Then r_i is just the reciprocal of the absolute value of κ_i. The step size L is then defined to be the length $r_i\Delta\theta$ of the arc from $\mathbf{p}_i$ to $\mathbf{A}$ of the osculating circle with center $\mathbf{C}$, where $\Delta\theta$ is some predefined constant angle tolerance. (It should be noted, however, that there is no need to actually compute the osculating circle because we only need the radius.) The point $\mathbf{B} = \mathbf{p}_i + L\mathbf{T}_i$ in Figure 13.12 was then used as a starting guess for a Newton-Raphson iteration to get the next point $\mathbf{p}_{i+1}$.

Timmer's Ordering Phase. We have to take our list of curve segments and combine them appropriately to get the complete curve. This is not totally trivial. Only the endpoints of the curve pieces are important. Basically, one starts with the first point uv_1 and the piece to which it belongs, then looks for another piece that has an endpoint that matches the second endpoint of our piece, and continues in this manner. For the example in Figure 13.10 we would get following connected pieces:

a, –c, d, –i, j, k
b, –e, f
h, –m, l, –g

This concludes our sketch of Timmer's algorithm. The problem with Timmer's algorithm is that it does not always give the correct answer and it is not as efficient as some more recent algorithms. Although more recent algorithms also fail at times, they give the correct answers in many more cases. We shall describe one more marching algorithm, namely, the one by Barnhill and Kersey ([BarK90]), although strictly speaking this algorithm is a hybrid method since it includes elements of recursive subdivision. Its primary goal was to be more general and more robust than previous such algorithms while still being efficient.

The Barnhill-Kersey Hunting Phase. The first step is to subdivide the rectangular or triangular domain of each of the parameterizations in an adaptive way. A quadtree data structure is used to represent a subdivision. Several types of subdivisions are performed. First, there is a uniform subdivision of the domain down to a user specified level for the quadtree. Then may come a further subdivision of those subpatches that do not meet a flatness and edge linearity criterion based on the angles between normals and tangents, respectively. For each subpatch we want the normals at the vertices and at one interior point to be almost parallel. Equivalently, the angles between them should be small. See Figure 13.13(a). In addition, for each edge of the subpatch, if **T** and **T′** are the unit tangent vectors at the endpoints of that edge, then

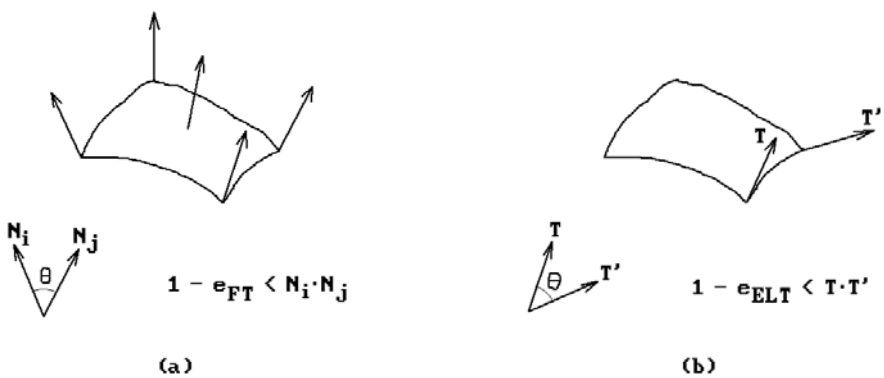


Figure 13.13. Flatness and edge linearity tests in the Barnhill-Kersey algorithm.

we also want those two vectors to be almost parallel or the angle between them small. See Figure 13.13(b). The e_{FT} and e_{ELT} in the figures represent some predefined **flatness** and **edge linearity tolerances**, respectively.

At this stage, a parallelopiped bounding box is associated to each subpatch. Care must be taken in the definition of these bounding boxes so that they satisfy their essential property, which is that they contain their subpatch. The initial guess for a bounding box is determined from the vertices of the patch but is then enlarged using geometric information about the edge tangents and normal vectors. The subdivision is assumed to be fine enough so that the simple geometric argument that is used works. One uses the bounding boxes to cull away subpatches from the two surfaces that cannot possibly intersect. Determining whether two parallelopipeds intersect is done by transforming the second into the skew coordinate system determined by the first. The problem reduces to determining whether a parallelopiped intersects the unit cube and this has a straightforward solution.

Once one has found potential subpatch intersections, there may be some more subdivisions of those to increase accuracy and/or achieve convergence for subsequent Newton-Raphson steps. Suppose now that we have two bounding boxes that intersect, one from each surface. One then takes the average of the corner vertices of the subpatches associated to the bounding boxes and relaxes it to the intersection. One does this for every pair of intersecting bounding boxes. This will give us a collection of points on the intersection that are used as start points for the tracing stage, but before we get to that we want to describe how points are relaxed in [BarK90]. Such an operation is performed repeatedly in the tracing stage.

Relaxing Points in the Barnhill-Kersey Algorithm. Let $\mathbf{p}$ be the point to be relaxed. The point $\mathbf{q}$ in the intersection to which $\mathbf{p}$ is relaxed will be the limit of a sequence of points $\mathbf{p}_i$, where $\mathbf{p}_0 = \mathbf{p}$. Assume that we have already found $\mathbf{p}_i$. We describe how the point $\mathbf{p}_{i+1}$ is defined. The first step is to find points $\mathbf{q}_1$ in $\mathbf{S}_1$ and $\mathbf{q}_2$ in $\mathbf{S}_2$ that are closest to $\mathbf{p}_i$. Ways to carry out this step are described in Section 14.2. If the points $\mathbf{q}_1$ and $\mathbf{q}_2$ are within a predefined **same point tolerance** e_{SPT}, then we have found our $\mathbf{q}$ and we quit. Otherwise, let $\mathbf{T}_j$ be the tangent planes to $\mathbf{S}_j$ at $\mathbf{q}_j$. The point $\mathbf{p}_{i+1}$ will be the midpoint of the projections of $\mathbf{q}_1$ and $\mathbf{q}_2$ onto the line which is the intersection of $\mathbf{T}_1$ and $\mathbf{T}_2$. See Figure 13.14. Alternatively, let $\mathbf{n}_j$ be a unit normal vector for plane $\mathbf{T}_j$ and define a third nonparallel plane $\mathbf{T}_3$ as the plane through the point $(1/2)(\mathbf{q}_1 +$

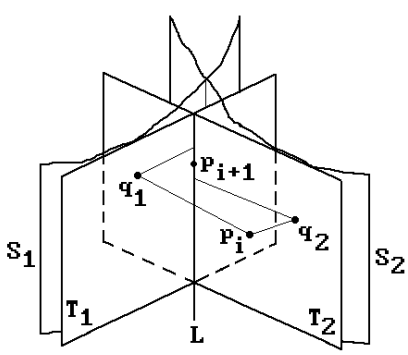


Figure 13.14. The Barnhill-Kersey relaxation algorithm.

$\mathbf{q}_2$) and normal $\mathbf{n}_3 = \mathbf{n}_1 \times \mathbf{n}_2$. Then $\mathbf{p}_{i+1}$ is the intersection of the three planes $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$. The special case where the planes $\mathbf{T}_1$ and $\mathbf{T}_2$ are parallel is easily handled separately. Convergence, although a problem theoretically, was not a problem in practice given that points were only needed within the tolerance e_{SPT}.

The method for relaxing points that we just described is what is used for determining the intersection in the interior of the surface patches, which is most of the time, but not for relaxing points onto the boundary of the patches. In that case one uses a Newton-Raphson approach to solve

$$p(u,v) - q(s,t) = \mathbf{0}$$

along with a parameter constraint determined by the boundary to which one is relaxing, for example, u = 0. We have a system of three equations in three unknowns. Parallel tangent planes are again a problem.

The Barnhill-Kersey Tracing Phase. This phase is started for each intersection point obtained in the hunting phase. For each start point, one generates a sequence of points that lie on the intersection. As one moves from one point $\mathbf{p}$ to the next one needs a step direction and step size. The tangent of the intersection curve at the current point is used as the step direction. Like in Timmer's algorithm, this is computed from the cross-product of the normal vectors to the surfaces at $\mathbf{p}$. One can save some multiplications by using Theorem 1.10.4(2) in [AgoM05] and use the direction

$$(p_u \times p_v) \times \mathbf{n}_2 = (p_u \bullet \mathbf{n}_2)p_v - (p_v \bullet \mathbf{n}_2)p_u$$

instead. Unfortunately, these formulas give the zero vector if the surfaces are tangent. In that case one can take the difference of previous intersection points. If there are no previous points and we are at the first intersection point, say $\mathbf{p} = p(u_0,v_0) = q(s_0,t_0)$, one samples points on the circles around (u_0,v_0) and (s_0,t_0) in the domains of the parameterizations to find another point on the intersection. The new step direction is then taken to be the difference between this point and $\mathbf{p}$.

Once one has the step direction one has to decide on a step size L. Like in Timmer's algorithm one wants to use a formula of the form $L = r\Delta\theta$, where r is the radius of curvature and $\Delta\theta$ is some predefined **angle tolerance**. However, the parameterizations were only assumed to be C^1 here and so the second derivatives are not directly available to compute this radius. Therefore, an approximation was used. Assume that we are a point $\mathbf{p}$ on the intersection. Two points a small distance ε from $\mathbf{p}$ on the tangent line to the intersection curve at $\mathbf{p}$ are relaxed to points $\mathbf{x}$ and $\mathbf{y}$ on the intersection. The three points $\mathbf{p}$, $\mathbf{x}$, and $\mathbf{y}$ determine a circle and we let r be its radius because we can consider this circle to be an approximation to the osculating circle. Formula 6.8.1 implies that

$$r = \frac{|\mathbf{a}||\mathbf{b}||\mathbf{a} - \mathbf{b}|}{2|\mathbf{a} \times \mathbf{b}|},$$

where $\mathbf{a} = \mathbf{x} - \mathbf{p}$ and $\mathbf{b} = \mathbf{y} - \mathbf{p}$. If the three points were collinear or L turned out to be larger than some predefined **curve refinement tolerance** e_{CRT}, then L was set to e_{CRT}.

Given a unit step vector $\mathbf{v}$ at a point $\mathbf{p}$, we march to the next point $\mathbf{p} + L\mathbf{v}$ and relax it to the intersection. This describes the basic tracing step, but there is one more complication caused by what are called "terminating" points. These are points where the intersection curve

(1) meets the initial point on the same curve segment, that is, the segment is a closed curve,
(2) meets the patch boundary, or
(3) meets an earlier curve segment.

Case (1) is simple and we indicated how case (2) is handled in the discussion of point relaxation. Points of type case (3) are discovered by checking for intersections of curve segments in the parameter space of the parameterizations. The points tend to be "branch" points, namely, places where multiple intersection curve segments meet. Once an intersection of parameter segments is found one has to find the branch point and make sure that the adjacent curve segments get sorted correctly.

Here is how one finds a branch point $\mathbf{p}$. We take two endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$ of distinct crossing curve segments $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively, as start points and then define a sequence of points $\mathbf{p}_i$. The points $\mathbf{p}_i$ should alternate between lying on $\mathbf{C}_1$ and $\mathbf{C}_2$. See Figure 13.15(a). One continues generating $\mathbf{p}_i$ until two successive ones are within a tolerance of e_{SPT}. If there is no convergence, then we do not have a branch point. Here is how $\mathbf{p}_{i+1}$ is defined, assuming that $\mathbf{p}_{i-1}$ and $\mathbf{p}_i$ have already been computed. See Figure 13.15(b). Let $\mathbf{q}$ be the orthogonal projection of $\mathbf{p}_i$ on the tangent line to the intersection curve segment passing through $\mathbf{p}_{i-1}$. If $\mathbf{n}_i$ is a normal vector to surface $\mathbf{S}_i$ at $\mathbf{p}_{i-1}$, then

$$\mathbf{q} = \mathbf{p}_{i-1} + \frac{(\mathbf{p}_i - \mathbf{p}_{i-1}) \bullet (\mathbf{n}_1 \times \mathbf{n}_2)}{|\mathbf{n}_1 \times \mathbf{n}_2|^2} \mathbf{n}_1 \times \mathbf{n}_2.$$

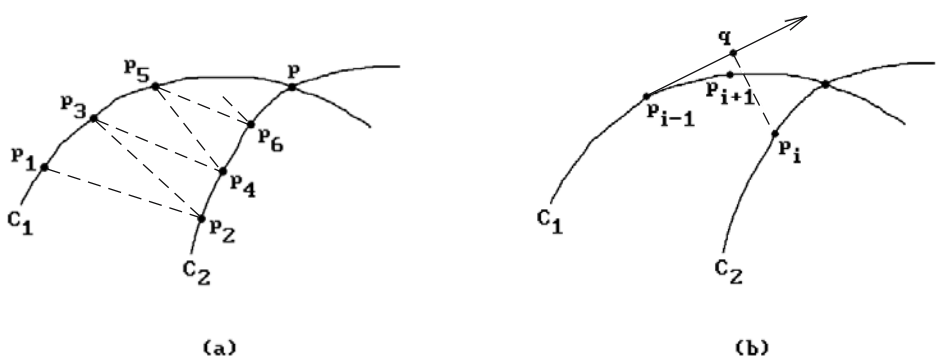


Figure 13.15. Finding branch points.

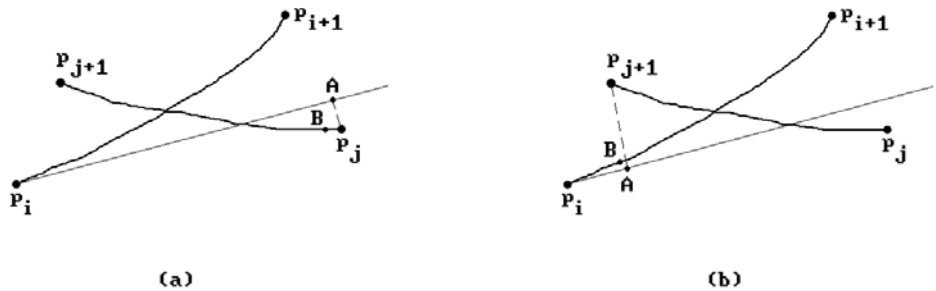


Figure 13.16. A branch convergence problem.

The point **q** is then relaxed to an intersection point that is defined to be $\mathbf{p}_{i+1}$. There are numerical problems if the tangents to the two curve segments at the branch point are almost parallel. A further problem is indicated in Figure 13.16. Suppose that we have detected an intersection in the parameter segments whose endpoints parameterize two pairs of points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$ and $\mathbf{p}_j$, $\mathbf{p}_{j+1}$. In Figure 13.16(a), if we were to choose $\mathbf{p}_i$ and $\mathbf{p}_j$ as start points of our iteration, then $\mathbf{p}_j$ projects to the point **A** on the tangent line at $\mathbf{p}_i$ and the point **A** relaxes to a point **B** which is again on the second curve segment. This violates our condition that our sequence of points should alternate between curve segments. If we had started with, say, $\mathbf{p}_i$ and $\mathbf{p}_{j+1}$, as indicated in Figure 13.16(b), then everything would have been fine. To avoid this problem, Barnhill and Kersey suggest testing the angle between the vectors $\mathbf{p}_i\mathbf{p}_{i+1}$ and $\mathbf{p}_j\mathbf{p}_{j+1}$. If the angle is less than 90 degrees, that is,

$$\mathbf{p}_i\mathbf{p}_{i+1} \bullet \mathbf{p}_j\mathbf{p}_{j+1} > 0,$$

then choose $\mathbf{p}_i$ and $\mathbf{p}_j$, otherwise, choose $\mathbf{p}_i$ and $\mathbf{p}_{j+1}$. There are numerical problems when the angle is near 0 or 90 degrees.

The algorithm in [GraK97] uses a different approach to finding terminating points in the case of tensor product spline surfaces. It uses an algebraic method for hunting and finds the terminating points **before** starting the tracing. The method is based on an algorithm that finds all solutions to the kind of nonlinear systems of equations one gets in the tensor product spline surface case.

Tolerances in the Barnhill-Kersey Algorithm. Several predefined tolerances were used in the algorithm. The main one was the same point tolerance e_{SPT}. In [BarK90] it was claimed that assigning it a value of 10^{-7} gave good results on a computer with 16-decimal-place precision. It was also recommended to use tolerances based on angles as much as possible to remove dependencies on surface magnitudes and units of measure.

The Data Structures in the Barnhill-Kersey Algorithm. The main structure is that of a quadtree node. Each such node maintained the following information:

its level,
the coordinate transformation and its inverse for the associated bounding box,
edge linearity and flatness measures,
the parameter domain,
the values, partials, and normals at the vertices and the centroid,
a pointer to a doubly linked adjacency list used for sorting, and
pointers to child nodes.

An adjacency list node consisted of a pointer to a doubly linked list of nodes that contained intersection information, namely,

the start and endpoint of the domain of the segment in the form of four reals u_1, v_1, u_2, and v_2,
the start and endpoint $\mathbf{p}_1$ and $\mathbf{p}_2$ of the segment in $\mathbf{R}^3$, and
a flag indicating whether the segment terminated.

There was a list of pairs of quadtree nodes whose bounding boxes intersected.

The Barnhill-Kersey Sorting Phase. The quadtree data structure facilitated the sorting phase, but is not necessary. Note that all non-closed polygonal curve segments generated by the tracing phase terminate at boundary or branch points. When sorting those segments, endpoints are considered the same if they are within a tolerance of e_{SPT}. The recursive divide-and-conquer algorithm described in [HEFS85] was used on the partial adjacency lists associated to the quadtree nodes.

This finishes our discussion of the algorithm in [BarK90]. The authors compared their algorithm to others, in particular to those in [BFJP87] and [HEFS85]. The advantages were that it also worked for parameterizations with triangular domains, did not need second derivatives, and handled more tricky cases correctly. It was at least as robust or more than other surface intersection algorithms. The tolerance constants described above could be adjusted manually to achieve better results in difficult cases. The reader might also look at the algorithm in [Luka89] for additional details on the Newton-Raphson mathematics.

As a third example of a marching method for finding the intersection of two surfaces, assume that the intersection curve lies in the uv-plane, is defined by an equation $f(u,v) = 0$, and we have already computed the ith solution point $\mathbf{p}_i = (u_i,v_i)$. Of course, one can look at this version of the intersection problem from various points of view. It can be thought of as an implicit curve or contour problem. Sections 14.5.1 and 14.6 discuss those. Here we shall describe a "step constraint" approach for getting the next point. Let us add another constraint $g(u,v) = 0$. For example,

$$g(u,v) = (u - u_i)^2 + (v - v_i)^2 - d^2 \tag{13.15}$$

corresponds to requiring that the next point $\mathbf{p}_{i+1}$ is on the intersection a distance d from $\mathbf{p}_i$. A step in the direction of the tangent of the curve at (u_i,v_i) is used as an initial guess and then a Newton-Raphson method is used to converge to the next solution. See Figure 13.17. There are the usual potential problems such as needing a start point

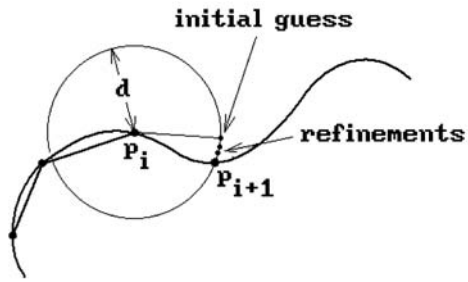


Figure 13.17. A step constrained marching method.

on the intersection curve and possible lack of convergence where the partials of f vanish. Additionally, the step constraint must be chosen carefully. For example, for a constraint of the type shown in equation (13.15), the step size d influences the result and the value of d may change from point to point. We do not want to step to another branch of the curve. We also have to check for the possibility that our solution takes us outside the given (u,v)-domain. If the intersection curve is closed, then we also want to be able to close our curve explicitly.

A fourth marching method approach tries to minimize a function that vanishes on the set of interest. For example, if the intersection is defined by

$$f(x,y,z) = 0$$
$$g(x,y,z) = 0$$

and we use a step constraint

$$h(x,y,z) = 0,$$

then we could try to minimize

$$F = f^2 + g^2 + h^2. \tag{13.16}$$

See, for example, [Powe72] or [PraG86].

We mention one last marching type approach. Here one uses vector fields and differential equations, where intersection curves are solutions to the latter. See, for example, [PhiO84], [Chen89], [MarM89], or [KrPW92].

At the heart of the typical marching method is the Newton-Raphson method. One ends up with a discrete approximation to the solution. As one generates the points one always starts with a guess for the next point and then uses the Newton-Raphson method to successively refine that guess until we get a point that lies on the actual solution set with desired accuracy. Let us recapitulate some of the common problems marching methods have to contend with:

(1) They need starting points.
(2) The direction and step size has to be selected very carefully to avoid missing entire pieces of the intersection or accidentally stepping from one component to another.

(3) They get tricky to implement near terminating points.
(4) They have problems at singular points.

What differentiates the various marching algorithms is how these problems are addressed. We finish this section with a discussion of these points.

When it comes to picking start points, we have seen how the Timmer and Barnhill-Kersey algorithms do it. For more on the subject we refer the reader to [AbdY97]. That paper discusses two general methods for picking start points along with an analysis and comparison.

The question of step direction and size really has to do with finding a good approximation to the next point on the curve. Obviously, the better these guesses are, the faster the Newton-Raphson method converges to the solution. Most of the time, the guesses are based on linear approximations to the solution. For example, a starting guess for the next point is often chosen from the points along the tangent line at the current point of an intersection curve. This is often only a crude guess. A higher-order approximation to the solution set would lead to quicker convergence of the Newton-Raphson method. [Stoy92] suggests using a second-order approximation, that is, parabolas. In that way one was also able to control the deviation of the actual intersection curve from its polygonal approximation.

We shall sketch the mathematics involved in getting higher order approximations. The details can be found in [BHLH88] and [Hoff89]. Consider the implicit/implicit case where the two surfaces $\mathbf{S}_1$ and $\mathbf{S}_2$ in $\mathbf{R}^3$ are defined by equations

$$f(x,y,z) = 0 \quad \text{and} \quad g(x,y,z) = 0. \tag{13.17}$$

As usual our object is to define a sequence of points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \ldots$, so that the polygonal curve they define approximates the intersection of $\mathbf{S}_1$ and $\mathbf{S}_2$. We need to describe how we get the first point and then how we get from one point to the next.

Assume that we are given a point $\mathbf{p} = \mathbf{p}_i$ that lies on the intersection of $\mathbf{S}_1$ and $\mathbf{S}_2$ and we want to get the next point $\mathbf{p}_{i+1}$. Assume further that the gradients $\nabla f(\mathbf{p})$ and $\nabla g(\mathbf{p})$ are not zero and that the surfaces intersect transversally at $\mathbf{p}$. If any of these conditions are not satisfied, the approach that will be described here will probably fail and an algebraic geometry approach would be appropriate if f and g are polynomials. The general assumption we make here is that the intersection of $\mathbf{S}_1$ and $\mathbf{S}_2$ can be defined by an analytic function of the form

$$\gamma(t) = \mathbf{a}_0 + \mathbf{a}_1 t + \mathbf{a}_2 t^2 + \ldots, \quad \text{where} \quad \mathbf{a}_i = \frac{1}{i!}\gamma^{(i)}(0), \tag{13.18}$$

defined in the neighborhood of the origin with $\gamma(0) = \mathbf{p}$. We use the first m terms of this power series to approximate $\gamma(t)$. The case m = 1 corresponds to the usual linear approximation and so of interest here is the case m > 1, specifically when m = 2 or 3. To accomplish the goal, we need to solve for the $\mathbf{a}_i$, i = 0, 1, . . . , m.

Let $\mathbf{q} = \mathbf{p} + \boldsymbol{\delta}$ and let $\boldsymbol{\delta} = (\delta_1, \delta_2, \delta_3)$. Consider the Taylor expansion

$$f(\mathbf{q}) = \sum_{d=0}^{\infty} \sum_{i+j+k=d} a_{ijk}\delta_1^i\delta_2^j\delta_3^k, \quad a_{ijk} = \frac{1}{i!\,j!\,k!}\frac{\partial^{i+j+k} f}{\partial x^i \partial y^j \partial z^k}(\mathbf{p}) \tag{13.19}$$

for f around **p**. Assume that $\gamma(t) = \mathbf{q}$. Since $\gamma(t)$ lies in $\mathbf{S}_1$, $h(t) = f(\gamma(t)) = 0$. Therefore, if one substitutes the series (13.18) into (13.19) all the coefficients of the resulting power series in t must vanish. Setting the coefficients of t^i, $i = 1, 2, \ldots, m$, to zero gives m equations in 3m unknowns $\gamma_j^{(i)}(0)$, $j = 1, 2, 3$, and $i = 1, 2, \ldots, m$. Applying the same argument to the function g that defines the surface $\mathbf{S}_2$ gives another m equations. This underdetermined system of equations can be solved. By adding some additional geometric constraints dealing with the curvature and moving triad associated to the space curve $\gamma(t)$, one can get a unique solution. The way that the additional constraints are chosen affects the parameterization $\gamma(t)$ of the intersection of the surfaces. [BHLH88] and [Hoff89] also discuss how to choose the step $\boldsymbol{\delta}$. One has to watch out that one does not get outside the radius of convergence of $\gamma(t)$. This is not easy, but one can give bounds for the maximum step size. In the case m = 3 these depend on the second and third derivative of γ.

Having gotten an approximation to the next point on the surface intersection, one performs a Newton-Raphson iteration consisting of points $\mathbf{q}_0 = \mathbf{q}, \mathbf{q}_1, \mathbf{q}_2, \ldots$, to find the point $\mathbf{p}_{i+1}$ that actually lies in that intersection with as much accuracy as desired. One point to note is that what we just did for $\mathbf{R}^3$ easily extends to computing the intersection of n − 1 hypersurfaces in $\mathbf{R}^n$.

Using higher-order approximations to curves takes computation time, which is why many algorithms are satisfied with linear approximations and step in the tangent direction. However, one must also decide on the step size. Intuitively, it makes a lot of sense to base the decision on the osculating circle since it is the best circle approximation to the curve at the point. Both the Timmer and Barnhill-Kersey algorithms step in the tangent direction and use the radius of the osculating circle to determine a step size. Since everything is an approximation anyway, the only question is how accurately we should compute the osculating circle and its radius r. Computing second derivatives is usually not cheap. Timmer's algorithm did compute r exactly. The Barnhill-Kersey algorithm used an approximation to the real r. Neither algorithm actually tried to compute the osculating circle and only needed its radius. The algorithms in [CheO88] and [Aste88], on the other hand, actually stepped along the circle. The first applied to parametric surfaces and the second to implicit surfaces. The approximation in [WuAn99] is better yet for parametrically defined surfaces. The authors define an approximation to the osculating circle that can be computed efficiently and step in its tangent direction (not along the tangent to the curve). The equation for their circle is based on the current and previously computed curve point and tangent. Let $\mathbf{T}_i$ denote the tangent to the intersection curve at point $\mathbf{p}_i$. Let **C** be the point that is the intersection of the following three planes $\boldsymbol{\pi}_1$, $\boldsymbol{\pi}_2$ and $\boldsymbol{\pi}_3$ described in point-normal form:

plane $\boldsymbol{\pi}_1$: point $\mathbf{p}_{i-1}$, normal vector $\mathbf{T}_{i-1}$
plane $\boldsymbol{\pi}_2$: point $\mathbf{p}_i$, normal vector $\mathbf{T}_i$
plane $\boldsymbol{\pi}_3$: point $\mathbf{p}_i$, normal vector $\mathbf{T}_{i-1} \times \mathbf{T}_i$

The point **C** is then assumed to be the center of the circle. The plane of the circle is assumed to have normal vector $\mathbf{Cp}_{i-1} \times \mathbf{Cp}_i$. See Figure 13.18. Degenerate cases are handled in the paper. One can show that this circle converges to the actual osculating circle as $\mathbf{p}_i$ approached $\mathbf{p}_{i-1}$. The marching distance is again taken to be $r\Delta\theta$, where $r = |\mathbf{Cp}_i|$ and $\Delta\theta$ is a predefined tolerance. It was found that this method leads to a robust marching algorithm that

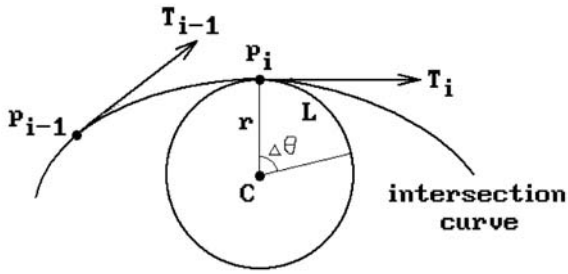


Figure 13.18. An approximation to the osculating circle.

(1) has the same complexity but better accuracy than using the tangent vector direction,
(2) is simpler than using parabolic approximations to the curve but not quite as accurate, and
(3) is as reliable as continuation methods but is more general.

This still leaves open the question as to whether one has chosen small enough steps so as to find all parts of the intersection curves. Therefore, the methods need an add-on for loop detection such as is described in [SedM88], [SeCK89], [Hohm91], and [MaLe98]. This also applies to finding the start points. Unfortunately, it is very hard to set tolerances correctly.

The problem of terminating points was encountered in our discussion of the Barnhill-Kersey algorithm where we had to make a special case out of finding those points and dealing with them. Finally, the problem with singular points is that the Newton-Raphson iteration method involves solving linear equations and the matrices for these equations usually have very bad condition numbers at the singular points, which means that one loses all accuracy. The iteration may not converge or may miss parts of the intersection curves.

[KrPP90] describes a hybrid algorithm that is basically a marching algorithm but uses algebraic methods to make sure that no part of the intersection is missed. It applies to the intersection of an algebraic and a rational parametric surface. The key point was being able to detect all the "significant" points on the intersection curve. Those are the boundary points, turning points (where the normal vector to the curve is parallel to the u- and v-axis), and singular points (where the first partials vanish).

13.5.3 Surface Homotopy Method

The problem of finding intersections of implicit surfaces can be thought of as a special case of the more general problem of solving systems of polynomial equations. One much-studied technique that is used is called the *homotopy method*. It has also been called the *homotopy continuation method* or simply a *continuation* or *embedding method*. The idea is to find the solutions to a related set of simpler equations first and then to deform these equations and their solutions into the given ones. A simple example, presented in [PraG86], considers the problem of finding the solutions to the equations

$$\begin{aligned} x^2 - xy - 1 &= 0 \\ y^2 - x - 5 &= 0. \end{aligned} \tag{13.20}$$

We note that the following related equations

$$\begin{aligned} x^2 - 1 &= 0 \\ y^2 - 5 &= 0 \end{aligned} \tag{13.21}$$

are easily solved and the solutions to the parameterized equations

$$\begin{aligned} x^2 - txy - 1 &= 0 \\ y^2 - tx - 5 &= 0 \end{aligned} \tag{13.22}$$

define a homotopy between the solutions to equations (13.20) and (13.21). Therefore, to solve the equations (13.20) we compute the incremental changes to the solutions to (13.21) as t changes from 0 to 1. One surface intersection algorithm that uses the homotopy method can be found in [AbdY96].

Here is an overview of the general homotopy method. A good survey can be found in [AllG90] and [Wats86]. Some other helpful papers are [GarZ79], [Morg83], [Wrig85], and [AllG93], where one can also find many additional references.

A system of m polynomial equations in n variables corresponds to a polynomial map f: $\mathbf{R}^n \to \mathbf{R}^m$ and an equation

$$f(\mathbf{x}) = \mathbf{0}. \tag{13.23}$$

We choose another system of equations

$$g(\mathbf{x}) = \mathbf{0} \tag{13.24}$$

defined by a map g: $\mathbf{R}^n \to \mathbf{R}^m$ whose zeros are known and consider the homotopy h: $\mathbf{R}^n \times \mathbf{R} \to \mathbf{R}^m$ between $g(\mathbf{x})$ and $f(\mathbf{x})$ defined by

$$h(\mathbf{x}, t) = (1-t)g(\mathbf{x}) + tf(\mathbf{x}). \tag{13.25}$$

Let $\mathbf{x}_0$ be a zero of g(x). The object is to find a curve $\gamma(t)$ in the zero set of h that starts at $(\mathbf{x}_0,0)$ and ends at a point $(\mathbf{x}_1,1)$, where $\mathbf{x}_1$ is a zero of f. More precisely, we look for a curve

$$\gamma : [0,1] \to \mathbf{R}^n \times \mathbf{R}$$

with the property that

(1) $\gamma(0) = (\mathbf{x}_0,0)$
(2) $h(\gamma(t)) = \mathbf{0}$, for all $t \in [0,1]$, and
(3) $\gamma(1) = (\mathbf{x}_1,1)$, with $f(\mathbf{x}_1) = \mathbf{0}$.

One way to find $\gamma(t)$ is to differentiate the equation in condition (2). This shows that $\gamma(t)$ must satisfy

$$J(\gamma(t),t)\frac{d\gamma}{dt}(t)+\frac{\partial h}{\partial t}(\gamma(t),t)=0, \tag{13.26}$$

where $J(\mathbf{x},t)$ is the Jacobian matrix

$$J(\mathbf{x},t)=\left(\frac{\partial h_i}{\partial x_j}(\mathbf{x},t)\right).$$

If the rank of $J(\mathbf{x},t)$ is n for all values t, then the implicit function theorem will guarantee that $\gamma(t)$ exists. We are left with the problem of solving the system of ordinary differential equations defined by (13.26) and can use any of the well-known ways to solve such equations.

A second way to find $\gamma(t)$ is to use a standard root finding method like the Newton-Raphson method to the equation in condition (2) above.

We mention two potential problems with using the homotopy method. In general terms, what sets the homotopy method apart from previous iterative schemes is the fact that it replaces a "local convergence" approach with a "global convergence" one. With the homotopy method we already have some knowledge about certain initial zeros (if our initial guess for a zero is poor when using a standard Newton-Raphson method then we may have extreme difficulty in converging to a zero) and we can use them to iterate to **all** of the zeros of our function. Nevertheless, we are still dealing with an iterative process and one big problem with the type of iterative methods that are used is that when relevant Jacobian matrices have singularities, then it is very hard to guarantee the convergence to a desired solution. It took a lot of work to overcome the singular Jacobian matrix problem and make the homotopy method work and be reasonably efficient.

Another problem that caused great inefficiencies in the homotopy method initially was the choice of a start function $g(\mathbf{x})$. The original approach was to use Bézout's theorem and choose a polynomial function $g(\mathbf{x})$ of total degree d that was the product of the degrees of the polynomials in $f(\mathbf{x})$. In practice, however, the number of zeros of $f(\mathbf{x})$ was often much smaller than d and so a lot of computation effort was wasted in generating curves $\gamma(t)$ that diverged to ∞ as $t \to 1$. It was important to find a better bound on the number of zeros of $f(\mathbf{x})$. This can be done in the case of *sparse polynomials*, that is, polynomials that have a relatively small number of monomial terms. One gets a more efficient homotopy method when $f(\mathbf{x})$ is a sparse polynomial. See, for example, the paper [VeVC94], which concentrates on sparse polynomial systems.

13.5.4 Surface Recursive Subdivision Methods

The general idea of using recursive subdivision (divide-and-conquer) in computer graphics is an old one. Display algorithms in early papers, such as [Catm74] and [LCWB80], made use of it. The first suggestion that it might be useful for intersection problems can be found in [LanR80]. Bézier and B-spline surfaces with their control

points are good candidates for this approach because their subdivisions have associated control nets. In [LanR80] the idea is to subdivide the surfaces enough so that the individual pieces are essentially flat (see also [Clar79] and [LCWB80]). One then finds the intersection of the approximating planar patches to get a polygonal approximation to the actual intersection. Because the surfaces satisfy the convex hull property, one can use bounding boxes to make the algorithm more efficient. This is the analog of the recursive subdivision for curve intersections in Sections 13.3.1 and 13.3.4.

One problem associated to recursive subdivision is the amount of data that could be generated potentially. The way to mitigate this problem is by not doing the subdivision down to a certain level blindly. One should keep subdividing an object only at those places where there is a possibility of it intersecting the other object. The typical way that such an adaptive approach is carried out is by using some sort of bounding box approach. Min-max boxes, which are obtained from the minimum and maximum values of the coordinates of the points of an object, are common choices because they are so simple. Other bounding objects are slabs (see Section 6.2), convex hulls in the case of Bézier or B-spline surfaces, or "fat" planes ([Carl82]) for Bézier surfaces. If the bounding objects do not intersect, then the objects will not intersect. Checking for intersections of the bounding objects is much easier than checking for intersections between the objects themselves. One subdivides the objects only at those places where the bounding boxes intersect. This approach implies that one has a termination criterion. The usual such is based on a flatness test. There are many ways to check for flatness, but one simple one is to use the normal vectors. The normal vectors will not change much over regions that are close to flat. Of course, no matter how simplified a test one uses, this will still take some extra time and so the more one can avoid having to use such tests, the better. One might be able to say in advance how much subdivision is needed.

The algorithm in [HEFS85] is an example of a recursive subdivision algorithm for C^1 parametric surfaces. It proceeds in four steps: subdivision, intersection, sorting, and refinement. The subdivision is done in stages. First, the surfaces are subdivided until the subpatches satisfy an initial flatness and edge linearity criterion. A stack of possibly intersecting pairs of subpatches is created. The test for intersection here is based on whether associated "oriented" bounding boxes (bounding boxes that roughly line up with their patches) intersect. The leftover subpatches are successively subdivided further to higher flatness tolerances. A tree data structure is used to maintain the data generated. When subpatches are flat enough, they are approximated by two triangles and the intersections of these triangles are used as approximations to the intersection of the patches. This produces a collection of segments that the sorting phase must then combine to get the polygonal curve segments that become the approximation of the intersection curves of the surfaces. Since the algorithm is based on adaptive subdivision there may be gaps between adjacent triangles, but the algorithm checks for this and redefines subpieces appropriately to prevent the problem from occurring. Finally, the triangle intersections are only approximations, therefore a refinement step iteratively tries to find points closer to the surfaces. We described a slightly improved version of this refinement in the Barnhill-Kersey algorithm in Section 13.5.2. An algorithm similar to the one in [HEFS85] is described in [FiMM86]. There bounding boxes are defined using derivative bounds.

On a related topic, [Nasr87] discusses finding intersections of recursive subdivision surfaces.

13.5.5 Surface Algebraic Methods

When one tries to use algebraic methods to solve intersection problems, one looks first for special cases that can be handled by special techniques. For example, plane/plane or plane/quadric intersections produce lines and conics, respectively, and can be solved for explicitly. In general, because the implicit/parametric case is relatively easy, attempts have been made to reduce other cases to this one.

An implicit surface **S** can be parameterized, but not necessarily by rational polynomial functions. If **S** is defined by linear or quadratic polynomials, then **S** can be parameterized by rational polynomial functions. If **S** is defined by higher-degree polynomials, then it may not admit such a parameterization. See Section 10.15 in [AgoM05] for more details. Parameterized surfaces, on the other hand, can always be represented implicitly by rational polynomial functions if the parameterization was also of that form. The only problem is that the standard implicitization algorithms may produce very complicated equations. See Sections 10.9 and 10.10 in [AgoM05]. For example, it can be shown that a bicubic patch is equivalent to an algebraic surface of degree 18 whose equation contains 1330 terms. For these reasons, algebraic geometry methods seem to be impractical currently, but there is a lot of ongoing research.

At any rate, because it is known that every algebraic curve in $\mathbf{R}^3$ can be mapped to an algebraic curve in $\mathbf{R}^2$ (although the latter may be more complicated than the former), one general algebraic approach to solving the surface intersection problem is:

(1) Map the intersection curve in $\mathbf{R}^3$ to a plane curve defined by an equation

$$h(u,v) = 0. \tag{13.23}$$

(2) Solve equation (13.23).
(3) Map the solution back to $\mathbf{R}^3$.

Step (2) is considered in more detail in Sections 14.5.1 and 14.6. Here we describe two approaches to (1) and (3). One is based on substitutions and the other, on projections. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be surfaces in $\mathbf{R}^3$.

The Substitution Approach. Suppose that surface $\mathbf{S}_1$ is defined implicitly by an equation

$$f(x,y,z) = 0 \tag{13.24}$$

and surface $\mathbf{S}_2$ is defined via a parameterization

$$g(u,v) = (g_1(u,v), g_2(u,v), g_3(u,v)).$$

Substituting into equation (13.24) gives

$$h(u,v) = f(g_1(u,v), g_2(u,v), g_3(u,v)) = 0. \tag{13.25}$$

If we solve equation (13.25) in the u-v plane, then we can map the solution back to $\mathbf{R}^3$ using g.

If both surfaces are defined parametrically, we can implicitize one of them using the Gröbner basis approach and thus reduce the problem to the previous case. We could also use the resultant to implicitize a surface, but that method has the disadvantage of introducing extraneous factors. See Section 10.9 in [AgoM05].

If both surfaces are defined implicitly, we would like to parameterize one of them to again reduce the problem to the case solved above. Unfortunately, as was pointed out earlier, it is not always possible to parameterize an implicitly defined surface by rational functions. On the other hand, it can be shown that the intersection of two implicitly defined surfaces always lies on a parameterizable surface **X**. Here is an algorithm that will produce such a surface. Assume that $\mathbf{S}_1$ and $\mathbf{S}_2$ are the zero sets of functions f(x,y,z) and g(x,y,z), respectively.

Step 1. Homogenize the function f(x,y,z) and g(x,y,z) to get homogeneous functions F(x,y,z,w) and G(x,y,z,w), respectively. The intersection of the surfaces $\mathbf{S}_1$ and $\mathbf{S}_2$ corresponds to the nonideal points of the intersection of the homogeneous hypersurfaces defined by F and G.

Step 2. Choose one of the variables appearing in F or G and express F and G as polynomials in that variable (with coefficients that are polynomial in the other variables). If we suppose that w was chosen, then we write

$$\begin{aligned} F &= a_0 + a_1 w + a_2 w^2 + \ldots + a_m w^m \\ G &= b_0 + b_1 w + b_2 w^2 + \ldots + b_{m'} w^{m'}, \end{aligned} \tag{13.26}$$

where a_i and b_j are polynomials in x, y, and z. Assume without loss of generality that $m \geq m' > 1$. We can assume that $m' > 1$ because otherwise f and g would be homogeneous polynomials and that special case will not be considered here. Define

$$\begin{aligned} F_1 &= a_m w^{m-m'} G - b_{m'} F \\ G_1 &= \frac{a_0 G - b_0 F}{w}. \end{aligned} \tag{13.27}$$

We can think of F_1 and G_1 as having been derived from F and G by removing their highest, respectively, lowest degree terms. Note that both are linear combinations of F and G and hence the intersection of the hypersurfaces defined by them contains the intersection of the hypersurfaces defined by F and G.

Step 3. Since the degree of F_1 and G_1 is less than n, we repeat Step 2 with F_1 and G_1 replacing F and G, thereby generating a sequence of polynomials F_i and G_i, until we finally end up with an F_k or G_k, which has degree 1. (The case where $F_i = G_i$ for some i and where we go from a degree larger than 1 to a degree 0 in one step is a special case not dealt with in our algorithm.) Using this linear polynomial in w we see that our (homogeneous) intersection lies on a hypersurface defined by an equation of the form

$$H(x,y,z,w) = wH_1(x,y,z) + H_2(x,y,z) = 0. \tag{13.28}$$

By induction, the polynomial H is a linear combination of F and G.

Step 4. The hypersurface defined by equation (13.28) can be parameterized (with homogeneous coordinates) by

$$\begin{aligned} x(r,s,t) &= r \\ y(r,s,t) &= s \\ z(r,s,t) &= t \\ w(r,s,t) &= -\frac{H_2(r,s,t)}{H_1(r,s,t)}. \end{aligned} \tag{13.29}$$

Substituting these functions into the formula for G, we get a homogeneous plane curve. Dehomogenizing this curve gives us an affine plane curve that is now solved.

13.5.5.1 Example. Consider the ellipsoid $\mathbf{S}_1$ centered at the origin and the sphere $\mathbf{S}_2$ centered at (1,0,0) defined by

$$f(x,y,z) = 4x^2 + y^2 + z^2 - 1 = 0 \tag{13.30}$$

and

$$g(x,y,z) = (x-1)^2 + y^2 + z^2 - 1 = x^2 - 2x + y^2 + z^2 = 0, \tag{13.31}$$

respectively. Figure 13.19 shows the x-z plane cross-section of the two surfaces. We show how to use Steps 1–4 above to map the intersection of $\mathbf{S}_1$ and $\mathbf{S}_2$ to a planar curve. (Of course, because the equations are so simple, we could have done this directly without following any fancy steps, but this is beside the point.)

Solution. Step 1 produces

$$\begin{aligned} F(x,y,z,w) &= 4x^2 + y^2 + z^2 - w^2 \\ G(x,y,z,w) &= x^2 - 2xw + y^2 + z^2. \end{aligned}$$

Step 2, based on the variable x, leads to

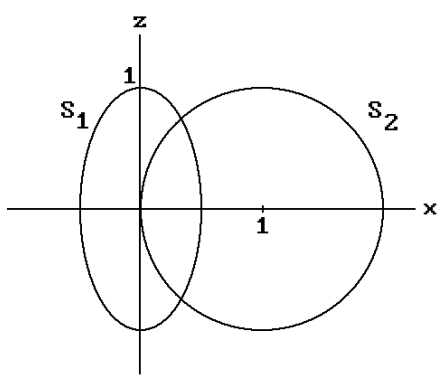


Figure 13.19. The ellipsoids of Example 13.5.5.1.

$$\begin{aligned} F_1 &= 4G - F = -8xw + 3y^2 + 3z^2 + w^2 \\ G_1 &= \left[(y^2 + z^2 - w^2)(x^2 - 2xw + y^2 + z^2) - (y^2 + z^2)(4x^2 + y^2 + z^2 - w^2)\right]/x \\ &= x(-3y^2 - 3z^2 - w^2) - 2w(y^2 + z^2 - w^2). \end{aligned}$$

Choosing the simpler equation $F_1 = 0$ to solve for x, we get

$$x = \frac{3y^2 + 3z^2 + w^2}{8w}.$$

Substituting this expression for x in G and dehomogenizing, that is, setting w to 1, gives us the equation

$$\left(\frac{3y^2 + 3z^2 + 1}{8}\right)^2 - 2\left(\frac{3y^2 + 3z^2 + 1}{8}\right) + y^2 + z^2 = 0. \tag{13.32}$$

Equation (13.32) defines the circle

$$y^2 + z^2 = 5/9$$

in the y-z plane. (To see this, let $u = y^2 + z^2$ in equation (13.32) and solve for u.)

Notice that we can run into serious problems if the wrong variable is chosen at Step 2. For example, choosing y we would get

$$\begin{aligned} F_1 &= G - F = -3x^2 - 2xw + w^2 \\ G_1 &= 3x^2y - w^2y + 2xwy \end{aligned}$$

and we are not able to solve for y. The reason that x worked and y does not is that the intersection projects nicely in a one-to-one fashion to the y-z plane using an orthographic projection parallel to the x-axis whereas projecting parallel to the y-axis collapses the intersection to a closed segment.

One problem with the above approach is that it has the potential to introduce extraneous factors and solutions. The intersection of $\mathbf{X}$ and $\mathbf{S}_1$ may be larger than the intersection of $\mathbf{S}_1$ and $\mathbf{S}_2$. See [Hoff89] for examples of this.

The Projection Approach. The idea here is to use a central projection from some point $\mathbf{p}$ to project the space curve to a plane curve. The only problem is choosing $\mathbf{p}$ correctly. We do not want the projection to introduce any singularities so that the map cannot be inverted. [Hoff89] describes a method that works with a high probability.

Step 1. Transform the surface equations by a linear transformation defined by a matrix with symbolic coefficients.

Step 2. Use the resultant to project the intersection.

Step 3. Substitute random values for the symbolic coefficients and check that the projection has the desired properties.

13.5.5.2 Example. We redo Example 13.5.5.1 using the projection approach.

Solution. We can skip Steps 1 and 3 this time because there are no singularities when resolving on x. Using the resultant which eliminates x from f and g, we get

$$R_x(f,g) = \det\begin{pmatrix} 4 & 0 & y^2+z^2-1 & 0 \\ 0 & 4 & 0 & y^2+z^2-1 \\ 1 & -2 & y^2+z^2 & 0 \\ 0 & 1 & -2 & y^2+z^2 \end{pmatrix}$$
$$= 9(y^2+z^2)^2 + 22(y^2+z^2) - 11.$$

This again defines a circle in the y-z plane. If we had tried to eliminate the y variable we would have a problem because

$$R_y(f,g) = \det\begin{pmatrix} 1 & 0 & 4x^2+z^2-1 & 0 \\ 0 & 1 & 0 & 4x^2+z^2-1 \\ 1 & 0 & x^2-2x+z^2 & 0 \\ 0 & 1 & 0 & x^2-2x+z^2 \end{pmatrix}$$
$$= (-3x^2-2x+1)^2.$$

We need Step 1 and 3 in general to make sure that there are no singularities in the projection. We were lucky when we chose x originally.

A major problem with algebraic approaches is that algorithms for finding solutions to high-degree polynomial equations are numerically unstable. It is also computationally expensive to compute resultants. By combining elimination theory with matrix computations the authors of [ManD94] tried to avoid these problems. Rather than finding roots of polynomials they had to find eigenvalues of matrices. Algorithms for finding eigenvalues are known to be stable. The algorithm in [ManK97] builds on this approach. Because one only needs the eigenvalues in a certain range, the new algorithm saves time by not computing those that are not needed.

13.6 Summary

In this chapter we have seen some of the difficulties involved in finding the intersection of objects. For that reason, many special cases have been studied extensively to get the best possible results. We have looked at some of these. Intersections of lines, conics, and quadrics are other special cases about which much is known. Here are a few of those and references to optimized algorithms for them:

Rectangular solids and convex polyhedra:	[Gree94]
Line and cylinder:	[Shen94]
Ray and cylinder:	[CycW94]
Line and cone:	[Shen95]
Ray and quadric surface:	[CycW92]
Parametric cubics:	[Klas94]

Some more references for intersections of quadrics are [OckS84], [GolM87], [Mill87], [OweR87], [SheJ87], [FaNO89], and [Gold83] (for quadrics of revolution). The special case of ruled surfaces is considered in [HeKE99]. [HMPY97] describes a robust interval analysis approach. [FaNO89] studies a class of degenerate quadric intersections that are rather common cases. Because the intersection curves in these cases admitted rational parameterizations, algebraic methods could be applied. Specifically, the approach described in [FaNO89] made use of both the Segre characteristic of a quadratic form (which is something determined from the multiplicities of roots to an associated polynomial) and the projecting cone of a space curve (which is the ruled surface determined by the pencil of lines through the origin and points on the curve) with multivariate polynomial factorization algorithms. An equivalent approach was described in [WilM93] that was somewhat simpler and facilitated the display of the intersection curves.

For cyclide intersections see [MaPS86] and [John93]. For formulas for the intersection of a ray with various objects see [Hain89] and [Hanr89]. More references for the general intersection problem can be found in [AbdY97] and [HosL93]. [MarM89] and [LuMM95] address the difficult tangential surface-surface intersection problem.

Note that nothing has been said about set operations on CSG objects. The reason is that there is nothing new to say in that regard. The fact is that if we apply set operations to two CSG objects, then we get another CSG object, so that the problem is dealt with in the context of the boundary evaluation algorithm for CSG. See Section 5.9.

Methods, such as marching methods, which use Newton-Raphson iteration somewhere along the line, are the most common. [BarK90] has a comparison of some marching methods. [DoSY89] compares marching and recursive subdivision methods. Pure marching techniques tend to be faster, because the Newton-Raphson method has a quadratic rate of convergence, but are liable to failure because they are very sensitive to local singularities. Pure recursive subdivision techniques need no starting points and tend to be more robust but are less efficient. They can produce an excessive amount of data for a fixed tolerance when singularities are present. [DoSY89] describes a method that is a combination of the two where one uses recursion to discover the geometry and iteration for accuracy. [AzBB90] compares the two methods for Bézier surfaces and shows that iteration is more accurate. An approach that combines marching and algebraic methods can be found in [KriM97]. It uses a matrix representation for the intersection curve.

The homotopy method described in Section 13.5.3 has not been used much in geometric modeling. Marching methods can actually be considered to be a kind of special case. The papers [LamM95] and [LamM96] compare the two methods, and the authors argue that the homotopy method has advantages over methods based on standard Newton-Raphson iteration. It also needs a "starting solution" but often has better convergence properties. Basically, the problem is that the basins of attraction for the Newton-Raphson method (the points to which one converges) tend to be fractal-like whereas the corresponding sets for homotopy methods are smoother. They are semialgebraic sets when one is solving algebraic systems. It is the chaotic nature of the Newton-Raphson method that causes its problems. The negative aspect of homotopy methods is that according to [LamM96] they are roughly 10–20 times slower than Newton-Raphson methods (**assuming** that the latter converge). The paper describes how homotopy curves are followed.

As mentioned above, getting the wrong connectivity of the intersection curve is a big problem. Three aspects of this problem are loop detection, jumping components, and incorrectly ordered components.

Loop Detection. The reason why it is good to have criteria for the existence of loops in the intersection is that if there are none, then the intersection must start in the boundary of the surfaces and one could then start with the simpler problem of intersecting a curve with a surface. Unfortunately, the flatness conditions in subdivision algorithms have a hard time dealing with loops. For example, consider the case of a large sphere and a plane. As the sphere approaches the plane, one passes from no intersection to the case where the intersection is a point and finally to where it is a circle. Locally the sphere is very flat and it becomes very hard to detect the circles when they are still small.

Jumping Components. In the case of tracing algorithms, one is trying to generate sequences of points, each of which describe one component of the intersection curve. It is not easy choosing a step size that will prevent the algorithm from jumping from one component of the intersection curve to another if they are very close. This would produce the wrong connectivity in the result.

Incorrectly Ordered Component. Similarly, one might accidentally jump from one part of the curve to another part of the same component if those parts passed close by each other. This would generate an incorrectly ordered sequence of points for that component.

[Hohm91] describes a loop detection criterion that can also be used to guarantee that a tracing algorithm does not jump components or generate points out of order. The criterion is based on properties of the Gauss map for the two surfaces. It isolates intersections with no loops quite well if the surfaces are relatively flat. An approach in case the loop criterion is not satisfied can be found in [KriM97]. [SedM88] describes a criterion that ensures that all branches of the intersection curve will be detected. An improved criterion is proved in [SeCK89]. [MaLe98] has an overview of approaches for loop detection with additional references and gives a topological criterion for the existence of loops in the intersection of two C^2 parametric surfaces.

[Wang92] also describes a way to deal with the topological inconsistency problems.

Finally, in this chapter we have been satisfied with simply getting the curve that is the intersection of two surfaces. In most algorithms one gets a polygonal approximation. There are times when one might want to get a higher-order approximation to the intersection. In that case it would be helpful to know the derivatives of the curve up to some arbitrary order k. Formulas for these and other geometric invariants such as tangents, curvature, and torsion, are derived in [YeMa99].

13.7 Programming Projects

1. Convex set intersections (Section 13.2)

   Implement and test Algorithm 13.2.1 for a world of rotated blocks.

2. Curve intersections (Sections 13.3.1, 13.3.3, 13.3.4)

   (a) Implement a ray-curve intersection algorithm as described in Section 13.3.1. Let a user define a world of curves and then specify various start points and rays interactively.
   (b) Implement a polygonal curve intersection algorithm for two user-defined polygonal curves. If they intersect, use an arrow to point at the intersection points.
   (c) Implement a curve-curve intersection algorithm as described in either Section 13.3.3 or 13.3.4. Let the user define two planar B-spline curves and then say whether they intersect or not. If so, use an arrow to point at the intersection points.

3. Surface sections (Section 13.4.3)

   Display the horizontal sections at a given level $z = h$, specified by the user, of all the surfaces in the world by drawing the intersection curves in some appropriate color.

4. Surface intersections (Section 13.5.3)

   Implement a surface-surface intersection algorithm. Let the user define two B-spline surfaces and then say whether they intersect or not. If so, draw the intersection curves in some appropriate color.

CHAPTER 14

Global Geometric Modeling Topics

Prerequisites: Basic calculus and vectors, Newton-Raphson method (Section 4.7 in [AgoM05])
Sections 1.9 and 3.12 in [AgoM05] (for Section 14.5.2)
Chapter 9 in [AgoM05] (for Sections 14.9 and 14.10)
Chapter 10 in [AgoM05] (for Sections 14.5.1)

14.1 Overview

This chapter takes a glimpse at a few properties of curves and surfaces that are global in nature. Section 14.2 looks at algorithms that compute various distances. Section 14.3 considers the so-far overlooked subject of how to best polygonize curves and surfaces. Section 14.4 describes an algorithm for displaying trimmed surfaces. Section 14.5 discusses implicitly defined curves and surfaces. Contour algorithms are described in Section 14.6. In Section 14.7 we describe some skinning algorithms. Section 14.8 describes how one can compute arc length and arc-length parameterization. Finally, in Sections 14.9 and 14.10 we briefly consider offset curves and surfaces and, more generally, envelopes. A major application of this subject is in milling operations using numerically controlled (NC) milling machines. These machines come in a variety of flavors distinguished mainly by the number of degrees of freedom of movement that they possess. The more degrees of freedom they have, the more complicated it is to describe their paths.

14.2 Distance Algorithms

This section deals only with distance algorithms between smooth objects with emphasis on parameterized objects. Formulas for basic linear objects were already presented in Chapter 6. Finding the distance between polygonal curves or faceted surfaces easily reduces to those cases.

We begin with some general observations that are the basis for all the algorithms that find the distance between two parameterized objects $\mathbf{O}_1$ and $\mathbf{O}_2$ in $\mathbf{R}^n$. Let $\mathbf{A} \subseteq \mathbf{R}^s$, $\mathbf{B} \subseteq \mathbf{R}^t$, and assume that $p:\mathbf{A} \to \mathbf{O}_1$ and $q:\mathbf{B} \to \mathbf{O}_2$ are parameterizations of $\mathbf{O}_1$ and $\mathbf{O}_2$, respectively. Define $h:\mathbf{A} \times \mathbf{B} \to \mathbf{R}$ by

$$h(\mathbf{u},\mathbf{v}) = (p(\mathbf{u}) - q(\mathbf{v})) \bullet (p(\mathbf{u}) - q(\mathbf{v})). \tag{14.1}$$

Finding the points of $\mathbf{O}_1$ and $\mathbf{O}_2$ that are closest is equivalent to finding the minima of $h(\mathbf{u},\mathbf{v})$. If $p(\mathbf{u})$ and $q(\mathbf{v})$ are closest points on $\mathbf{O}_1$ and $\mathbf{O}_2$, respectively, then $(\mathbf{u},\mathbf{v})$ must satisfy one of the following two properties:

(1) $(\mathbf{u},\mathbf{v})$ is a critical point of h in the interior of $\mathbf{A} \times \mathbf{B}$.
(2) $(\mathbf{u},\mathbf{v}) \in \partial(\mathbf{A} \times \mathbf{B})$, that is, $\mathbf{u}$ is on the boundary of $\mathbf{A}$ and/or $\mathbf{v}$ is in the boundary of $\mathbf{B}$.

Finding the minima of h is therefore a two-step process. First, the critical points of h are found by searching for zeros of the derivative of h. (We will always assume that functions are as differentiable as needed.) If we are not lucky to have formulas for these zeros, then any method that finds zeros of functions, such as the Newton-Raphson method, can be used here. We shall see in our examples below that having the derivative of h zero at some point $(\mathbf{u},\mathbf{v})$ corresponds geometrically to the saying that the vector $p(\mathbf{u}) - q(\mathbf{v})$ is orthogonal to the tangent planes of $\mathbf{O}_1$ and $\mathbf{O}_2$ at $p(\mathbf{u})$ and $p(\mathbf{v})$, respectively. This is intuitively what we would expect to have happen at two closest points. It generalizes the fact that the vector between two closest points on two lines is orthogonal to both of the lines. Of course, do not forget the possible special case where the objects intersect. The zero vector is orthogonal to everything.

Now the derivative of h is defined by the Jacobian matrix, which in turn is defined by the partials of $p(\mathbf{u})$ and $q(\mathbf{v})$. The exact form they take depends on the specific functions. Next, after having found any closest points on the interior of $\mathbf{A} \times \mathbf{B}$, we have to compare them with those on the boundary of $\mathbf{A} \times \mathbf{B}$. This is basically another problem of the same type but of one lower dimension.

Point-curve Distance. To find the distance between a point $\mathbf{p}$ and a parameterized curve $q:[c,d] \to \mathbf{R}^3$ we need to solve the equation

$$(\mathbf{p} - q(v)) \bullet q'(v) = 0 \tag{14.2}$$

for v. The left-hand side of equation (14.2) is what the derivative of the function h in equation (14.1) reduces to here. There may be more than one solution. For example, in Figure 14.1 both points $\mathbf{q}_1$ and $\mathbf{q}_2$ satisfy equation (14.2). Therefore, after finding all the solutions we need to check which of the solutions that lie in [c,d] correspond to the closest point. Finally, we need to compare the distance of that closest point to the distances from $\mathbf{p}$ to q(c) and q(d), the endpoints of the curve. Note how q(d) is actually the closest point on the curve in Figure 14.1.

Curve-curve Distance. Assume that two curves are parameterized by functions p(u) and q(u), respectively. To find points on these curves that are closest to each other we

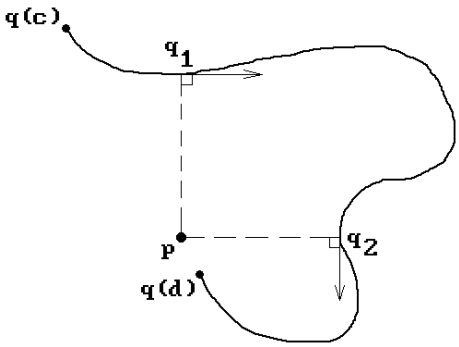


Figure 14.1. Distance from a point to a curve.

first find the critical points of the function h(u,v) in equation (14.1). These are the points where both of its partial derivatives vanish, that is, we must solve

$$\frac{\partial h}{\partial u} = 2(p-q)\bullet p' = 0,$$
$$\frac{\partial h}{\partial v} = 2(p-q)\bullet q' = 0. \tag{14.3}$$

Notice again that, geometrically, we are looking for points **p** and **q** on the curves with the property that either **p** = **q** or the line through **p** and **q** is orthogonal to the tangents of the curves at those points. See Figure 14.2(a). Finally, we also have to check the distances from each endpoint of a curve to the other curve.

14.2.1 Example. To find the distance between the curves defined by

$$p(u) = (u, u^2) \quad \text{and} \quad q(v) = (v, 2v+4).$$

Solution. See Figure 14.2(b). Equation (14.1) translates into

$$h(u,v) = (u-v, u^2-2v-4)\bullet(u-v, u^2-2v-4).$$

It is easy to check that the solutions to the equations

$$\frac{\partial h}{\partial u} = 2(u-v, u^2-2v-4)\bullet(1,2u) = 2[u-v+(u^2-2v-4)(2u)] = 0$$

and

$$\frac{\partial h}{\partial v} = 2(u-v, u^2-2v-4)\bullet(-1,-2) = -2[u-v+2(u^2-2v-4)] = 0$$

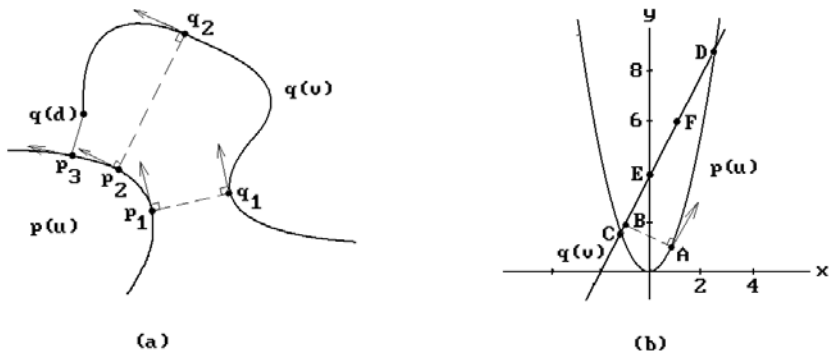


Figure 14.2. Distance between two curves.

are

$$(u,v) = (1,-1), (1-\sqrt{5},1-\sqrt{5}), \text{and } (1+\sqrt{5},1+\sqrt{5}).$$

The distance between **A** = p(1) = (1,1) on p(u) and **B** = q(−1) = (−1,2) on q(v) is a local maximum. The curves intersect at **C** = $(1 - \sqrt{5}, 1 - \sqrt{5})$ and **D** = $(1 + \sqrt{5}, 1 + \sqrt{5})$. We have not said anything about the domain of the curves. If it is **R**, then the distance between them is zero since they intersect. On the other hand, if, for example, the domain of p(u) is **R** but we had restricted the domain of q(v) to [0,1], so that q(v) would only trace out the segment [**E**,**F**] in Figure 14.2(b), then the values we just computed would not apply since they lie outside the domain of the current q(v). In fact, in a real algorithm we would have gotten the answer that h(u,v) has no critical points given our domains. To find the distance between p(u) and q(v) we would now have had to go on and check the distance of the endpoints **E** = q(0) = (0,4) and **F** = q(1) = (1,6) to p(u) and pick the closest distance.

Next, we move on to finding distances involving surfaces. If p(u,v) is a parameterization for a surface, let

$$n_p(u,v) = p_u(u,v) \times p_v(u,v).$$

$n_p(u,v)$ will be a normal vector to the surface at p(u,v).

Point-surface Distance. To find the distance between a point **p** and a parameterized surface

$$q : [a,b] \times [c,d] \to \mathbf{R}^3$$

we look for a point **q** = q(u,v) on the surface with the property that **p** − **q** is orthogonal to the tangent plane of the surface at **q**, that is, we need to solve the equations

$$\begin{aligned}(\mathbf{p} - q(u,v)) \bullet q_u(u,v) &= 0\\ (\mathbf{p} - q(u,v)) \bullet q_v(u,v) &= 0.\end{aligned} \tag{14.4}$$

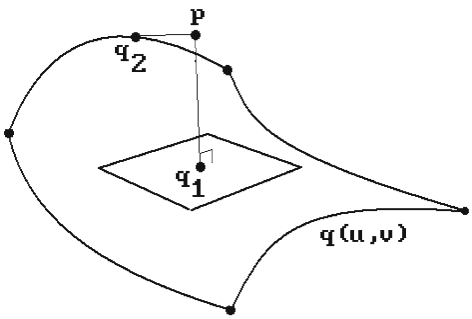


Figure 14.3. Distance from a point to a surface.

Equivalently, we can look for a **q** so that the vector **p** − **q** is parallel to the normal $n_q(u,v)$, that is,

$$(\mathbf{p} - q(u,v)) \times n_q(u,v) = 0. \tag{14.5}$$

If there are more than one solution to equations (14.4) or (14.5), then we need to check the distances between all those points and **p** and pick the closest. After that we will also have to find and check the distance from **p** to the four edge curves of $q|\partial([a,b] \times [c,d])$. The need for that is shown in Figure 14.3. Solving equations (14.4) and (14.5) would determine the point $\mathbf{q}_1$ in the figure, but the closest point is actually $\mathbf{q}_2$.

Curve-surface Distance. To find the distance between a parameterized curve

$$p : [a_1, b_1] \to \mathbf{R}^3$$

and a parameterized surface

$$q : [a_2, b_2] \times [c_2, d_2] \to \mathbf{R}^3$$

we need to solve the equations

$$\begin{aligned} &(p - q) \times (q_u \times q_v) = (p - q) \times n_q = 0 \\ &(p - q) \bullet p' = 0. \end{aligned} \tag{14.6}$$

See Figure 14.4. In addition, we also need to check the distances of $p(a_1)$ and $p(b_1)$ to $q(u,v)$ and the distance of $q|\partial([a_2,b_2] \times [c_2,d_2])$ to $p(u)$.

Surface-surface Distance. To find the distance between parameterized surfaces

$$p : [a_1, b_1] \times [c_1, d_1] \to \mathbf{R}^3 \quad \text{and} \quad q : [a_2, b_2] \times [c_2, d_2] \to \mathbf{R}^3$$

we need to solve the equations

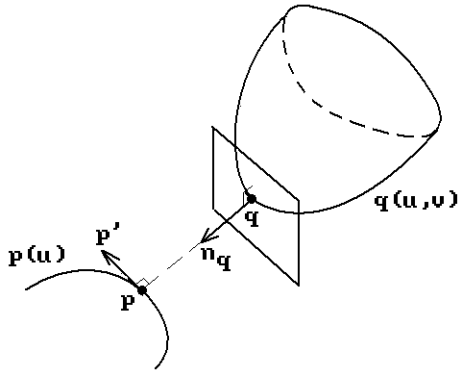


Figure 14.4. Distance from a curve to a surface.

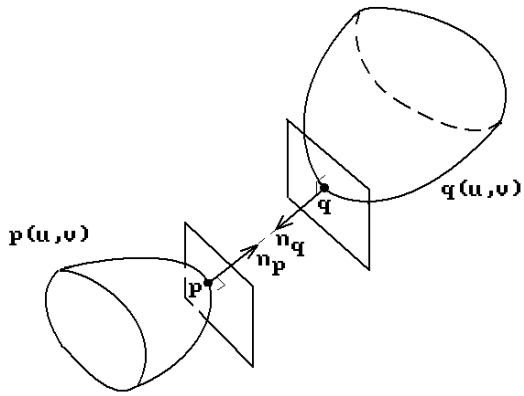


Figure 14.5. Distance between two surfaces.

$$(p-q)\times(p_u\times p_v)=(p-q)\times n_p=0$$
$$(p-q)\times(q_u\times q_v)=(p-q)\times n_q=0 \tag{14.7}$$

See Figure 14.5. Again we need to check distances for the boundaries, namely, the distance of $p|\partial([a_1,b_1]\times[c_1,d_1])$ to $q(u,v)$ and the distance of $q|\partial([a_2,b_2]\times[c_2,d_2])$ to $p(u,v)$. As an example of how one can improve the efficiency of these algorithms in special cases see [KimK03]. Kim describes algorithms for computing the distance between a canal surface and a plane, sphere, cylinder, cone, and torus.

Finally, for objects defined implicitly by equations $f(\mathbf{p}) = 0$ and $g(\mathbf{q}) = 0$ we must find the simultaneous solutions to these two equations.

14.3 Polygonizing Curves and Surfaces

In previous chapters we have already seen a number of algorithms that applied to smooth objects but which used linear approximations to these objects to accomplish

their tasks. For example, we pointed out that smooth objects are typically rendered by rendering collections of line segments for curves and collections of facets for surfaces. Some of the intersection algorithms were based on piecewise linear approximations. There are many other CAD/CAM applications where one wants or needs such approximations. Another is the grid or mesh representations needed for finite element models. In all such situations it is important that the shapes of the original objects were matched sufficiently faithfully. The reader might have gotten the impression that to accomplish this one simply has to subdivide enough in some straightforward way, but there is more to it than that. This section discusses some issues that have to be looked at when deciding on an approximation. In fact, we shall look at the general topic of how to "polygonize" objects. That term will mean representing an object by an organized collection of edges or facets, depending on the dimension of the object. The terms "tile," "tesselate," or "discretize" are sometimes used to mean the same thing as "polygonize." The term "tile" is especially popular when it comes to implicit objects.

Although there are many algorithms for polygonizing objects, they are not all equally good. Their worth depends on the particular application, but one cannot make choices if one does not understand the issues involved. Some properties that one usually wants the polygonized object to have are:

(1) One wants it to be a good approximation to the shape of the original object. Its points should be within a specified distance from the object.
(2) It should be efficient. Usually this means that we want to use as little space as possible to achieve property (1), but the time an algorithm takes to produce the approximation may also play a role. Ideally, the linearization should be adaptive with fewer edges or facets in places where the object is relatively flat.
(3) In the case of surfaces, one would like "well-shaped" facets. For example, facets that are thin slivers are bad for finite element methods.
(4) The approximation should not have "cracks" and have the same topology locally as the part of the object it is approximating.

We start our discussion of polygonization algorithms by looking at some that apply to parametric objects. In this case we really have two problems. First, we have to polygonize the domain of the parameterization and second, we often want to ensure that this polygonization will get mapped to a "good" polygonization of the actual object. By "good" one means that, in the case of surfaces, the approximating facets have a reasonable size and shape. As we just mentioned, long thin facets are considered bad. The hard part is the second problem because the domain of a parameterization is just an interval in the case of curves and typically a rectangle in the case of surfaces. The exception is trimmed surfaces, but those will be considered in Section 14.4. It would be nice if a good polygonization of the domain of a parameterization would induce a good polygonization of the object, but for that to happen, the parameterization would have to be close to a similarity transformation and that is rarely the case.

Suppose that $p:[a,b] \to \mathbf{R}^n$ defines a parametric curve. We can approximate the curve by a polygonal one by dividing its domain into k intervals $a = u_0 < u_1 < \ldots < u_k = b$ and use the polygonal curve defined by $\mathbf{p}_i = p(u_i)$. The common choice is to use the uniform subdivision $u_i = a + i\,(b - a)/k$. This may not be a good choice, however, for two reasons:

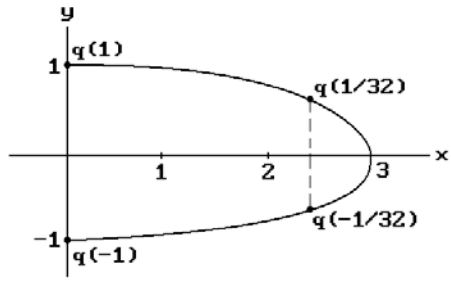


Figure 14.6. Problems with a uniform subdivision.

(1) The approximation may not be very good at places of high curvature along the curve. For example, consider the parabola

$$p(u) = (-3u^2 + 3, u), \quad u \in [-1,1],$$

in Figure 14.6. The substitution $u = v^{1/k}$, k odd, reparameterizes the curve to

$$q(u) = (-3v^{2/k} + 3, v^{1/k}).$$

Let k = 5 and suppose that we divide [−1,1] into 64 equal parts. Then

$$q(-1), \ldots, q\left(-\frac{1}{32}\right) = \left(\frac{9}{4}, -\frac{1}{2}\right), q\left(\frac{1}{32}\right) = \left(\frac{9}{4}, \frac{1}{2}\right), \ldots, q(1)$$

is not a good approximation to the curve even though we have a rather fine subdivision.

(2) If k is large to get a good fit, then the fact that we generate a lot of data may become a problem. If the curve has pieces that are essentially flat, then we might have saved a lot of space and effort by not subdividing those pieces as much. The extreme case is where the curve is a straight line segment and all we really need is p(a) and p(b).

These two reasons suggest that an adaptive subdivision would be more appropriate, where we subdivide highly curved segments more and flat ones less.

This raises the next issue, namely, how to define "flat." On an intuitive level, flatness over a segment [c,d] is often thought of as a measure of how much the curve deviates from the chord [p(c),p(d)].

Definition. The value

$$\max_{u \in [c,d]} \text{dist}(p(u), [p(c), p(d)])$$

is called the **chordal deviation** of p(u) over [c,d].

The chordal deviation of the curve in Figure 14.7 is the distance between the points **A** and **B**. On the other hand, if one is interested in preserving shape, then a better cri-

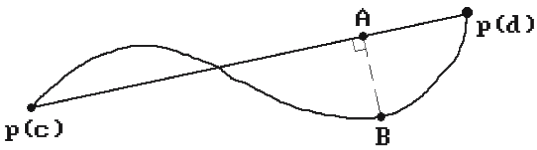


Figure 14.7. Chordal deviation.

Inputs: A curve $p : [a,b] \rightarrow \mathbf{R}^n$
A function Flat($\mathbf{p}_1$,$\mathbf{p}_2$,$\mathbf{p}_3$) that returns **true** or **false** depending on whether the three consecutive points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ pass some flatness test

Output: A sequence of points $\mathbf{p}_i$ that define a polygonal approximation to the curve p(u)

```
begin
    Output (p(a));
    Sample (p,a,b,p(a),p(b));
end;

procedure Sample (curve p; real c, d; point pc, pd)
begin
    real s;
    point ps;

    s := random number in (c,d);
    ps := p(s);
    if Flat (pc,ps,pd)
        then    Output (pd)
        else
            begin
                Sample (p,c,s,pc,ps);
                Sample (p,s,d,ps,pd);
            end
end;
```

Algorithm 14.3.1. Adaptive curve subdivision algorithm.

terion for flatness should be based on the curvature function for the curve, but to compute that would involve computing the first and second derivatives. De Figueiredo ([Figu95]) describes a recursive algorithm, Algorithm 14.3.1, that generates a good adaptively sampled polygonal approximation to the curve p(u) but takes less work. What the algorithm does is check to see if the curve is flat over the input domain by calling a function Flat. If so, then it approximates the curve with the segment defined by the endpoints of the curve segment; otherwise, it divides the domain into two and recursively calls itself over the two subdomains.

The flatness test for the curve p(u) over a subinterval [c,d] carried out by the function Flat is based on picking some $s \in (c,d)$ and checking for one of the following properties of the sequence of three points $\mathbf{p}_c = p(c)$, $\mathbf{p}_s = p(s)$, and $\mathbf{p}_d = p(d)$:

(1) The triangle $\mathbf{p}_c\mathbf{p}_s\mathbf{p}_d$ has small area.
(2) The angle between the vectors $\mathbf{p}_s\mathbf{p}_c$ and $\mathbf{p}_s\mathbf{p}_d$ is close to 180 degrees. Equivalently, the dot product $\mathbf{p}_s\mathbf{p}_c \bullet \mathbf{p}_s\mathbf{p}_d$ is close to –1.
(3) The point $\mathbf{p}_s$ lies close to the segment $[\mathbf{p}_c,\mathbf{p}_d]$.
(4) $|\mathbf{p}_c\mathbf{p}_s| + |\mathbf{p}_s\mathbf{p}_d|$ is close to $|\mathbf{p}_c\mathbf{p}_d|$.
(5) The tangent vectors of p(u) at c, s, and d are approximately parallel.

The five tests were found to be equally effective, with the first having the edge because no square roots are involved. The last test is the slowest because it involves the derivative of the curve. A tempting choice for s might be the midpoint (c + d)/2. The potential problem with that choice is that it may cause bad sampling, as in the case of uniformly undulating curves like a sine curve. De Figueiredo therefore suggests choosing s randomly but biased to the midpoint. In fact, a uniform distribution on [c + 0.45(d – c),c + 0.55(d – c)] was found to be satisfactory. Two other approaches are described by Filip et al. ([FiMM86]) and Lindgren et al. ([LiSH92]). The first uses bounds on the derivatives to guarantee a specified accuracy of the linear approximation and the second is based on an automated heuristic test for flatness.

Next, we consider parametric surfaces. Uniform subdivisions have the same problems that one ran into in the case of curves. One would again like to subdivide in an adaptive manner based on the local curvature but this is a much more complicated concept mathematically than for curves. One can choose from the "easier" curvature functions such as principal curvatures, Gauss curvature, or mean curvature, but even these would take a lot of work to compute. A simpler approach is described by Lichten and Samek ([LicS87]) that applies to Bézier, B-spline, and special bicubic Coons surfaces. The idea is to compare the distance between adjacent control points with the arc length of the curve segments between these points and requiring that the ratio should be in a specified range. If not, the midpoint of the curve is used to add another control point. When one splits, one has the choice of splitting the whole row (or column) at that parameter or one can split individual patches. With the latter choice one has to watch out that no cracks appear in the surface. Preventing cracks is something one often has to worry about when subdividing. [HEFS85], [BaDD87], and [DehZ91] describe ways to avoid the problem. One problem with the approximations in [LicS87] is that if the bad curvature occurs in the interior of a patch, then it will be missed. The authors considered a recursive triangulation based on adjacent normals being within a specified range but rejected this approach because of problems when surfaces had nonsmooth features.

Another adaptive subdivision algorithm is described by Filip ([Fili86]) and applies to surfaces defined by Bézier triangles. The subdivision test has three steps:

(1) Use a curve flatness test to see if the boundary curves are flat. If not all are, then subdivide the patch and repeat the test on the subtriangles.
(2) Use a surface flatness test and if that fails on the patch, then subdivide it and repeat all the tests on the subtriangles.

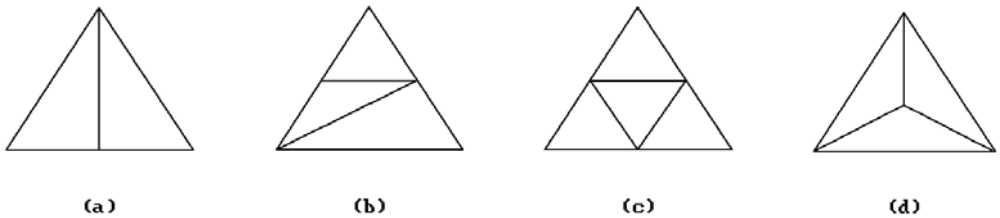


Figure 14.8. Possible triangle subdivisions.

(3) If the patch passes both the curve and surface flatness test, then output the planar triangle defined by the three corner vertices.

When subdividing triangles one has several choices. See Figure 14.8. The curve flatness test basically was to check if the Bézier control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ for any edge are sufficiently close to the segment $[\mathbf{p}_0,\mathbf{p}_n]$. The surface flatness test was similar. One checked if the Bézier control points for a patch with corner control points $\mathbf{q}_0$, $\mathbf{q}_1$, and $\mathbf{q}_2$ are close enough to the planar triangle $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$. Crack prevention was based on the approach in [BaDD87].

A further subdivision algorithm can be found in [FiMM86]. It determines the fineness of the subdivision based on computing bounds for the derivatives of the C^2 parameterizing functions, so that one can also specify the accuracy of the linear approximation. It produces right triangles in the parametric domain and cracks between neighboring patches. The subdivision is not adaptive, but the authors claim that it is several times faster than the adaptive algorithms in [LanR80] and [Fili86]. The adaptive algorithm by Herzen and Barr ([HerB87]) produced a triangulation using a quadtree for parameter space based on curvature bounds. One again got right triangles in parameter space and a potentially large number of triangles with cracks between patches.

We mention two more papers with a slightly different flavor. Cuillière ([Cuil98]) describes an automatic mesh generation algorithm useful for finite element mesh generation where one is given some a priori nodal density function that might have been obtained from some knowledge of the object's features. This approach differs from the more usual approach where one starts with nodes that are a crude approximation and then refines them as needed. Here we already have some information about where the nodes should be. Volpin et al. ([VSBJ98]) start with a smooth model and want to produce another model that is within a given tolerance of the original one but that is based on a simpler facet structure. The problem can be considered to be an example of the model simplification problem that we shall describe later. The algorithm in the paper first divides the input model into regions over which a discrete curvature related value is within a specified range. Next, a quadrilateral mesh is defined from those regions and finally a smooth surface is constructed for this mesh. The mesh is well suited for finite element analysis.

The majority of the early work on polygonizing curves and surfaces involved various types of recursive subdivisions of the parameter space. [HerB87] gives a good overview of this approach along with references. A quadtree type data structure was a major ingredient in the surface case. A more direct approach is described by Kosters

([Kost91]). Kosters deals both with curves and surfaces and defines what he calls *angle parameterizations*. These correspond to special reparameterizations that are computed numerically from the original parameterizations using their first and second derivatives. Using these parameterizations one simply uses a fine enough **uniform** subdivision of the domain of the parameterization to get the desired polygonization of the curve or surface. Kosters also discusses the advantages and disadvantages of his approach compared with recursive subdivision. For surfaces, two advantages are that it is easier and works better for boundaries. Recursive subdivision methods are more flexible.

We end the discussion of approximations to parametric curves and surfaces with some observations. The first relates to the amount of work that is involved in subdivision. Computing functions can sometimes be a lot of work, but in certain cases it can be done relatively efficiently. Specifically, subdivision of B-splines corresponds to knot insertion and subdivision of Bézier objects corresponds to de Casteljau subdivision. Second, a common approach to polygonization is to find a single step size, so that the linear approximation corresponding to the subdivision using that uniform step size is guaranteed to be within the desired tolerance over the entire domain of the parameterization. It is important here that this step size is not chosen too conservatively; otherwise, one does a lot of extra work subdividing more than would be necessary. Zheng and Sederberg ([ZheS00]) describe an algorithm for rational curves and surfaces that produces step sizes that are substantially larger than obtained by previous methods. Finally, many polygonization algorithms for surfaces work only for parameterizations with rectangular or triangular domains and do not work for trimmed surfaces. We shall consider that case in Section 14.4 and indirectly revisit the whole polygonization problem.

Next, we consider tilers for implicit objects but shall restrict ourselves to general comments. Sections 14.5 and 14.6 will describe some specific algorithms. Implicit tilers are also referred to as *isosurface generation algorithms* when the data is presented in the form of values of a function f and one wants to extract from that data the surface defined by an equation $f(\mathbf{p}) = c$. In general, at the top level the tilers in question end up dealing with data associated to the vertices of some sort of spatial partitioning. In other words, space has been divided into polygonal cells, typically squares or triangles in the two-dimensional case, and polyhedral cells, typically cubes or tetrahedra in the three-dimensional case. The tilers fall into two classes: *discrete tilers* that deal with discrete data and *continuous tilers* that deal with continuous data.

By discrete data we mean data obtained either experimentally, such as medical data from MRI or CT scans, or from volumetric computations, such as discrete grids from fluid flow simulations. The data typically corresponds to contour data for some unknown function. In the continuous data case we assume that we actually have a (smooth) function whose zeros define the object, in addition to having a cell decomposition with data at the vertices.

In both the discrete and continuous data case the goal of the tilers is to determine a linear approximation to the curve or surface that the data is assumed to be specifying. Specifically, one wants to determine how the object intersects the edges and/or faces of the cells and create a polygonization of the object from that information. In the discrete data case one has to interpolate the given data at the vertices of the cells to find intersections. In the continuous data case, the intersection can be computed accurately. If the cells in our partition are of different sizes, then our tiler is called an

adaptive tiler. As usual, one must be careful to make sure the cell partition is fine enough so that our approximation is accurate, but not too fine so that we create too much data. A review of implicit tilers can be found in [Kalv92], [NinB93], and [Bloo97].

Algorithm 14.3.2 is an outline for the typical implicit tiler. We shall now explain it in more detail. Our discussion will concentrate on surface tilers. Curve tilers are similar but much easier. The first step is to divide space into cells. Bloomenthal ([Bloo97]) separates the approaches into three broad types: subdivision, enumeration, and continuation.

In the **subdivision approach** one builds an octree representation of the object (a quadtree for planar curves). The cells are usually cubes or tetrahedra because it is easy to subdivide them into cells of the same type. The **enumeration approach** applies to volume modeling situations where one starts with a large three-dimensional grid of data thought of as the values of some unknown function. The data could, for example, come from MRI or CT scans. The grid is then searched to find all the cells intersected by what would correspond to a contour surface of that function. This is the kind of situation to which the marching cube algorithm applied. The **continuation approach** is one based on incremental steps. One does not start out with a fixed subdivision. Instead, one needs a starting point for each component of our surface. One then marches out from that point generating an approximation to each component. Predictor-corrector type methods often work fine for curves. In that case one marches along the tangent line from the start point and uses some correction mechanism like the Newton-Raphson method to get back to the curve and a new point on

Step 1: Partition space into cells

Methods:	subdivision	–subdivide space and represent ob ject via a data structure like a quadtree or octree
	enumeration	– get a predefined grid of values with the surface corresponding to a contour
	continuation	– starting at certain points on the object expand the subdivision of it into cells in an incremental way

Step 2: Polygonize object

Method: Find intersection of object with the boundary of each cell and then "fill in" intersection with the interior of the cell.

Problem: Ambiguity of the topology of the intersection with a particular cell.

Strategies to overcome the problem:

Topology inference, preferred polarity, or cell decomposition

Algorithm 14.3.2. Outline for implicit tilers.

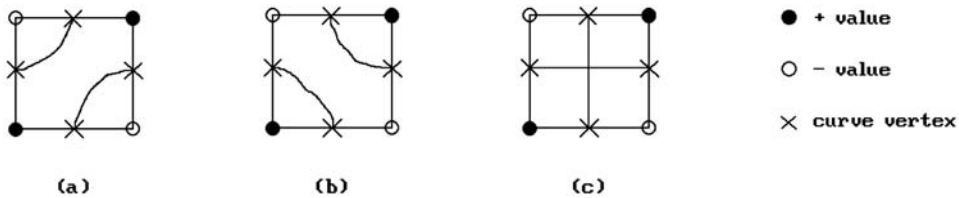


Figure 14.9. Ambiguous cell data.

the curve to use as the next start point. This approach becomes difficult when applied to surfaces. A more common approach is to determine which collection of tetrahedra ([AllS85], [AllG87], [AllG90], [AllG91]) or cubes ([WyMW86]) covers the surface. One again finds this collection by marching out from a given start point.

After one has the partition data, one tries to create the faceted approximation to the surface from it. Unfortunately, this is where one can run into ambiguity problems in general. The problem is that the cells may not satisfy an important condition, namely, that the intersection of the object and the cell is simple enough so that one can establish its topology. One first determines how the object intersects the boundary of the cell and then tries to "fill in" the part of the intersection that lies in the interior of the cell by some sort of interpolation. Typically, when working with a cell the only information that one has to determine the intersection are values at the vertices of the cell that say on which "side" of the object the vertex lies. The "side" is specified by the sign of the value. This is not always enough to determine the connectivity of the intersection in the cell interior. Figure 14.9 shows the problem in the case of curves. The "×'s" indicate the computed intersection of the curve with the boundary of the cell, but the given data does not allow one to know how to connect them. There are three legitimate interpretations.

Ning and Bloomenthal ([NinB93]) divide the methods that have been used to disambiguate into three types: topology inference, preferred polarity, and cell decomposition. The **topology inference** approach tries to disambiguate with some more data sampling or interpolation. For example, one could sample the data at the center of each cell in addition to the values at the corners. The **preferred polarity** approach tries to disambiguate by adding some rules on how to connect boundary values. For example, one could specify that "+" corner values should always be separated. This rule would choose Figure 14.9(b) over 14.9(a). The **cell decomposition** approach proceeds by subdividing the cell that generated the ambiguity until, hopefully, it disappears. Tetrahedral cells never have any ambiguity, so that one could subdivide any cubical cell into tetrahedra or one could have started with tetrahedra in the first place. Unfortunately, tetrahedral subdivisions generate much more data than subdivisions into cubes. [NinB93] shows that tetrahedral decompositions generate roughly twice as many triangles in the polygonized surface as would a cubical decomposition. On the other hand, subdividing a cube into smaller cubes means that one might have to recursively do this subdivision an arbitrary number of times before the ambiguity disappears.

The decompositions above do not have to have all the cells of the same size although this is the most popular choice because it is easiest to implement. There are

at least two reasons for using an adaptive tiler. One is that one would usually use less space. The other is that, like in the parametric object case, we may want to get a better approximation to the actual shape if its curvature varies substantially. See [Bloo97] for a discussion of this topic and for references.

Finally, Velho et al. ([VeDG99]) describe a polygonization algorithm that works for both implicit and parametric surfaces. The paper also provides an overview and references for many of the previous polygonization algorithms. Their algorithm is an adaptive one that creates a hierarchical mesh. The construction guarantees that there will be no cracks. There are three steps:

Step 1: Determine a rough triangulation of the surface.
Step 2: Sample the edges of the triangulation to produce a hierarchical approximation to the corresponding edge on the surface.
Step 3: Subdivide the triangles of the triangulation into smaller triangles based on the number of vertices in their boundary edges.

The only requirement in Step 1 is that the triangulation is faithful to the topology of the surface. For parametric surfaces this boils down to triangulating planar domains. The examples in the paper show that for parameterizations with rectangular domains, simply dividing the rectangle into two triangles with the diagonal will often be adequate. The algorithm however also works for trimmed surfaces. In the case of implicit surfaces, any one of a number of uniform polygonization algorithms can be used as long as the edges of the mesh lie in a tubular neighborhood of the surface. One such algorithm that does this is the one in [StaH97]. We shall describe it later in Section 14.5.2.

In Step 2 one makes a single pass over all the edges in the triangulation. To each edge one associates a data structure that consists of its two endpoints, a binary tree, and a real number that gives the maximum deviation of the edge from the surface. The binary tree comes from recursively replacing an edge $\mathbf{e} = [\mathbf{p},\mathbf{q}]$ with two new edges $\mathbf{e}_R = [\mathbf{p},\mathbf{r}]$ and $\mathbf{e}_L = [\mathbf{r},\mathbf{q}]$. The nodes in the tree consists of a vector $\mathbf{v}$ from the midpoint of $\mathbf{e}$ to $\mathbf{r}$, the maximum of the deviations of the edges $\mathbf{e}_R$ and $\mathbf{e}_L$ from the surface, and pointers to the tree nodes associated to $\mathbf{e}_R$ and $\mathbf{e}_L$. See Figure 14.10. Ideally, to get a nice balanced tree one would like $\mathbf{r}$ to be the midpoint of the curve corresponding to the edge, but since this would be too complicated to compute exactly, an approximation is used. In the end this produces an adaptive multi-resolution approximation to the curve corresponding to the edge.

Step 3 is carried out in a way that depends solely on the edge structure obtained from Step 2. First, the original edges of the triangulation are classified as simple or complex, depending on whether Step 2 subdivided them or not. Then the way a triangle gets subdivided depends on how many simple edges it has. The four cases are shown in Figure 14.11. If all three edges are simple, we do not subdivide. Otherwise,

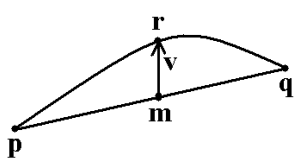


Figure 14.10. Edge sampling in [VeDG99].

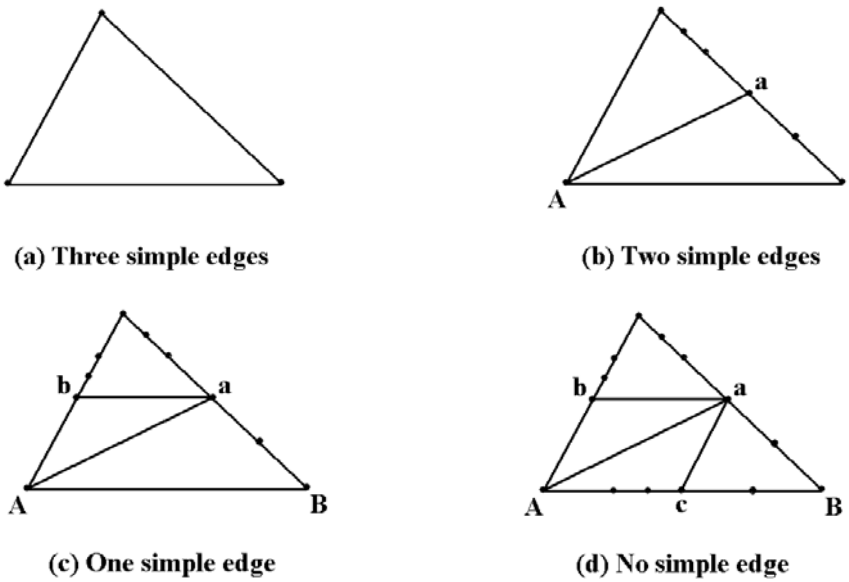


Figure 14.11. Subdividing triangles in [VeDG99].

we get two, three, or four subtriangles for the subdivision. There are several possible choices in the cases corresponding to Figure 14.11(c) and (d). A local optimization criterion is used to make a decision here. The choice will influence the quality of the final mesh and so it is worthwhile having a good criterion. The entire triangle subdivision process is again a recursive one.

Only Step 1 requires global knowledge about the surface. All the other steps use only local information. One advantage with this approach is that one gets a multi-resolution progressive structure. The authors mention applications to tolerance analysis, multi-resolution editing, level-of-detail rendering and compression, and progressive transmission and display. (*Progressive transmission* refers to being able to see better and better approximations of an object as it is incrementally received. See [Hopp96]).

We conclude this section with a brief mention of a closely related topic of research, namely, that of curve and surface simplification. Because models are being created with more and more detail, there are times when it is important to simplify them in order to save storage space, to save time in transmitting them from one place to another, or to speed up computations (the level of detail of a far away object does not have to be as high as one close up). Two good surveys of this subject can be found in [HecG97] and [Lueb01].

A general object simplification problem: Given an object with n vertices, find an approximating polygonized object with m vertices. An additional condition might be some error tolerance condition.

The simplification problem has two aspects: refinement and decimation. **Decimation** is the obvious interpretation of the problem. It is a "bottom-up" process where

we start with a polygonization and keep eliminating vertices until our polygonization is coarse enough. However, there is a reverse aspect. For example, in cartography one may have lots of data points (vertices) specifying a terrain and one would like a more compact polygonized surface representation of it. Similar situations arise in areas of scientific visualization and computer vision. Solving these problems is what one refers to as **refinement**. It is a "top-down" process and, in fact, is usually considered to include the polygonization problems for parameterized objects described earlier in the section. (Interpret a parameterized object as one defined by its points and thus defined by an infinite number of "vertices".)

The simplification problem is fairly well understood for curves, but less so for surfaces. One can sometimes avoid simplification by not generating excessive data in the first place by using an adaptive subdivision method. Algorithm 14.3.1, when applied in reverse, is a variant of what is commonly called the **Douglas-Peucker algorithm** that is used in cartography and scene analysis. We start with a polygonized curve and consider the segment from the first to the last point. If the other points are close enough to the segment we throw them away and replace the original curve with the segment; otherwise, we divide the curve into two and repeat the process for the two halves.

Multiresolution modeling is an area where one runs into the simplification problem. In this type of modeling one wants to be able to zoom in and out of levels of detail for the geometry. This is important for flight simulation and video games, for example.

14.4 Trimmed Surfaces

Surfaces are often designed in a piecewise manner. For example, one may start with a collection of surface patches and then add blends or fillets (see Section 15.6) between them to get the final global surface. This involves "trimming" away parts of the original patches. Also, when one performs set operations on solids, such as in CSG, the resulting surface patches are best described as trimmed patches. For that reason, the ability to represent trimmed surfaces is important for modelers and we now discuss some approaches to doing this.

Most parameterized surface patches have a parameterization function p(u,v) that has a rectangular domain. This means that the obvious way to display such a surface is to evaluate p(u,v) on a grid of parameter values in its domain and then to display the corresponding grid of lines, in the case of a wireframe display, or to use the grid rectangles or triangles as an approximation to the surface patch in some scan line or z-buffer algorithm. Trimming a surface patch involves restricting the domain of p(u,v) to an arbitrary subregion that will not be rectangular in general. The boundary of the subregion is called a *trimming curve*. In the context of blending, such curves are also called *contact* or *link curves*. Polygonizing trimmed regions is not so simple. It may get even more complicated if there is more than one parameterized patch involved and the trimming overlaps patches. A number of algorithms have been developed over the years to deal with such issues. We shall begin with a brief overview of some of the existing algorithms. Keep in mind though that, in the end, all the algorithms want a polygonization of the trimmed surface that has all the same properties that we

wanted for the polygonizations described in Section 14.3. The algorithms therefore need to be judged with respect to those properties in addition to any new constraints.

One early algorithm for representing trimmed surfaces is described by Rockwood et al. in [RoHD89]. The main steps for this algorithm are:

(1) Convert the given surface representation to a grid of Bézier patch representations. Each trimming curve is converted into a Bézier or linear trimming curve. A NURBS surface and its NURBS trimming curves are converted by means of knot insertions and change of basis. If trimming regions cross patches, they are divided so that they can be dealt with on a patch-by-patch basis.

(2) Subdivide all the trimming regions for a patch in (u,v)-parameter space into uv-monotone regions with respect to the u- or v-axis, where a region is said to be *uv-monotone* with respect to an axis if every line perpendicular to that axis intersects the region in a convex set, that is, an interval. Figure 14.12(a) shows four such uv-monotone regions.

(3) Each uv-monotone region is then further subdivided into rectangles in the interior and a triangular coving along its boundary. The size of the rectangles, specifically the step sizes s_u and s_v in the u- and v-direction, respectively, is determined by ensuring that each rectangle projects into a sufficiently small region in screen space. The user specifies how small the screen space regions should be. See Figure 14.12(b).

(4) Each trimming curve in parameter space is approximated by a polygonal curve whose vertices lie on the curve. The spacing of the vertices is determined from the parameterization of the curve and from the size of the rectangles if the curve lies in the interior of the patch. The vertices and the rectangles determine the triangular coving near the boundary of a region. The manner in which this is done guarantees that adjacent parametric patches will not generate any cracks because adjacent patches will always use the same vertices along their common boundary even if the rectangle size is different in each patch.

The step sizes s_u and s_v are based on curvature bounds for Bézier functions and are view dependent. The uv-monotone regions are determined by finding local maxima and minima of the trimming curves using a root finder. Problems with this approach are that the algorithm for finding uv-monotone regions is complicated and the coving may produce undesirable triangles. In addition, although it prevented cracks between

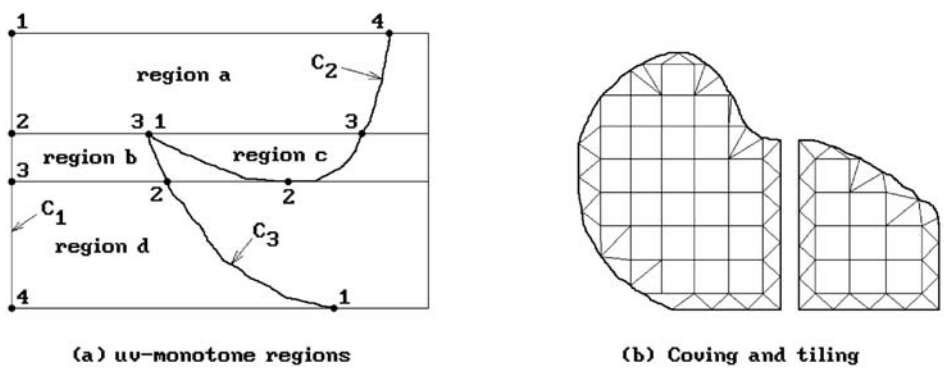


Figure 14.12. From the algorithm in [RoHD89].

Bézier subpatches for a single original surface, it did not prevent cracks between distinct trimmed surfaces. For example, the surface of the union of two solids might lead to two trimmed surfaces and the algorithm applied to each might not produce a common boundary since each surface may have used a different definition for that boundary curve.

Kumar and Manocha ([KumM95] and [KumM94]) describe an algorithm that is similar to the one in [RoHD89]. NURBS surfaces and NURBS trimming curves were converted into a sequence of Bézier representations because Bézier representation makes some computations simpler than they would be for B-spline representations. One also gets better bounds on derivatives and curvature. The curves are polygonized and a triangulation of the trimmed surface is generated via uniform subdivisions. The uniformity greatly simplifies the work. What one needs therefore is to determine the u- and v-step sizes for the surface tessellation and the step sizes for the trimming curves. Two possible criteria for finding these are:

The Deviation Criterion: The triangles should approximate the surface and their image in screen space should not deviate by more than a user specified bound from the surface. This involves second derivative bounds.

The Size Criterion: The triangles should have a reasonable size with their edges in screen space shorter than a predefined user tolerance. This only involves first derivative bounds.

Even though the size criterion may not work well on small patches that have a large variation in their curvature, it was used in [KumM95] because it is expensive to compute second derivatives for rational surfaces. uv-Regions were not used and coving was only done in rectangles that are intersected by trimming curves. The algorithm was simpler than the one in [RoHD89] and produced fewer triangles. Cracks and singularities were avoided not only between patches but also between surfaces. To avoid cracks between surfaces one considered the trimming curves in $\mathbf{R}^3$ and, once the matching surfaces were found, one used only one representation for both curves. The rendering algorithm described in [KumM94] computed the polygonization dynamically based on the viewing parameters, used back-patch culling (an approximation to the normal was used for efficiency), and made use of spatial and temporal coherence between frames.

The algorithm by Luken in [Luke96] is another that tried to avoid some of the problems that arose in [RoHD89]. In the Rockwood et al. algorithm, a surface was divided into patches and the trimming regions intersected with each patch to get a new collection of subpatches and trimming curves for them. See Figure 14.13(a). Because each subpatch was rendered independently, one had to do extra work so that no cracks appeared between patches. The coving done by [RoHD89] was avoided in [Luke96] by not dividing a surface into subpatches but defining a subdivision grid for the entire surface. One polygonized the trimming curves once for the entire surface and did not have to find intersections with subpatch boundaries. The uv-domain of the surface was divided into v-intervals that produced horizontal slices over the whole u domain. See Figure 14.13(b). Trimming polygons are clipped to these slices. A uniform subdivision of the u-parameter then subdivides each slice into rectangles, each of which is then handled separately, although the handling of those introduced

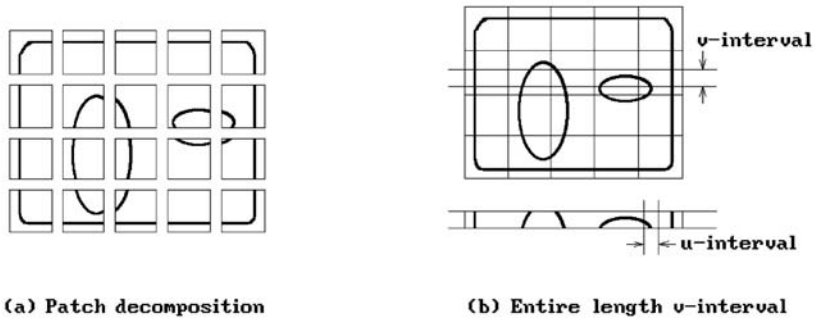


Figure 14.13. Differences between the algorithms in Rockwood, A., et al. "Real-Time Rendering of Trimmed Surfaces," SIGGRAPH 89, **23**(3), July 1989, 107–117. By permission of the publisher, ACM and Luken, William L., "Tessellation of Trimmed NURB Surfaces," CAGD, **13**(2), March 1996, 163–177.

some new complexities. However, there was no need for computing maxima and minima of curves or uv-monotone regions, as was the case in [RoHD89].

Sheng and Hirsh ([SheH92]) describe an algorithm for triangulating trimming surfaces that are grids of polynomial patches. The intended application was in stereolithography where one wanted to generate a "solid hard copy" directly from a three-dimensional CAD model. There the machines that carry this out need a very accurate triangulation. First, the trimming curves in parameter space were approximated by polygonal curves and merged to produce planar polygons with possibly holes. These polygons were triangulated using a special Delaunay triangulation algorithm and then this triangulation was refined until the edges of the triangles satisfied allowed tolerances. Cracks were avoided not only between patches of a single surface but also between surfaces. To avoid cracks between different surfaces one first determined the matching common boundaries (the distance between the boundaries must be sufficiently small) and then used a merging process to combine the two sets of vertices into one. One disadvantage with the approach is that the size of the triangles is determined by a global flatness bound for each patch that depends on the second derivative of the parameterization and is not adaptive, so that potentially a very large number of triangles are generated. The triangles one gets may also not have a desirable shape. A similar algorithm was described by Piegl and Richard in [PieR95]. Although it also used a global, but different, bound on the edge length of the triangles to ensure that the triangulation was within a user-specified tolerance, it had a special way to select the vertices of the triangles giving the triangles a more even size.

Many algorithms assume that surfaces are C^2. The Piegl and Tiller algorithm in [PieT98] only assumes a C^0 parameterization p(u,v) for a surface. Given a trimming tolerance ε_c and a triangulation tolerance ε_s, the algorithm outputs polygonal trimming curves and a triangulated trimmed surface. The polygonal approximation for each trimming curve $\gamma(t)$ in the domain of p(u,v) is within a tolerance of ε_c of the trimming curve $p(\gamma(t))$. The triangulated surface is within a tolerance of ε_s of the surface p(u,v). First, the trimming curves, which do not have to be closed curves, are linked together into lists that correspond to closed curves and are marked as being either inner or outer loops. Then they are polygonized. If $\gamma(t)$ is a trimming curve, its domain

is subdivided into subintervals $[t_j,t_{j+1}]$ with the property that the cubic curve which interpolates the four points $p(\gamma(t_j))$, $p(\gamma((2/3)t_j + (1/3)t_{j+1}))$, $p(\gamma((1/3)t_j + (2/3)t_{j+1}))$, and $p(\gamma(t_{j+1}))$ deviates from the chord $[p(\gamma(t_j)), p(\gamma(t_{j+1}))]$ by an amount that is between $(1/2)\varepsilon_c$ and ε_c. The lower bound $(1/2)\varepsilon_c$ is used to make sure that the polygonization does not consist of too many small segments. Next, the surface is subdivided into a grid of rectangular patches that are sufficiently flat using a modification of the algorithm in [LanR80] and [Peter94]. Since p(u,v) was only assumed to be C^0, one has no curvature information and so the flatness test is based purely on the flatness of the grid. Specifically, one checks how close corner vertices are to lying in a plane and how flat the boundary curves are. While subdividing one also tags the rectangles with respect to the polygonal trimming curves:

IN:	The rectangle is in one loop and outside all inner loops.
OUT:	The rectangle is out all outer loops or in one inner loop.
ON:	The rectangle intersects a loop.
OVER:	At least one loop is entirely inside the rectangle.
ONANDOVER:	There are some loops that are inside the rectangles and some that intersect it.

The OUT rectangles are discarded. See Figure 14.14(a). The tags OVER and ONANDOVER are needed in the case where many small punctures are made in a surface. A rather complicated algorithm now merges the trimming loops with the subdivision rectangles. One has to handle many special cases. See Figure 14.14(b). Finally, the polygonal regions are triangulated. See Figure 14.14(c).

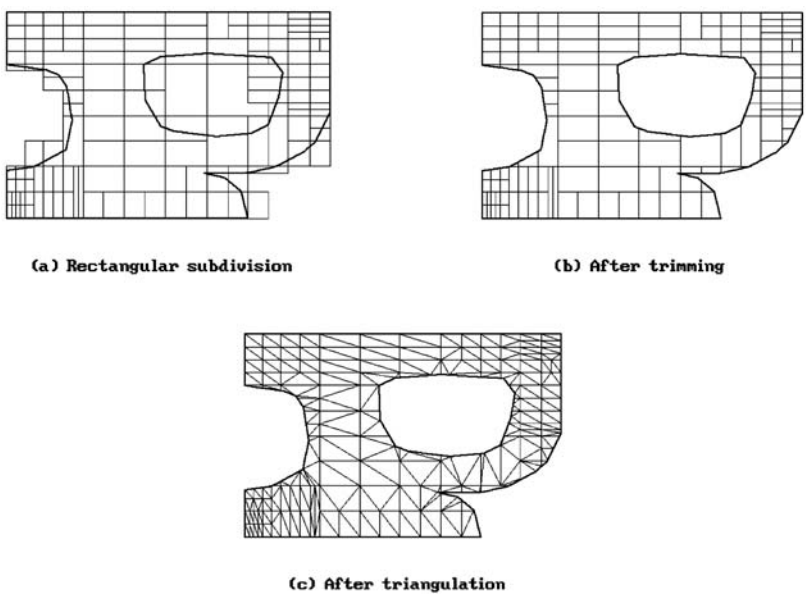


Figure 14.14. The Piegl and Tiller algorithm (Piegl, L.A., et al. "Geometry-Based Triangulation of Trimmed NURBS Surfaces," CAD, **30**(1), January 1998, 11–18. By permission of the publisher Elsevier).

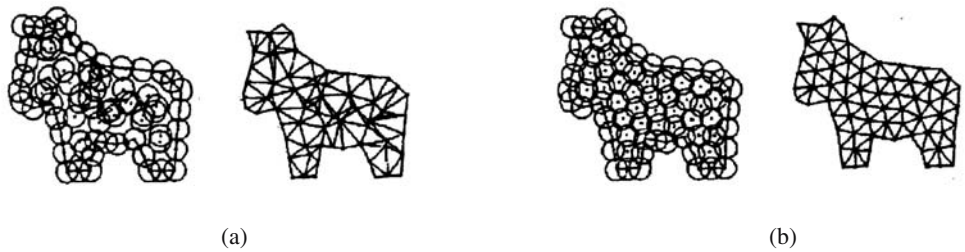


Figure 14.15. Shimada and Gossard bubble meshing (Modeling and Applications, Third ACM Symposium, May 1995, p. 416, © 1995 ACM, Inc. (Reprinted by permission).

A quite different algorithm for generating two- and three-dimensional triangular meshes for surfaces and solids that also applies to trimmed surfaces is described by Shimada and Gossard in [ShiG95]. The algorithm, called *bubble meshing* by the authors, consists of two steps. First, one defines a packing of bubbles or spheres with centers on the object. To minimize the gaps and overlaps, one makes an initial guess for the placement based on a hierarchical spatial subdivision. Proximity-based repulsive and attractive forces associated to the bubbles are defined and a physically based dynamic simulation searches for a force-balanced solution that involves adding and deleting bubbles depending on an overlap ratio.

After one has a packing, a constrained Delaunay triangulation or tetrahedrization is used to select the best tessellation. Figure 14.15(a) shows an initial **random** placement of bubbles **without** the use of a hierarchical subdivision and one can see that the induced triangulation has undesirable properties. Figure 14.15(b) shows the final bubbles and triangulation after the relaxation process. One motivation for the algorithm is that the centers of a tightly packed set of bubbles mimics the structure of a well-shaped Voronoi diagram, so that the number of poorly shaped triangles or tetrahedra is greatly reduced. The problem with the algorithm is that it is complex and the relaxation method is computationally expensive. It also applied only to a single patch and did not address the problem of cracks between patches.

Another adaptive trimming algorithm for a parametric surface p(u,v) is described by Vigo and Brunet in [VigB95]. Its focus was also on stereolithography applications. First, let $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ be any three points in the domain of p(u,v). Let $\alpha(u,v)$ be the standard linear map from the triangle $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ to the triangle $p(\mathbf{p}_1)p(\mathbf{p}_2)p(\mathbf{p}_3)$.

Definition. The triangle $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ is said to be *admissible* with respect to a tolerance ε if

$$|p(u,v) - \alpha(u,v)| \le \varepsilon$$

for all $(u,v) \in \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$. A triangulation of a region in the domain of p(u,v) is said to be an *admissible triangulation* if all of its triangles are admissible.

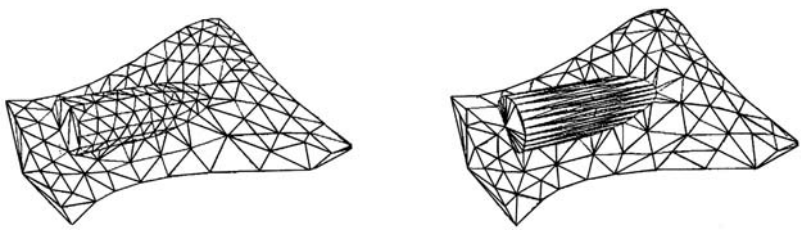
To test for admissibility Vigo and Brunet defined a function $R(\mathbf{p})$ based on bounds involving the second derivative of p(u,v) such that the following was true.

Admissibility Criterion. A triangle in the domain of p(u,v) is admissible if and only if $|\mathbf{pq}| \le \max(R(\mathbf{p}),R(\mathbf{q}))$ for each of its edges $\mathbf{pq}$.

For each patch, a quadtree structure is built using the function R(**p**). The boundary curves for trimming regions for each patch were then polygonized using this structure. These curves consisted of parts of trimming curves and possibly parts of boundary curves for patches. Cracks were prevented by defining a single polygonized curve for a common boundary. A sufficient number of vertices were then placed randomly in the interior of all the trimming regions. This number was determined by making an estimate of the number of vertices in an admissible triangulation. Finally, a relaxation method was used on the selected vertices to move them into vertices of an admissible triangulation. The method was different from the one used in [ShiG95] because the attraction-repulsion force approach was considered to have drawbacks and also could not guarantee that one got permissible triangles without additional testing. Instead, a constrained Delaunay triangulation of the interior vertices and the vertices of the trimming curves and any boundary curves was computed after each relaxation step and checked for admissibility. The relaxation step involved moving vertices if they belonged to edges that were too long or too short. Each offending edge determined a translation of the vertex and the actual move of the vertex was the vector sum of the translation vectors defined by all the incident edges. Care had to be taken so as not to move outside of the trimming region.

Anglada et al. ([AnGC99]) describe another variant of the algorithm in [VigB95]. The main difference between the two is that a different and often smaller set of triangles is generated. In [AnGC99] the triangles are determined using directional bounds. One takes directional aspects of curvature into account, so that if the curvature is small in one direction and larger in another one, then one get elongated triangles aligned with the first direction. For example, in the extreme case of a cylindrical shaped surface one would get thin triangles aligned with the axis of the cylinder and as long as it. Figure 14.16(a) and (b) show the difference in results after applying the algorithm in [VigB95] and [AnGC99], respectively.

The paper by Cho et al. ([ChPP98]) takes quite a different approach to the trimmed surface problem. One problem encountered by many of the algorithms described above is that they triangulate a surface **S** by triangulating the domain of a parameterization p(u,v). Even if one makes sure that the resulting triangulation is a good enough approximation, the map p(u,v) from parameter space to $\mathbf{R}^3$ inevitably introduces distortions in the triangles since it is not an isometry. The approach suggested in [ChPP98] is to reparameterize the surface so that the new parameterization is close



(a) (b)

Figure 14.16. Results of the algorithms in Vigo, M., et al. "Piecewise Linear Approximation of Trimmed Surfaces," in [HaFN95], 341–356 and Anglada, M.V., et al., "Directional Adaptive Surface Triangulation," CAGD, **16**(2), February 1999, 107–126.

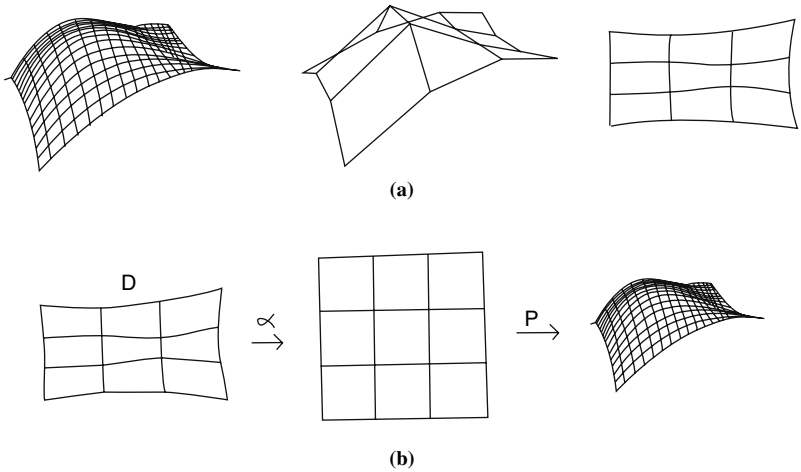


Figure 14.17. Tessellating surfaces via locally isometric approximations (Cho, W.P., et al. "Approximate Development of Trimmed Patches for Surface Tessellation," CAD, **30**(14), December 1998, 1077–1087.).

to a local isometry. Of course, this has to be done in a way that does not involve excessive computations. Starting with a rectangular grid of points on the surface, the authors construct a mapping from this grid to a grid G in the plane that is approximately a local isometry in the sense that edges adjacent to a vertex should be mapped to edges in the plane of approximately the same length. In Figure 14.17(a) we see an original surface, then a grid on that surface, and finally the grid G in the plane. Cho et al. define a global error function and use a minimization argument to accomplish this. Next, they define a one-to-one and onto linear map α from the region **D** defined by G to the rectangular domain of p(u,v). The map $\varphi(u,v) = p(\alpha(u,v))$ will be a parameterization of the surface with domain **D**. See Figure 14.17(b). A nice triangulation of **D** or the trimmed regions in **D** will then map to a reasonably nice triangulation of **S** by $\varphi(u,v)$.

Most algorithms that triangulate trimmed surfaces obtain the triangulation by triangulating the region bounded by the trimming curves in parameter space. This means that the trimmed surface problem has as part of it a problem that is interesting on its own, namely, the **triangulation problem** for such planar regions. This is a well-known problem in computational geometry and Section 17.6 discusses algorithms for triangulating planar polygons without holes and their complexity. On the other hand, since we are interested in the rendering of trimmed surfaces, we do not necessarily need triangles. We shall now describe an implementation of a relatively simple algorithm that solves the following general problem.

> **Trapezoidation problem:** Given a closed planar region bounded by k closed polygonal curves, $k \geq 1$, decompose it into a collection of trapezoids.

Of course, once a region has been divided into trapezoids, it is obviously easy to get a triangulation, so that we will also have solved the corresponding triangulation

problem. To use our algorithm to triangulate trimmed surfaces one would first have to use some other algorithm to represent the trimming curves in parameter space as a suitable collection of closed polygonal curves. Additional criteria would have to be used to determine if the triangulation is fine enough. It should also be noted beforehand that since our trapezoidation algorithm does not concern itself with the parameterization of the trimmed surface, the induced triangulation of the surface may contain very thin triangles, so that extra work would have to be done to avoid that. Nevertheless it leads to a quick-and-dirty algorithm for displaying trimmed surfaces and we shall describe the algorithm with that application in mind. (We should mention the algorithm in [ZalC99] that seems to be based on similar ideas; however, the author has used the algorithm we are about to describe for a number of years even though it was never published until now.)

Our trapezoidation algorithm is based on an idea from Vatti's clipping algorithm ([Vatt92]) that we described in Section 3.3.5. Recall that the central idea of his algorithm is to define polygons in terms of left and right bounds and then to obtain the final clipped regions using a fairly standard scan line approach with active edge lists. This idea of polygon bounds can also be used effectively to generate a trapezoidal decomposition of a polygonal region. The domain can be quite general for our algorithm. It may have holes or consist of more than one component.

The main task will be to show how to subdivide a given polygonal region into trapezoids using the left and right bounds of the bounding curves for the polygons of the region. (Trapezoids in this discussion will include the "degenerate" case of triangles.) Because of the application to trimmed surfaces, we also give an algorithm for subdividing a trapezoid appropriately for a wireframe display of the parametric surface defined on it. This would also apply to a z-buffer type display algorithm. The complete details for an algorithm that displays trimmed parameterized surfaces (above and beyond what is discussed here) depend on the type of display that is desired. As an example of one approach for shaded surfaces using a z-buffer algorithm see [RoHD89] but replace the regions that are used there with the trapezoids from here.

The discussion in this section will depend heavily on what was done in Section 3.3.5. To be able to handle our regions we can use a simplified version of the algorithm described in Section 3.3.5. Major simplifications arise from the fact that, although the region may consist of more than one polygon, none intersect here. On the other hand, we shall need to worry about "knots," so that the data structures will be modified slightly to fit our current needs. We shall concentrate the discussion on those aspects that are different from those in Section 3.3.5. Like in that section we shall only sketch the main ideas. A more thorough discussion can be found in the document VattiTrim on the accompanying CD.

Given our planar region, first determine the left and right bounds of the polygons making up the region and store this information in a local minima list (LML) as before. Next, compute the trapezoidal decomposition of the region from the LML by scanning the region from the bottom to the top using scan beams whose y-values are kept in a scan beam list (SBL). Recall that the values for this list are not generated all at once. We start with the list of all the local y-minima of the polygon boundaries and the y-values of the top endpoint of the edges that start at the local minima and then add to the list incrementally on an edge by edge basis. As we scan the world one scan beam at a time, an active edge list (AEL) lists all the edges intersecting the current scan beam just like in Section 3.3.5.

When we begin processing a scan beam, before we look at any of the edges of the AEL, we check the LML to see if any of its bound pairs start at this level. These bounds correspond to local minima. Each of these will start a trapezoid or break an existing one into two depending on whether the local minimum starts with a left-right or right-left edge pair. The first nonhorizontal edge from each bound of a local minimum is added to the AEL. Note that horizontal edges are never put on the AEL. Intermediate horizontal edges basically are skipped but they do create "knots" as will be described later. After any new edges from the LML are added to the AEL, we process the edges of the AEL one at a time starting from the left. See Algorithm 14.4.1 for a more precise description of the top-level algorithm described so far. Compare this algorithm with Algorithm 3.3.5.1.

```
{Global variables}
real list SBL;              {an ordered list of distinct reals thought of as a stack}
bound pair list LML;        {a list of pairs of matching polygon bounds}
edge list AEL;              {a list of nonhorizontal edges ordered by x-intercept
                              with the current scan line}
trapezoid list TRAPS;       {the trapezoids are stored here as algorithm progresses}

trapezoid list function ConvertToTrapezoids (curve list PL)
{The list PL represents a polygon with holes. The first curve in the list is the outer boundary
 of the polygon and the others, if any, are the holes.  The procedure partitions the polygon
 into trapezoids.  The list of these trapezoids is returned to the calling procedure.}
begin
    real yb, yt;

    Initialize LML, SBL to empty;

    {Define LML and the initial SBL}
    for each  curve P in PL  do  UpdateLMLandSBL (P);

    Initialize TRAPS, AEL to empty;

    yb := PopSBL ();                    { bottom of current scan beam }
    repeat
        AddNewBoundPairs (yb);
        yt := PopSBL ();                { top of current scan beam }
        ProcessEdgesInAEL (yb,yt);
        yb := yt;
    until  Empty (SBL);

    return (TRAPS);
end;
```

Algorithm 14.4.1. The trapezoid creation algorithm.

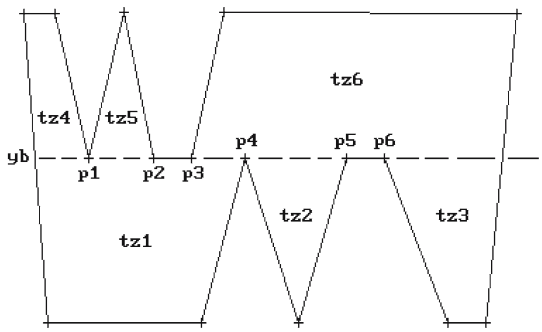


Figure 14.18. Complications for the AddNewBoundPairs procedure.

Like in Section 3.3.5, the UpdateLMLandSBL procedure finds the bounds of a polygon, adds them to LML, and also updates SBL. This time we will have an *adjacent trapezoid* associated to edges rather than an adjacent polygon. It may change as we move from one scan beam to the next. Because horizontal edges complicate matters, in order to make dealing with horizontal edges easier, we again assume that the matching left and right bound pairs in the LML list are normalized (see Section 3.3.5 for a definition).

Next, consider the AddNewBoundPairs procedure. The task of this procedure is to add those bounds that start at the current scan line into the AEL list. This is not as straightforward as it may seem. Some of the complicating possibilities are shown in Figure 14.18, which shows some of the events that can occur in the processing of the scan beam between the two scan lines at yb and yt. As we deal with local minima at the points **p1** and **p2**, we have to worry about potential "knots" for either new or old trapezoids. By a *knot* for the top or bottom edge of a trapezoid we mean the x-coordinate of a vertex of a polygon that must be included in any subdivision of the trapezoid. Knots are needed because adjacent trapezoids might have only part of an edge in common. Without keeping track of the knots, subdivisions of these adjacent trapezoids might subdivide the part of the edge that they have in common in different ways that would lead to gaps in the surface defined over these trapezoids. In Figure 14.18, when we process the local minimum at **p1** we have to add a top knot to trapezoid tz1. When we process **p2**, we must add two top knots corresponding to **p2** and **p3** to tz1. The knots corresponding to points **p4**, **p5**, and **p6** in the bottom edge of trapezoid tz6 would have been specified when the scan beam below the current one was processed and the local maxima were encountered. Only knots **interior** to an edge are kept track of in the trapezoid data structure. Although the endpoints of the top and bottom edges of a trapezoid clearly fall under the label of knots, we do **not** include them in the knot arrays, because that would be redundant data.

After these comments, assume that the trapezoids use the data structure shown in Data 14.4.1. The leftDx and rightDx fields are simply the increments by which leftX and rightX changes as we move from one scan line to the next. Here is what we have to do when we add the edges of a bound pair to the AEL list. In general, at a **left-right** edge pair minimum, we

```
trapezoid = record
    real        leftX,        { the x-coordinate of the bottom left vertex }
                leftDx,       { the reciprocal of the slope of the left edge }
                rightX,       { the x-coordinate of the bottom right vertex }
                rightDx,      { the reciprocal of the slope of the right edge }
                bottomY,      { the y-coordinate of the bottom edge }
                topY;         { the y-coordinate of the top edge }
    real array  topKnots,     { the interior knots for the top edge }
                botKnots;     { the interior knots for the bottom edge }
end;
```

Data 14.4.1. The trapezoid data structure.

(1) create a new trapezoid T with all fields defined, except that at this point it is degenerate because the **topY** and **bottomY** fields are temporarily set to the same value, and
(2) make T be the adjacent trapezoid of the two edges.

A local minimum at a **right-left** edge pair is more complicated. Basically, the trapezoid below us is about to split. First of all, let us introduce some notation. Call a trapezoid T *closable* if

(1) its **topY** and **bottomY** fields have the same value, and
(2) the common value of these two fields is less than the y-coordinate of the bottom of the current scan beam.

The process of setting the **topY** field to a value larger than the value of the **bottomY** field (thereby making the trapezoid "nondegenerate") will be referred to as *closing the trapezoid*. Sometimes when a trapezoid is closed, we need to update its **rightX** and **rightDx** fields. A trapezoid whose **topY** and **bottomY** fields have distinct values is said to be *closed*. Closed trapezoids may still have incomplete knot arrays however.

Dealing with a **right-left** edge pair (**e1**,**e2**) therefore involves three steps. Let **prevE** be the edge to the left of **e1** and let **belowTrap** be the trapezoid below this one which is splitting. Let BottomX(**e**) of an edge **e** denote the x-coordinate of the intersection of **e** with the horizontal line at the bottom of the current scan beam.

Step 1: If the trapezoid adjacent to **prevE** is still open, then close it and create a new trapezoid T1 for **e1** and **prevE**. For example, in Figure 14.19(a), when we get to **p1**, tz1 would be adjacent to **e1**. We would close it and create tz2 for **e1** and **e3**. On the other hand, when we get to **p2**, we do **not** need to close tz3.

Step 2: Create a new open trapezoid T2 and make it the adjacent trapezoid for **e2** and the edge that follows **e2**. Move any knots from the bottom of **prevE**'s trapezoid that are bigger than BottomX(**e2**) to T2. For example, in Figure 14.19(b), tz4 would already have bottom knots corresponding to **p4**, **p5**, and **p6** as we start processing the scan beam. At **p1** (and **p2**) the knots **p4**, **p5**, and **p6** would be shifted.

Step 3: The new local minimum at BottomX(**e1**) creates a knot in the top of **belowTrap**. See point **p1** in Figure 14.19. If BottomX(**e2**) is larger than BottomX(**e1**), then we get a second knot. See edge **p2p3** in Figure 14.19(b). These knots must be added to the top knots of **belowTrap**.

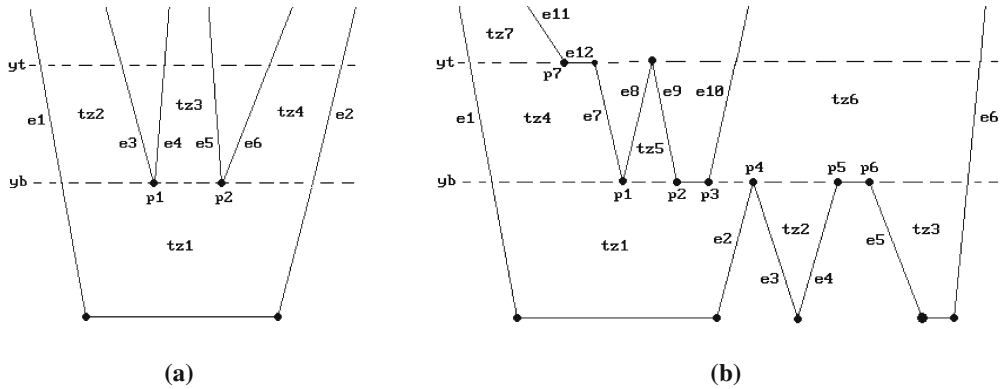


Figure 14.19. Closing and splitting trapezoids.

Table 14.4.1 Stages of the AddNewBoundPairs procedure.

	AEL	Trapezoid values
At beginning	{**e1**,**e6**}	tz4 (x1b,dx1,x6b,dx6,yb,yb,{},{xp4,xp5,xp6}) tz5, tz6, and tz7 not yet defined topKnots of tz1: {}
After (**e7**,**e8**) pair	{**e1**,**e7**,**e8**,**e6**}	tz4 (x1b,dx1,xp1,dx7,yb,yb,{},{}) tz5 (xp1,dx8,x6b,dx6,yb,yb,{},{xp4,xp5,xp6}) tz6 and tz7 not yet defined topKnots of tz1: {xp1}
After (**e9**,**e10**) pair	{**e1**,**e7**,**e8**,**e9**,**e10**,**e6**}	tz4 (x1b,dx1,xp1,dx7,yb,yb,{},{}) tz5 (xp1,dx8,xp2,dx9,yb,yb,{},{}) tz6 (xp3,dx10,x6b,dx6,yb,yb,{},{xp4,xp5,xp6}) tz7 not yet defined topKnots of tz1: {xp1,xp2,xp3}

By the way, keeping track of the trapezoid below the current position in the scan line is not hard, because trapezoids are added to TRAPS in a bottom-up, left-to-right manner, so that we merely need to keep moving a pointer to the right appropriately.

One determines whether an edge is a left or right edge at the time that it is put on the AEL. This is done by a parity-type argument. If there are an even number of edges on the AEL to the left of the new edge, then this edge is a left edge, otherwise, it is a right edge.

Figure 14.19(b) and Table 14.4.1 should clarify how the AddNewBoundPairs procedure works. We use the notation shown in Figure 14.19(b) and the following abbreviations:

xpi is the x-coordinate of point **pi**.
dxi is the value of the dx field in edge **ei**.
xib is the x-coordinate of the point where edge **ei** meets scan line at yb.
xit is the x-coordinate of the point where edge **ei** meets scan line at yt.

Because the trapezoids tz2 and tz3 are already defined and do not change, we do not include them in the table.

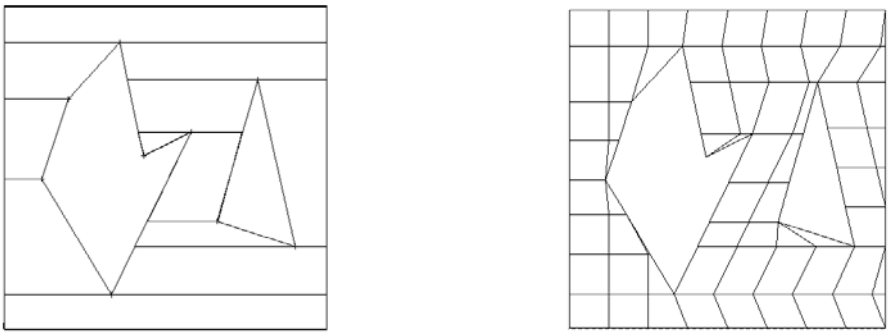
Next, we describe the AEL processing procedure. After adding any bottom edges from local minima to the AEL, we process the edges of the AEL one at a time starting from the left. If an edge extends past the current scan beam, then we simply update the x-intersection field of the edge. If the edge ends at the top of the current scan beam, then one or two trapezoids are closed and the edge is processed in one of two ways depending on whether the top vertex is an intermediate vertex or a local maximum. If the vertex is an intermediate vertex, then the edge is replaced in the AEL by its successor edge and the left/right flag is passed on to the new edge. In the case of a left intermediate vertex, the adjacent trapezoid is closed. In the case of a right intermediate vertex, we close the trapezoid to its left if it is still open. If the top vertex is a local maximum, then two bounds meet that may belong either to the same or to different trapezoids. If the bounds belong to the same trapezoid, that is, the edge **e** was a left edge, then the trapezoid adjacent to **e** is closed. If the bounds belonged to different trapezoids, then two trapezoids (one to the left of **e** and the other to the right of the second bound) must be closed and a new merged trapezoid T started. Knots are a complicating factor in this last case. The top endpoint of **e** will contribute a knot to the **botKnots** array of trapezoid T. We must also check if our local maximum has a top horizontal edge because then a second knot will be added to the same **botKnots** array. Table 14.4.2 gives an example of these steps using Figure 14.19(b) again. Figures 14.20(a) and 14.21(a) show some sample polygons and their final trapezoidal decompositions.

Returning to the topic of trimmed surfaces, once we get a trapezoidal decomposition of their domain as described above, there is still work to be done when it comes to displaying them. For one thing, the trapezoids may be large and may need to be subdivided. Any subdivision has to take the knots into account to avoid gaps in the image. We shall describe a scan line approach that would also be suitable in the context of a z-buffer or related type display. Basically, we need to come up with a convenient way to subdivide the trapezoidal u-v domain appropriately.

Assume that du and dv are the maximum u and v step sizes, respectively, that we are to use. Assume that trapezoids have a Boolean **doBottom** field that specifies whether or not the bottom edge of the trapezoid is to be drawn (we always draw the top edge). The reason for this is that if two trapezoids are adjacent, then we might not want to draw their common edge twice. Because of the way that trapezoids are

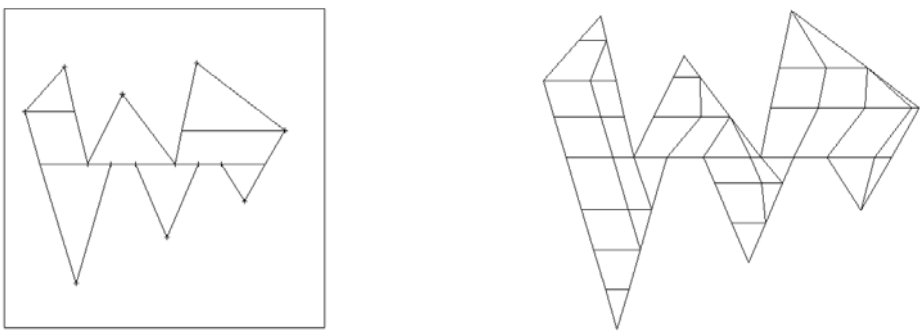
Table 14.4.2 Stages of the AEL processing procedure.

	AEL	Trapezoid values (topY,topKnots)	Edges (x,adjacent trapezoid)
At beginning	{**e1**,**e7**,**e8**,**e9**,**e10**,**e6**}	tz4 (yb,{}) tz5 (yb,{}) tz6 (yb,{})	**e1** (x1b,tz4)
After **e1**	{**e1**,**e7**,**e8**,**e9**,**e10**,**e6**}	no change	**e1** (x1t,tz4)
After **e8**	{**e1**,**e11**,**e9**,**e10**,**e6**}	tz4 (yt,{xp7}) tz7 (yt,{})	**e1** (x1t,tz7)
After **e10**	{**e1**,**e11**,**e10**,**e6**}	no change	**e10** (x10t,tz6)



(a) (b)

Figure 14.20. Trapezoidal decompositions of polygons.



(a) (b)

Figure 14.21. More trapezoidal decompositions of polygons.

generated they never have common side edges. Incorporating this **doBottom** field in the trapezoid generation algorithm is easy. It is set to true for all trapezoids whose bottom edge comes from a local minimum horizontal edge or that meet a local maximum horizontal edge.

Given a trapezoid we first want to subdivide it into horizontal slices that are at most dv units high and then to subdivide each of these slices into rectangular or triangular patches whose top and bottom edges are at most du units wide. Furthermore, any knots that the trapezoid may have must appear as vertices for the patches. Getting the horizontal slices is easy. The tricky part is to divide those slices vertically in a way that will include the knots in the division. One can do this by dealing with various cases separately. The three types of generic cases are shown in Figure 14.22 and one basically divides the trapezoids along the dotted lines shown in the figure, except that the existence of knots may mean that the actual lines along which the trapezoids are divided may be a slightly perturbed version of those. In the end, drawing a slice is

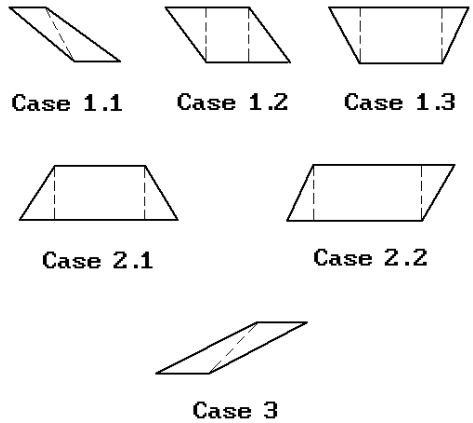


Figure 14.22. Cases when decomposing horizontal trapezoid slices.

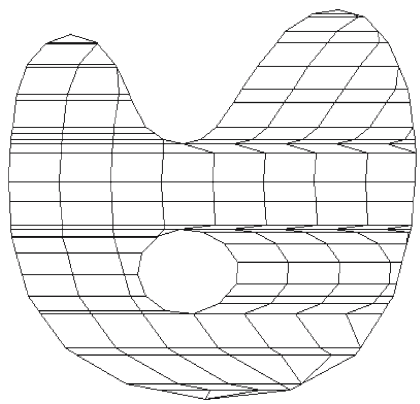


Figure 14.23. Trapezoidal decomposition of a trimmed Bezier patch.

reduced to drawing a sequence of subdivided triangles with a single bottom vertex, subdivided triangles with a single top vertex, and/or subdivided parallelogram type regions. See the document VattiTrim on the accompanying CD for more details. Sample outputs using this approach are shown in Figures 14.20(b) and 14.21(b).

Finally, consider Figure 14.23. This shows the triangles and parallelograms that are generated for a trimmed Bezier surface patch where the bounding curves are cubic B-splines. Notice that parts of the domain were subdivided more than others. This may not be desirable. The "feature" has two causes:

(1) All trapezoids are drawn no matter how thin they are.
(2) The vertical height of trapezoids is determined by scan beams and these are usually thinner than the resolution of the bounding curves would suggest.

One can mitigate these causes by using a smaller resolution for the bounding curves, expecting the trapezoids to be smaller anyway, or one could get fancier with the algorithms described above and change the subdivision of the bounding curves in

a dynamic way from one scan beam to the next. Alternatively, one could do some post-processing of the trapezoids. Actually, the problem of thin triangles or trapezoids is only part of the problem. As we pointed out in Section 14.3, how the domain of a parameterization is polygonized has, in principle, little bearing on the polygonization of the surface that this polygonization induces. If getting a uniform polygonization for a trimmed surface is important, then an approach like in [ChPP98] could be considered.

We shall end this section with a few words about a problem related to trimming curves. A trimming curve for a parametric surface p(u,v) is usually assumed to be a curve $\gamma(t)$ in parameter space of the parameterization p(u,v). One may need to deal with the trimming curve $p(\gamma(t))$ in the surface directly. The general problem is to find an appropriate representation of the composite of two functions given a representation for each separately. For example, suppose that both $\gamma(t)$ and p(u,v) have a Bézier representation, can one represent the composite $p(\gamma(t))$ as a Bézier curve? Among other applications, an affirmative answer would be useful for data exchange between CAD systems. The representation of composite functions has been studied for various types of functions. We shall leave the reader with one reference for this subject. In [LasB95] it is shown how the composite of a Bézier trimming curve and a Bézier tensor product surface can be given an explicit Bézier representation.

14.5 Implicit Shapes

14.5.1 Implicit Curves

This section considers ways to find and describe the solutions to a polynomial equation of the form

$$f(x,y) = 0. \tag{14.8}$$

We shall look at rasterization, marching, and algebraic geometry approaches.

In Sections 2.9.2 and 2.9.3 we tried to generate a solution to a special class of such equations on a pixel-by-pixel basis. A number of rasterization algorithms exist for the general case of equation (14.8). The simpler ones, such as [Chan88], may fail at singularities. An algorithm that also works at singularities is described by Taubin ([Taub94]). The basic idea of the algorithm is to traverse all the pixels (thought of as squares) and to output those that the curve intersects. Of course, it is much too inefficient to look at every pixel since the curve would only intersect a few. Therefore, Taubin uses an approach similar to the Warnock visible surface algorithm. One checks if the screen is intersected by the curve. If it is not, then one is done. If it is, then it is subdivided into four rectangular pixel areas and the same process is repeated for each of the subrectangles. One keeps subdividing until one gets down to an individual pixel. The key element in the algorithm was coming up with an efficient test whether the curve intersected a box. Actually, Taubin achieves efficiency by not having a yes-or-no test but rather an "approximate" maybe-or-no test. However, and this is an important condition in order to get a correct algorithm, if it said "maybe," then eventually it would say no to any of the subrectangles that were not intersected. We

never end up at a pixel with a "maybe" as an answer. Taubin indicated how the algorithm could be extended to three-dimensional rasterization of surfaces and surface intersections. The answers in that case were collections of boxes useful for volume rendering type situations.

A number of algorithms that solve equation (14.8) are based on marching methods. They produce polygonal approximations to the curve. For example, Timmer's algorithm for finding surface intersections can also be used to compute implicit curves. One simply defines a grid in the plane. Another marching algorithm is described by Snyder in [Snyd92]. It uses interval analysis and is better and more robust than the algorithm by Timmer or [BHLH88]. Section 18.5 has an overview of Snyder's algorithm. Of course, one could also use the algorithm described later in Section 14.6 since implicit curves can be thought of as contours.

A final approach to solving equation (14.8) that we want to describe is one that uses algebraic geometry methods. For more details see Bajaj et al. ([BHLH88]) and Hoffmann ([Hoff89]). Algebraic geometry approaches to solving equation (14.8) fall into two categories:

(1) One can try to find a parameterization for the curve.
(2) Starting with a point on the curve, one can use a standard incremental approach to crawl along the curve when one is not near any singularities, but use algebraic geometry to get past singularities.

Techniques for converting from implicit to parametric representations are discussed in Section 10.15 in [AgoM05], but not all algebraic curves are rational and Sederberg et al. ([SeZZ89]) describe a way to get approximate parameterizations. Here we describe an incremental approach. We start with a fundamental result from algebraic geometry (Theorem 10.13.26 in [AgoM05]) that says that every such curve can be transformed birationally into a plane curve $g(u,v) = 0$ which contains at most ordinary singularities, that is, points where the curve crosses itself in a transverse fashion. (We shall not count ordinary singularities as singularities in the discussion that follows.) Actually, there is a stronger theorem (Theorem 10.13.27 in [AgoM05]), which states that a curve is birationally equivalent to a curve without any singularities at all, but unfortunately the latter will in general be a curve in higher dimensions. At any rate, the basic idea for tracing a curve **C** defined by equation (14.8) will therefore be to toggle between two modes:

(1) If we are at a nonsingular point of f, then trace **C** using f and some standard Newton-Raphson root-finding method until done or until we approach a singularity of f.
(2) If we are near a singularity, then trace **C** using g in a neighborhood of this singularity until we are past the singularity and then return to tracing **C** using f again.

This approach works because a curve has only a finite number of singular points (Theorem 10.6.4 in [AgoM05]). The reason that we cannot simply use g for the whole process is that this would involve tracing through points at infinity. Algorithm 14.5.1.1 shows the steps of the algorithm. We shall now describe them in more detail.

The curve **C** is traced using what is called a *place* of f in algebraic geometry. Assume that $\mathbf{p}_0 = (x_0,y_0)$ is a point of **C** and that the formal power series

How algebraic geometry can be used to tile a planar curve defined by

$$f(x,y) = 0$$

Step 1: Let **p** be a nonsingular point of f.
Step 2: Trace f one step starting at **p** in the tangent direction to the next point **p′**.
Step 3: If we are done, then quit.
Step 4: If **p′** is a nonsingular point, then let **p** be **p′** and go to Step 2.
Step 5: Transform the curve f to a curve f_1 so that the singular point **p′** gets sent to the origin.
Step 6: Use a birational map F to transform the curve f_1 to a curve g, so that

(a) There is a neighborhood **U** of the curve f_1 about the origin and **U–0** gets mapped in a bijective way onto F(**U**)–F(**0**) by F.
(b) Each place of f_1 centered at **0** gets mapped to a regular place of g.
(c) The center of these places of g is a regular point of g.

Step 7: Trace g to get past the singularity of f and to a new nonsingular point **p**.
Step 8: Go to Step 2.

Algorithm 14.5.1.1. Incremental curve tiling algorithm.

$$\begin{aligned} x(t) &= a_0 + a_1 t + a_2 t^2 + \dots \\ y(t) &= b_0 + b_1 t + b_2 t^2 + \dots \end{aligned} \tag{14.9}$$

corresponds to a place of f with center $\mathbf{p}_0$. We can solve for these power series by substituting them into the equation (14.8) and setting the coefficients to 0.

14.5.1.1 Example. To find the parameterization (14.9) at the origin for $f(x,y) = x^3 - x^2 + y^2$.

Solution. In this case $(a_0,b_0) = (0,0)$ and we need to solve

$$\begin{aligned} b_1^2 - a_1^2 &= 0 \\ a_1^3 - 2a_1a_2 - 2b_1b_2 &= 0 \\ 3a_1^2a_2 - 2a_1a_3 - a_2^2 + 2b_1b_3 + b_2^2 &= 0 \\ \dots \qquad & \dots \end{aligned} \tag{14.10}$$

Two solutions to (14.10) are

$$\begin{aligned} x(t) &= t \\ y(t) &= t - (1/2)t^2 - (1/8)t^3 - \dots \end{aligned} \tag{14.11}$$

and

$$\begin{aligned} x(t) &= t \\ y(t) &= -t - (1/2)t^2 - (1/8)t^3 - \ldots \end{aligned} \tag{14.12}$$

The fact that there is more than one solution indicates that we are at a singular point because only one place exists at a regular point.

In general, the equations like (14.10) that one uses to solve (14.9) for a_i and b_i can actually be put into the form

$$\nabla f(\mathbf{p}_0) \bullet \gamma^{(m)}(t) = c_{f,m},$$

where $\gamma(t) = (x(t), y(t))$. One deals with them in a way similar to how equations (13.18) and (13.19) are handled in Section 13.5.2. At nonsingular points where we have a **unique** solution we trace f using an approximation to $\gamma(t)$, say by a cubic approximation by using the first three terms of the series, and then use a Newton-Raphson method to converge to a point on the curve.

So far there is nothing new. Things get interesting when we approach a singularity on **C**. Assume that the singularity is at the origin. We need to find a transformation that will map our curve into a singularity-free curve.

Removing a Singularity at the Origin. We use the quadratic transformations

$$T_1: \quad u = x \qquad \text{or} \qquad T_2: \quad u = \frac{x}{y} \tag{14.13}$$
$$v = \frac{y}{x} \qquad\qquad v = y.$$

One can show that T_1 has the following properties (with similar properties for T_2):

(1) The y-axis gets mapped to infinity.
(2) Away from the y-axis T_1 is one-to-one.
(3) The tangent lines of **C** at the origin will get mapped to distinct tangent lines at distinct regular points of the curve defined by g.

Furthermore, it is known (see Section 10.12 in [AgoM05] and in particular Lemma 10.12.15) that the singularity-free curve $g(u,v) = 0$ we are seeking can be obtained by applying a finite number of transformations of the form T_1 or T_2. If our curve has no vertical tangent lines, then we use T_1, otherwise we use T_2. The transformations resolve our singularity into a finite number of points on the curve defined by g.

We shall clarify the singularity removal process with an example. First, recall that it is easy to compute the equation of any transformed implicitly defined object. In our case, the transformations T_1 or T_2 map the curve **C** into a curve that has equation

$$g(u,v) = f\left(T_1^{-1}(u,v)\right) = 0 \quad \text{or} \quad g(u,v) = f\left(T_2^{-1}(u,v)\right) = 0,$$

respectively, and the inverses of T_1 and T_2 are defined by

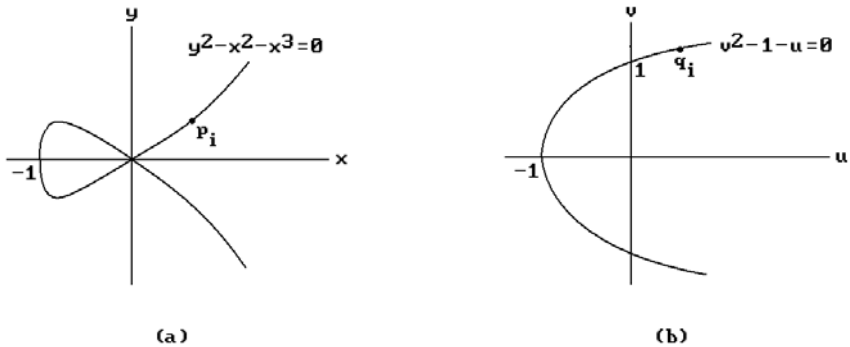


Figure 14.24. Removing the singularity of $f(x,y) = y^2 - x^2 - x^3$.

$$T_1^{-1} : x = u \quad \text{and} \quad T_2^{-1} : x = uv$$
$$y = uv \qquad\qquad y = v. \tag{14.14}$$

Also, property (3) of T_1 is simply a consequence of the fact that, except for the origin, the points of the line $y = mx$ gets mapped by T_1 to the line $v = m$.

14.5.1.2 Example. To remove the singularity of $f(x,y) = y^2 - x^2 - x^3$.

Solution. See Figure 14.24. The map T_1 transforms $f(x,y)$ into $g(u,v) = u^2 (v^2 - 1 - u)$. This means that the points of the curve **C** defined by $f(x,y) = 0$ that have nonzero x-coordinates get mapped into the curve defined by $v^2 - 1 - u = 0$. To see what happens to the branches of the curve at the origin, let us solve the equation $f(x,y) = 0$ for y. We get

$$y = \pm x\sqrt{1+x}.$$

This shows that one branch at the origin of our curve **C** satisfies equation $f_1(x,y) = 0$, where

$$f_1(x,y) = y - x\sqrt{1+x}.$$

Applying T_1 to $f_1(x,y)$ gives

$$g_1(u,v) = u(v - \sqrt{1+u}).$$

From this it is easy to see that if $\mathbf{p}_i$ are points on the branch of **C** defined by $f_1(x,y)$ that converge to the origin, then the points $\mathbf{q}_i = T_1(\mathbf{p}_i)$ converge to (0,1). In other words, the origin that was a singular point for $f(x,y)$ has been resolved into the nonsingular point (0,1) for $g(u,v)$. Finally, note that at the origin, the tangent line of the curve defined by $f_1(x,y)$ has slope 1 and the line $y = x$ gets mapped to the line $v = 1$.

Let us return to our curve tracing. Suppose that we are at point **p** on the curve **C** and we are approaching a singularity at the origin. What we have to do is the following:

(1) Determine the tangent line at the origin of the branch of f that we are currently on. Suppose that it has slope equal to m.
(2) Compute the singularity-free curve $g(u,v) = 0$ using a composite T of the quadratic transformations T_i as described above. Let $\mathbf{q} = T(\mathbf{p})$.
(3) Trace g from $\mathbf{q}$ to a point $\mathbf{q}'$ that is a small distance past (0,m) and map these traced points back to f using the inverse of T.

We will now be at the point $\mathbf{p}' = T^{-1}(\mathbf{q}')$, which is past the singularity of f and we continue tracing f as before until we get to a new singularity.

Three issues were brushed over in the description of our algorithm: determining when we are approaching a singularity, moving it to the origin, and making sure that we do not invert our tracing direction when we move to tracing g.

Finding Singularities. Our singular points are defined by the constraints $f = f_x = f_y = 0$. With infinite precision there would be no problem, but without that it turns out that we should use the condition number of the matrix

$$\begin{pmatrix} f_x & -f_y \\ f_y & f_x \end{pmatrix}$$

to determine when we are getting close to a singularity. [Hoff89] suggests several approaches. Two possible iterative approaches are using a least squares method or some sort of constrained minimization. Two possible direct approaches for finding roots are to use resultants or Gröbner bases.

Moving Singularities to the Origin. The problem here is that we need our singularity to be at the origin to apply our quadratic transformations, but we may have to apply a series of these and they are sensitive to numeric errors in the transformed functions.

Preserving the Tracing Direction. Since the vector $\nabla f = (f_x,f_y)$ is normal to the curve, the orthogonal vector $\mathbf{v} = (-f_y,f_x)$ will be tangent to it and will be the default direction in which to start traversing f. This choice is motivated by the fact that the ordered basis $(\nabla f,\mathbf{v})$ induces the standard orientation of $\mathbf{R}^2$ because

$$\det\begin{pmatrix} \nabla f \\ \mathbf{v} \end{pmatrix} > 0.$$

Intuitively this corresponds to preferring a counterclockwise direction, as for example in the case of the unit circle defined by $f(x,y) = x^2 + y^2 - 1$.

Definition. We call $(-f_y,f_x)$ the *standard trace direction for f*.

Now the actual tracing of a curve is done using a parameterization $\gamma(t) = (x(t),y(t))$ as defined by equations (14.9). The parameter t in the parameterization induces a direction on the curve that will agree with our choice if

$$d_{f,\gamma} = \det\begin{pmatrix} \nabla f \\ (a_1, b_1) \end{pmatrix} > 0.$$

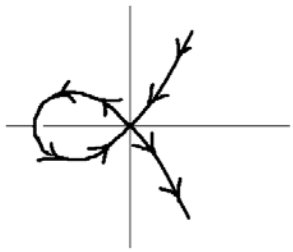


Figure 14.25. Tracing directions $(-f_y, f_x)$ for $f(x,y) = y^2 - x^2 - x^3$.

If this is not true, then we need to trace the parameterization in the negative t direction. From this we see that the issue of the tracing direction of a curve involves the following two points:

(1) The direction in which we trace is not determined intrinsically by a curve **C** but by the function f which is used to define it. If we replace f by –f, then the curve will be traced in the opposite direction.
(2) In addition to finding a parameterization for the curve, we need to check $d_{f,\gamma}$, which determines whether the parameter t induces the same direction as the one in which we want to trace.

If the curve we are tracing has no singularities, then no more needs to be said. If singularities exist, in particular if the origin is a singularity, then the standard tracing directions may not vary continuously as we pass through the origin. We can see this in Figure 14.25 ([Hoff89]). Of course in that case we do not trace the curve using the function f alone but use a transform g of it to get past the singularities. Tracing g will, just like in the case of f, involve the gradient of g and a parameterization of g. We need to establish how the tracing direction of g should be determined from that of f. Assuming that $a_0 \neq 0$, so that we can use a T_1 transformation, then $g(u,v) = f(T_1^{-1}(u,v))$, and so by the chain rule $Dg = Df \circ DT_1^{-1}$. If $(u,v) = T_1(x,y)$, then

$$\begin{aligned}\nabla g(u,v) &= (f_x(x,y) \quad f_y(x,y))\begin{pmatrix}1 & 0\\ v & u\end{pmatrix}\\ &= x\left(\frac{xf_x + yf_y}{x^2}, f_y\right).\end{aligned} \tag{14.15}$$

The parameterization for g at (u,v) induced by γ is

$$\begin{aligned}u(t) &= x(t)\\ v(t) &= \frac{y(t)}{x(t)} = c_0 + c_1 t + c_2 t^2 + \ldots\end{aligned}$$

It is easy to show by long division that

$$c_0 = \frac{b_0}{a_0}$$

$$c_1 = \frac{b_1 a_0 - a_1 b_0}{a_0^{\,2}}$$

$$\ldots$$

Because u(t) = x(t), the parameterizations of the curve and its transform should be taken either both in the positive or both in the negative t direction. From equation (14.15) we see that $\nabla f(x_0,y_0)$ and $\nabla g(T_1(x_0,y_0))$ differ in the second coordinate by the factor x_0. We therefore use the sign of x_0 to decide whether the tracing direction with respect to ∇g needs to be changed. Putting all this together, we determine the orientations for tracing as follows:

(1) Assume the current tracing direction at (x,y) with respect to f is $d(-f_y,f_x)$, where $d = \pm 1$.
(2) If we switch to tracing g, then trace g in direction $xd(-g_y,g_x)$, that is, we use g's standard trace direction if and only if $xd > 0$.
(3) When we finally are ready to switch back to tracing f, if we are tracing in direction $d(-g_y,g_x)$, then start tracing in direction $xd(-f_y,f_x)$.

14.5.1.3 Example. To apply the above steps to $f(x,y) = y^2 - x^2 - x^3$ and show how the problem indicated in Figure 14.25 disappears.

Solution. See Figure 14.26 ([Hoff89]). If we start our tracing at **A** moving toward the singularity at the origin, we eventually get to **B** where we switch to the transform g and the curve $v^2 - 1 - u = 0$. We start at the point $\mathbf{B}_1$ on that curve and then trace in direction $(-g_y,g_x)$ until we get to point $\mathbf{C}_1$ at which time we have passed the singularity and revert to tracing f at **C**, but with tracing direction $(f_y,-f_x)$.

One common problem for all methods that try to compute an implicitly defined set of points is to make sure that we end up with a set that has the correct topology. Note that we ran into a similar problem in the last chapter in the context of finding the intersection of two surfaces. Great strides have been made in the efficient application of algebraic geometry to this issue. One example of this is the paper [GonN02], where one can also find references to additional work.

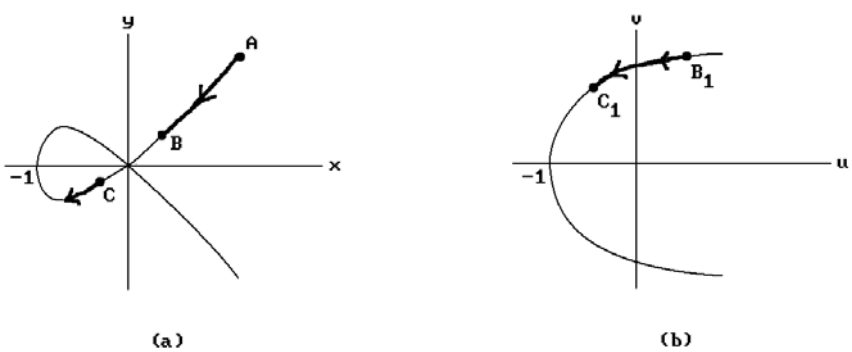


Figure 14.26. Adjusting the tracing direction at a singularity.

14.5.2 Implicit Surfaces and Quadrics

This section takes a brief look at implicitly defined surfaces. Specifically, we want to study the solution of a polynomial equation of the form

$$f(x,y,z) = 0. \tag{14.16}$$

The reader should review the general comments on implicit tilers at the end of Section 14.3.

The important special case of quadric surfaces is often dealt with separately. If all one wants to do is to render them, then [Bloo97] presents a scan line–rendering algorithm. We already know the types of quadrics that exist. See Section 3.7 in [AgoM05]. Assume that the quadric is represented in world homogeneous coordinates by the equation

$$h(x,y,z,w) = (x \;\; y \;\; z \;\; w)\, Q \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = 0. \tag{14.17}$$

Recall the discussion of the graphics coordinate system pipeline in Chapter 4. By composing the homogeneous version of the world-to-camera coordinates transformation (equation (4.4)) with the homogeneous camera-to-clip coordinates transformation (equation (4.10)), we get a transformation M, which maps from homogeneous world coordinates to homogeneous clip coordinates. It follows that the equation for the quadric in homogeneous clip coordinates is

$$(x' \;\; y' \;\; z' \;\; w')\, Q^* \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = 0, \tag{14.18}$$

where $Q^* = M^{-1}Q(M^{-1})^T$. Actually, since M^{-1} differs from the adjoint, adj(M), by a constant (the reciprocal of the determinant of M), we may simplify equation (14.18) and assume that

$$Q^* = \mathrm{adj}(M)\, Q\, \mathrm{adj}(M)^T$$

in that equation. For a pixel on the screen with integer coordinates (x,y) let $(x_s,y_s) \in [0,1] \times [0,1]$ denote its normalized device coordinates. With this notation, we can now describe the algorithm for rendering the quadric defined by equation (14.17).

For each pixel (x,y), we solve the following quadratic equation in z:

$$(x_s \;\; y_s \;\; z \;\; 1)\, Q^* \begin{pmatrix} x_s \\ y_s \\ z \\ 1 \end{pmatrix} = 0. \tag{14.19}$$

If there is no solution, then the quadric misses that pixel. Otherwise, choose the smallest z and transform the point $(x_s,y_s,z,1)$ back to camera coordinates to compute shading and texture information at that point.

For shading we need normals, but these are easy to compute for the quadric defined by equation (14.17) because it corresponds to the solution to the equation (14.16) with $f(x,y,z) = h(x,y,z,1)$ and the normal at (x,y,z) to a surface defined by an equation like (14.16) is just $\nabla f(x,y,z)$. Fix w and define $\sigma(x,y,z) = (xw,yw,zw,w)$. Then $f = h\sigma$. Therefore, applying the chain rule

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{4} D_i h \frac{\partial \sigma_i}{\partial x} = w\frac{\partial h}{\partial x},$$

where σ_i is the ith component function of σ. Similarly,

$$\frac{\partial f}{\partial y} == w\frac{\partial h}{\partial y} \quad \text{and} \quad \frac{\partial f}{\partial z} = w\frac{\partial h}{\partial z}.$$

This shows that ∇f is a multiple of

$$\left(\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, \frac{\partial h}{\partial z}\right).$$

Differentiating equation (14.17) with respect to x gives

$$\frac{\partial h}{\partial x} = 2(x \;\; y \;\; z \;\; w)\, Q \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

with similar formulas for the other partials. Finally, we can combine these facts to conclude that

$$N = 2(x \;\; y \;\; z \;\; w)\, Q \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \tag{14.20}$$

is a normal to the quadric at the point $(x/w,y/w,z/w)$.

Once one finds one visible point one does not need to check all the remaining pixels on the screen. It is possible to be more efficient and restrict the remaining pass to a smaller subset. Subsequent z and normal vector computations can also be speeded up by making incremental computations. The reader is referred to [Bloo97] for the details.

Moving on to the general case, an approach to getting a piecewise linear approximation to a solution to equation (14.16) and higher dimensional analogs in the case

of a regular map where the solution is a manifold is described in [AllS85], [AllG87], and [AllG91]. One does need to get a start point for each component of the solution. A Newton-Raphson–type method is used to do this. One then "marches" out from that start point building a triangulation as one goes along.

Snyder ([Snyd92]) describes a marching type algorithm that uses interval analysis and, because of that, is more robust than the one in [Bloo88]. Another interesting algorithm that uses interval analysis to guarantee robustness is described by Stander and Hart in [StaH97]. This algorithm is based on the handle decomposition of a manifold defined by the nondegenerate critical points of a Morse function for it. See Sections 4.5, 4.6, 8.6, and 8.7 in [AgoM05] for a discussion of the general theory behind this. One assumes that equation (14.16) defines a surface **S** that is the boundary of a solid defined by $f(x,y,z) \geq 0$ and that the function $f(x,y,z)$, or a slight perturbation of it, has only nondegenerate critical points. Assume further that the critical values of the function lie in an interval $(-c,c)$ for some c. Consider the contours $\mathbf{C}_t = f^{-1}(t)$. As t ranges from c down to 0, the sets $\mathbf{C}_t$ start with the empty set and end with **S**. In between critical values the topology of the sets $\mathbf{C}_t$ does not change. As one passes a critical value, the topology changes in a well-defined manner that is a very simple special case of the surgery described in Section 8.7 in [AgoM04]. Stander and Hart use interval analysis to find the critical points of f. They are just the zeros of the gradient of f. They then incrementally polygonize the contours $\mathbf{C}_t$ moving from one critical value to the next. The polygonization of **S** that they get in the end is then guaranteed to be topologically correct. They claim that their approach is faster than the one in [Snyd92] and produces fewer small polygons.

Finally, one can again use algebraic geometry, either to try to find a parameterization or to get past singularities that are the places where algorithms tend to run into difficulties.

14.6 Finding Contours

Problems dealing with contours come in different flavors. In this section we consider the following:

> **The general contour problem:** Given a function $f: \mathbf{R}^n \to \mathbf{R}^m$ and a point $\mathbf{c} \in \mathbf{R}^m$, find the *contour* $\mathbf{X} = f^{-1}(\mathbf{c})$. (Other terms for such a set **X** are *level curve* or *isoline* or *isosurface* with *isovalue* **c**.)

Sometimes the subset

$$\{(\mathbf{p}, f(\mathbf{p})) \mid \mathbf{p} \in \mathbf{X}\}$$

of the graph of f is called the contour rather than **X**, but this set would be easy to get once one knows **X**, and so we prefer our formulation. It should be pointed out that the contour finding problem is closely related to the problem of finding spaces that are defined implicitly because finding $f^{-1}(\mathbf{c})$ is equivalent to finding the zeros of the function $g(\mathbf{p}) = f(\mathbf{p}) - \mathbf{c}$. The two problems are just two sides of the same coin. The justification for discussing the problems separately is that they have different conno-

tations. In one case we are looking for a slice of a given object and in the other, the zero set of a function is the entire object of interest.

An entirely different context for contour problems is where we are simply given some discrete data and we are asked to find contour lines or surfaces for this data. Specifically, we would want a structured definition of the line (an ordered list of points) or surface (its edge and facet structure), not simply a random list of its points. This problem can be thought of as a kind of special case of the contour problem stated above by thinking of the data as consisting of sampled points for the zero set of a function which we do not know. We ran into the problem earlier in Chapter 10 when we discussed volume rendering and the marching cube algorithm. We shall not discuss the discrete contour problem any further here. The interested reader is referred to [Sabi85] and [Dowd85]. [HosL93] also has quite a few references. A related problem is fitting curves and surfaces to discrete data that has also been studied extensively.

We now return to the contour-finding problem as we have stated it and the case $m = 1$. If $n = 3$, then we would expect $f^{-1}(c)$ to be a surface. If $n = 2$, then we expect to get a curve. This is the case we consider in this section and "finding the contour" will mean finding a polygonal approximation for the curve. We restate the problem for this case as

> Given a real-valued function of two variables $f(x,y)$ defined on some domain $\mathbf{D}$ in $\mathbf{R}^2$ and a real number c, find a polygonal approximation to $\mathbf{X} = f^{-1}(c)$.

In general one would expect the set $\mathbf{X}$ to consist of components which are curves, but without some extra conditions on the function $f(x,y)$, it could be pretty much anything. For example, if

$$f(x,y) = x^2 + y^2,$$

then $c = 1$, 0, or -1 produces an $\mathbf{X}$ that is the unit circle, the single point (0,0), or the empty set, respectively. A more degenerate case is the constant function $f(x,y) = c$, in which case $\mathbf{X}$ is all of $\mathbf{D}$. One well-known condition that guarantees that $\mathbf{X}$ **does** consist of nice curves is that f is a continuously differentiable function whose derivative has rank 1 on $\mathbf{X}$. This condition is not one that one can always assume in practice however, and so any algorithm that finds contours has to be able to handle some degenerate cases.

Contour finding algorithms typically consist of two steps:

Step 1: One has to find starting points, that is, one has to find one point on each component of the contour.

Step 2: Given a starting point in a component, one then has to trace out the rest of that component.

Step 1 is the problem of finding solutions to or zeros of equations about which much is known. See Section 4.7 in [AgoM05]. We say no more about it here and concentrate instead on Step 2.

Assuming that one has found one point $\mathbf{p}_0$ of a component $\mathbf{C}$ of $\mathbf{X}$, two basic approaches to finding the rest of it are:

Approach 1: We can use the tangent to **C** at $\mathbf{p}_0$ to guess another nearby point.
Approach 2: We can use a triangulation of the domain, replace the function f by an approximation g obtained via linear interpolation of the values of f at the vertices of the triangulation, and then find the intersection of **C** with the edges of the triangulation.

Approach 1 is basically what we used in Section 2.5.2 and 2.9.2 to draw lines and circles. If **v** is a tangent vector to **C** at $\mathbf{p}_0$, we start with $\mathbf{q}_0 = \mathbf{p}_0 + \varepsilon\mathbf{v}$ as a guess for the next point on **C** and then adjust this guess to compensate for any deviation of the contour from the tangent line. For example, we can look along the normal to the tangent line at $\mathbf{q}_0$ to move back towards **C**. See Figure 14.27.

Of interest to us in this section is Approach 2 and the algorithm that is described by Dobkin et al. in [DLTW90]. That paper also compares the two approaches. Roughly speaking, Approach 1 produces an approximate solution using exact values of the function and Approach 2 produces an exact solution using an approximation to the function.

Because the complete algorithm described in [DLTW90], including handling of various degenerate cases, is very lengthy, we only present an overview of it here and refer the reader to the paper for the missing details. There is no loss of generality by assuming that c = 0, and so our problem will be the following:

> Given one point $\mathbf{p}_0$ with $f(\mathbf{p}_0) = 0$, find the rest of the component **C** of the contour $f^{-1}(0)$ to which $\mathbf{p}_0$ belongs.

The general idea of the algorithm is to triangulate the domain of f and to find the points where the component intersects the edges of the triangles. The component is then the polygonal curve obtained by connecting these intersection points with segments. See Figure 14.28. In essence we are solving the contour problem for an approximation g to f, where g agrees with f on the vertices of the triangulation but is a linear interpolation of the vertex values over the rest of each triangle. The advantages of this approach are

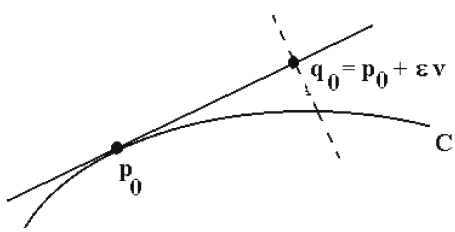


Figure 14.27. Tangent line approximation to a contour.

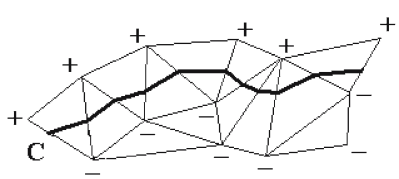


Figure 14.28. Connecting edge intersections.

(1) One only needs to evaluate f at the vertices at most.
(2) Given the way that the tracing proceeds this means that we need at most as many function evaluations as there are triangles.

The main part of the algorithm is iterative in nature. At each stage we assume that we have just "entered" a new triangle from some point $\mathbf{p}_i$ on one of its edges ($f(\mathbf{p}_i) = 0$). If the initial point does not lie on an edge of a triangle, then we use interpolation to replace it with one that does. Therefore, assume

(1) the triangle we are entering has vertices $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$,
(2) $\mathbf{p}_i$ lies in the interior of edge $[\mathbf{v}_0,\mathbf{v}_1]$, and
(3) $f(\mathbf{v}_1) < 0 < f(\mathbf{v}_0)$.

The contour will leave the triangle through an edge $\mathbf{e}$ that is determined as follows: if $f(\mathbf{v}_2) > 0$, then the contour leaves through $\mathbf{e} = [\mathbf{v}_1,\mathbf{v}_2]$, otherwise, $f(\mathbf{v}_2) < 0$, and the contour leaves through $\mathbf{e} = [\mathbf{v}_0,\mathbf{v}_2]$. In either case, a simple linear interpolation of the function values at the vertices finds us the new point $\mathbf{p}_{i+1}$ in the interior of the edge through which the contour leaves the triangle ($f(\mathbf{p}_{i+1}) = 0$). We now repeat these same steps with the point $\mathbf{p}_{i+1}$ and the triangle adjacent to the current one along edge $\mathbf{e}$.

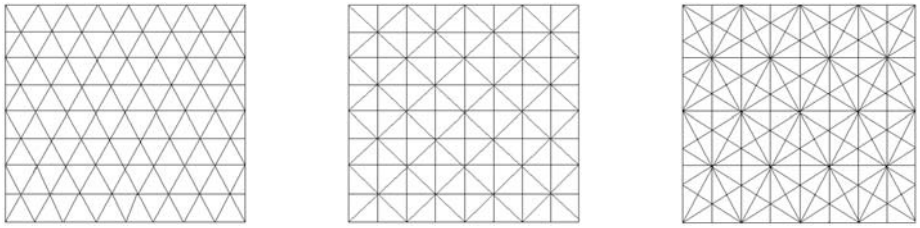
The iteration finishes when we return to the starting triangle or if we leave the domain $\mathbf{D}$. In the first case we have found our component and it is a closed loop. In the second case we again start with $\mathbf{p}_0$ but move in the opposite direction to find the other "half" of the component that now is a curve with two ends.

There is one final aspect of this algorithm. It has to do with the efficient way that the triangulation is dealt with in [DLTW90]. In particular, there is no need to somehow precompute a global triangulation of the domain or the values of the function on its vertices. All one needs to do is to be able to generate the triangulation "locally." The properties one wants are to be able to find vertices of adjacent triangles easily, all triangles should have roughly the same size, and their dimensions should roughly be the same in different directions (no "thin" triangles). There are triangulations, called *monohedral triangulations generated by reflections*, which satisfy these properties. The name means that the simplices in the triangulation are all congruent and all the simplices are generated from a fixed one by reflection about faces. There is a complete classification of such triangulations. The paper [DLTW90] discusses them and also gives further references. The three possible triangulations of this type in the plane are shown in Figure 14.29(a–c). They are the so-called *Coxeter triangulations* of type P_3, R_3, and V_3, respectively. The P_3 and R_3 triangulations are actually just a two-dimensional instance of a family of triangulations P_m and R_m, which exist in dimension $m - 1$. We shall only summarize the main conclusion relevant in the context of the special contour finding algorithm being described here.

The reflection rule associated to finding the new vertex of a triangle adjacent to an edge is simplest when using a triangulation of the P_3 type shown in Figure 14.29(a):

The P_3 reflection rule: To reflect vertex $\mathbf{v}_i$ of a triangle about the opposite edge, replace it by $\mathbf{v}_{i-1} + \mathbf{v}_{i+1} - \mathbf{v}_i$, where indices are taken modulo 3.

This rule and the other reflection rules for the R_3 and V_3 triangulations can actually be applied to **any** triangle and therefore produces an algorithm for generating a tri-



(a) P_3 rule **(b)** R_3 rule **(c)** V_3 rule

Figure 14.29. Three triangulations by reflections.

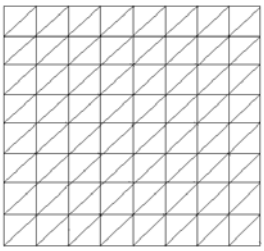


Figure 14.30. The P_3 reflection rule applied to a basic R_3 triangle.

angulation along with a labeling of the vertices of this triangulation. For example, applying the P_3 reflection rule to a basic R_3 triangle in Figure 14.29(b) produces the triangulation shown in Figure 14.30. Here is what is recommended by [DLTW90]: The P_3 triangulation seems to be the best geometrically for the contour-finding algorithm. Start off with the triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ centered at the origin, where

$$\mathbf{v}_0 = (0,0), \quad \mathbf{v}_1 = 2\left(\frac{1}{\sqrt{6}},0\right), \quad \text{and} \quad \mathbf{v}_2 = \left(\frac{1}{\sqrt{6}},\frac{1}{\sqrt{2}}\right).$$

Scale this triangle to the appropriate size and then translate it so that our start point becomes its barycenter. Now start the iterative algorithm described above. If one wants to find several components of the contour and we want all of them to use the same triangulation, then things get a little trickier. See [DLTW90].

With our choice of triangulation one can simplify the termination test, which checks for loops. This can be done by expressing points with respect to the basis $\mathbf{v}_1$ and $\mathbf{v}_2$. For example, the coordinates of the barycenters of all simplices will then have the form (1/3)k, where k is an integer unique to the simplex. Checking whether a contour has terminated in a loop therefore amounts to simply checking if two integer vectors are equal. The authors of [DLTW90] suggest that it is possible to avoid some round-off errors if other computations are also done in that basis. Of course, the conversion from Euclidean coordinates to the coordinates with respect to that basis costs a little time.

Finally, we need to say something about the limitations of the algorithm just described. First of all, we only dealt with the nondegenerate case. There are two degenerate cases that must be handled in special ways. The first is where the derivative of

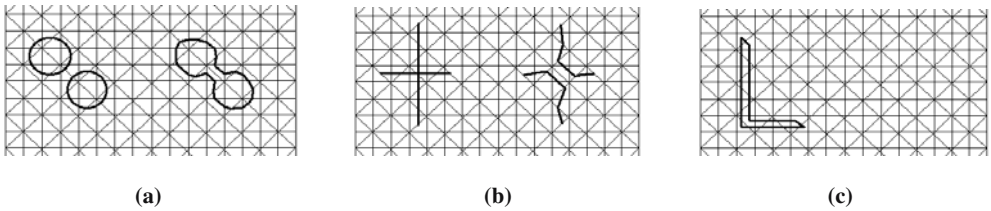


Figure 14.31. Problems for nondegenerate cases.

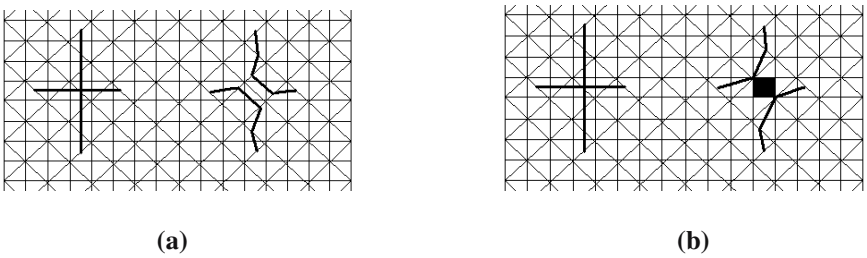


Figure 14.32. Biasing a function toward zero.

f vanishes, which basically corresponds to the case where entire simplices are mapped to 0 and the contour is not one-dimensional. The second is where the contour passes through a vertex of the triangulation. We refer the reader to the paper for ways to handle these cases. However, there are some potential problems even in the nondegenerate case if one does not choose the triangulation well. Some of the ways that the algorithm can produce results with incorrect topologies are shown in Figure 14.31. Figure 14.31(a) shows how disconnected contour components can lead to a connected answer. Figure 14.31(b) shows the opposite. Figure 14.31(c) shows how a component can be lost entirely if its interior does not contain any vertices. It is therefore important to choose the "resolution" of the triangulation carefully. It must be noted, however, that although one would hope to have an algorithm of this type produce the correct results for f's that appear in **real** applications, one should **not** expect this for arbitrary f because one can find differentiable f that have **arbitrary** compact subsets of $\mathbf{R}^2$ as their contour. Furthermore, consider Figure 14.32 which shows the contour of the function f(x,y) = xy. That example shows that increasing the resolution is not good enough by itself, because, as shown in Figure 14.32(a), the algorithm would represent the actual connected contour by two disjoint pieces. That means that, given a starting point on the contour, the algorithm would only trace out one of the pieces and leave out the other one. One way to deal with this is to add a bias to 0, meaning that we replace the function values at vertices that are less than some ε by 0. Figure 14.32(b) shows how this can correct the problem in Figure 14.32(a), but with the potential effect of blacking out some squares. Furthermore, it is not easy to choose the correct ε. Making it too big would black out too many squares. [DLTW90] suggests a trial and error method as the best approach here.

This finishes our discussion of contours, although much is also known for the higher dimensional cases. As one reference for the contour surface problem we mention [Rock90].

14.7 Skinning

Sections 13.4.3 and 14.6 dealt with what might be called finding level curves of a given surface. This section turns the problem around.

Definition. A *skinning surface* for a sequence of sets $\mathbf{C}_0, \mathbf{C}_1, \ldots, \mathbf{C}_k$ is a surface **S** satisfying

(1) **S** interpolates the sets, and
(2) there is a continuous function f: $\mathbf{S} \to [0,1]$ and real numbers $0 = t_0 < t_1 < \ldots < t_k = 1$ so that $\mathbf{C}_i = f^{-1}(t_i)$.

The process of finding such a surface is called *skinning*. In practice, the sets $\mathbf{C}_i$ consist of one or more curves, in which case they are sometimes called *skinning curves*.

In many cases, skinning is just a more recent term for lofting. Condition (2) of a skinning surface is simply trying to capture the intent that the curves $\mathbf{C}_i$ should be contours or level curves of the surface with respect to a function. Some skinning algorithms may at times produce self-intersecting surfaces, but that would be an undesirable "feature" of the algorithm. A common case for the curves is where they are planar and correspond to parallel cross-sections of a surface. It is easy to imagine though how complicated things might get if one is given an arbitrary collection of curves.

Of course, like in all interpolation problems, one wants more from the skinning surface **S** than that it interpolates the curves. Specifically, some properties that one wants **S** to satisfy are:

(1) the shape of the surface should match the shape suggested by the curves and it should not have any unnecessary wiggles.
(2) The surface should be smooth if all the curves are.
(3) The algorithm for getting **S** should be *affinely invariant*, meaning that if the curves are moved by an affine transformation T, then the algorithm applied to the new curves $T(\mathbf{C}_i)$ should produce $T(\mathbf{S})$.

Skinning algorithms have been defined in two quite different contexts, a polygonal and a smooth one. The polygonal case, which is often the harder one, is where the curves are polygonal and we are looking for a faceted **S**. The smooth case is where the curves are smooth parametric curves and one wants a smooth parameterization for **S**. This section will only touch on a few aspects of the skinning problem. Two papers that have numerous references to work on this subject are [MeSS92] and [ParK96].

We consider the polygonal case of the skinning problem first. Meyers et al. ([MeSS92]) break this problem into four subproblems:

The correspondence problem: One has to determine a correspondence between closed curves of adjacent contours. This defines some crude topological properties of an object. For example, if two closed curves of one contour correspond to one closed curve on the next, this would indicate a saddle structure for the object. Clearly, the only hope for coming up with a reasonable answer to the correspondence problem is to assume that the contours of the skinning surface have been spaced close enough together so that simple heuristics will work.

The tiling problem: One has to determine a triangulation for the surface that is to span two adjacent contours. There is no unique triangulation, and so solutions to this problem try to optimize some function that measure the appropriateness of spanning surfaces. Many such "metrics" have been proposed. Each one has its problems.

The branching problem: If there is more than one closed curve on adjacent contours, then one may need to determine how two or more closed curves in one contour get merged into one closed curve in the next contour. This would typically also entail determining how parts on the closed curves of the first contour are related. The tiling problem is more complicated when there are branches. Early work on tiling tended to assume that there were no branches or expected a user to help out interactively.

The surface-fitting problem: After one has a tiling, one may want to fit a smooth surface to that tiling.

Let us look at a simple aspect of the tiling problem. Assume that two adjacent contours are defined by two closed polygonal curves with distinct vertices $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_m$ and $\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_n$, respectively. One would like a triangulation of the skinning surface $\mathbf{S}$ to look something like the surface shown in Figure 14.33, where m = n = 4. Now, in general, m and n are not equal, but the real problem is that one is not given any information about which point $\mathbf{q}_j$ should connect to which point $\mathbf{p}_i$. For example, in Figure 14.33, how do we determine that $\mathbf{q}_2$ is the point that should be connected to $\mathbf{p}_0$ and $\mathbf{p}_1$? We certainly do not want to connect $\mathbf{p}_0$ and $\mathbf{q}_0$. It is not the case that there is no algorithm for finding the triangulated surface as shown in Figure 14.33. One could do an exhaustive search, but this would be prohibitively slow to carry out. The issue is finding an efficient algorithm. Fuchs et al. ([FuKU77]) reduced the problem to one of searching a toroidal graph. (A *toroidal graph* is a graph that can be embedded in a torus.) They used a "divide-and-conquer" technique along with an area-minimizing heuristic. A number of other approaches have been proposed using different heuristics. For such heuristics to work reasonably well, a common assumption

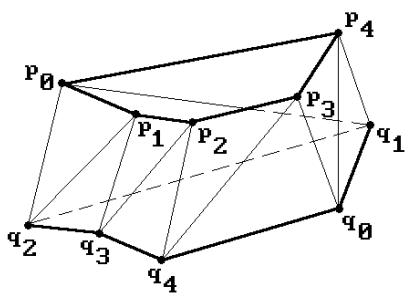


Figure 14.33. Skinning polygonal curves.

is that the two curves satisfy some not unrealistic coherence condition. Basically, one assumes that the curves are closed, planar, coherent in size and shape, and mutually centered. By transforming the planar curves in a preprocessing stage one can always arrange it so that this is the case. [GanD82] describes an algorithm that runs in time O(m + n) using a heuristic that seems to work well in practice. [GanD82] also has references to other work. We refer the reader to that paper for details.

Meyers et al. ([MeSS92]) describe a skinning algorithm that handles the correspondence problem without requiring overlapping contours and generates a tiling even in the presence of relatively complex branching of contours and without adding new vertices. Park and Kim ([ParK96]) address the surface-fitting problem. They represent contour curves as cubic closed B-spline curves with a common knot vector and find an approximation to the skinning surface that is a C^2 bicubic closed B-spline surface.

Next, we consider the skinning problem for a sequence of curves that are B-splines and describe the standard algorithm for creating a skinning B-spline surface. Algorithm 14.7.1 outlines the steps. Let

$$p_j(u) = \sum_{i=0}^{n} N_{i,d}(u)\mathbf{p}_{i,j}, \quad 0 \le j \le k,$$

be the parameterization of the curve $\mathbf{C}_j$ after steps 1–3 in the algorithm. Piegl and Tiller ([PieT95]) suggest defining the parameters w_j and knots v_j in Step 4 as follows in order to match the chordal lengths:

Inputs: A sequence of B-spline curves $p_0(u), p_1(u), \ldots, p_k(u)$
Output: A skinning B-spline surface p(u,v) for these curves

Step 1: Change the domain of the curves, if necessary, so that they all have the same domain.
Step 2: Arrange it so that all curves have the same degree. Use the degree elevation algorithm to raise the degree of any particular curve to the maximum degree d of all the curves.
Step 3: Merge the knots of all the curves into a knot vector $\{u_i\}$. For each curve use the knot refinement algorithm to get a new B-spline that has $\{u_i\}$ as its knot vector. All the curves will now have the same knot vector. Let $\mathbf{p}_{0,i}, \mathbf{p}_{1,i}, \ldots, \mathbf{p}_{n,i}$ be the control points of the ith curve.
Step 4: Choose a degree e ($\le$ k) for the v-direction, parameters $\{w_j\}$, and a knot vector $\{v_j\}$.
Step 5: For each j, $0 \le j \le n$, let $q_j(v)$ be the B-spline of degree e that interpolates the points $\mathbf{p}_{j,0}, \mathbf{p}_{j,1}, \ldots, \mathbf{p}_{j,k}$ and that satisfies $q_j(w_i) = \mathbf{p}_{i,j}$. Let $\mathbf{q}_{0,j}, \mathbf{q}_{1,j}, \ldots, \mathbf{q}_{k,j}$ be the control points of the B-spline $q_j(v)$.
Step 6: The skinning surface is then defined to be the tensor product B-spline surface p(u,v) that has control points $\mathbf{q}_{i,j}$ and knot vectors $\{u_i\}$ and $\{v_j\}$.

Algorithm 14.7.1. A B-spline skinning algorithm.

$$w_0 = 0, \quad w_k = 1,$$
$$w_j = w_{j-1} + \frac{1}{n+1}\sum_{i=0}^{n}\frac{|\mathbf{p}_{i,j} - \mathbf{p}_{i,j-1}|}{L_i}, \quad 0 < j < l$$

where L_i is the chord length of the polygonal curve with vertices $\mathbf{p}_{i,0}, \mathbf{p}_{i,1}, \ldots, \mathbf{p}_{i,k}$, and

$$v_0 = \ldots = v_e = 0, \quad v_{k+1} = \ldots = v_{k+e+1} = 1,$$
$$v_{s+e} = \frac{1}{e}\sum_{i=s}^{s+e-1} w_i, \quad s = 1, \ldots, k-e.$$

There are problems with this approach.

(1) The surface may not have the desired shape.
(2) There may be lots of knots $\{u_i\}$ with some very close together causing potential numeric problems in applications that may use this data.
(3) The surface may have self-intersections.

Part of the problems is caused by the fact that we had to choose the **same** knot vector for all the curves in the v-direction. In order to overcome some of these problems [FilB89] describes a skinning surface in a procedural way. This new method also does not require the curves to be B-splines. The steps to compute the parameterization p(u,v) at (u,v) are outlined in Algorithm 14.7.2. In practice, a cubic approximation to the function $v_i(u)$ simplified the computations and was found to be adequate.

Finally, we point out that skinning is really a special case of the more general surface reconstruction problem. We are given a set of scattered data points and want to fit a surface to those points. There is no time to go into this topic here, but we leave the reader with sample references for this and a related problem. In [BoiC00] the reconstructed surface is an implicitly defined smooth manifold. The paper [ACDL00] simplifies a previous algorithm and also proves that if the data came from a surface, then the construction is homeomorphic to the input surface. In [HuaM02] there is an algorithm for creating a combinatorial manifold mesh given some unorganized point samples from an existing object. A related two-dimensional problem is image reconstruction that involves fitting a continuous intensity surface through image samples and [YuMS01] describes an approach that does not depend on patch boundaries lining up with coordinate axes. These papers contain many additional references.

14.8 Computing Arc Length

Let

$$p : [a,b] \to \mathbf{R}^3 \tag{14.21}$$

be a parametric curve. This section discusses algorithms that solve the following two problems:

Inputs: Contour curves $p_i(u)$, real numbers u and v
Output: p(u,v)

Step 1: Because one no longer requires the same v knot vector for all u, one computes a knot vector $v_0(u), v_1(u), \ldots, v_k(u)$ as follows:

$$v_0(u) = 0 \ , \ v_1(u) = 1 \ , \ \text{for all } u$$
$$v_i(u) = \frac{L_i}{L_k} \in [0,1] \ , \ 1 < i < k \ ,$$

where L_j is the length of the polygonal curve with vertices $p_0(u), p_1(u), \ldots, p_j(u)$.
Step 2: Determine s so that $v_s(u) \le v < v_{s+1}(u)$.
Step 3: Evaluate $p_{s-1}(u)$, $p_s(u)$, $p_{s+1}(u)$, and $p_{s+2}(u)$.
Step 4: Determine the cubic curve D(w), $0 \le w \le 1$, which starts at $p_s(u)$, ends at $p_{s+1}(u)$, and has tangent vectors at its endpoints that match the tangent vector to the parabolas defined by $p_{s-1}(u)$, $p_s(u)$, $p_{s+1}(u)$ and $p_s(u)$, $p_{s+1}(u)$, $p_{s+2}(u)$, respectively.
Step 5: Let $p(u,v) = D\left(\frac{v - v_s}{v_{s+1} - v_s}\right)$

Algorithm 14.7.2. A procedural skinning algorithm.

The Arc length problem: Compute the length L of p.

The Arc-length parameterization problem: Given $s \in [0,L]$, find the point **q** on the curve so that the part of the curve from p(a) to **q** has length s. In practice, the problem is one of finding $u \in [a,b]$, so that p(u) is that point.

By definition, the length L of the curve p in (14.21) is the limit of sums of the form

$$L(\{u_i\}) = \sum_{i=1}^{n} |p(u_{i-1})p(u_i)|, \tag{14.22}$$

where the limit is taken over all partitions $\{u_i\}$ of [a,b], $a = u_0 < u_1 < \ldots < u_n = b$, whose norms go to zero. It follows that $L(\{u_i\})$ will be a good approximation to L provided that the polygonal curve $p(u_0), p(u_1), \ldots, p(u_n)$ is a good approximation to the curve p. Therefore, a quick and dirty way to estimating arc length is just to make a guess as to a reasonable subdivision $\{u_i\}$ and then to use the value $L(\{u_i\})$. We can also get a quick estimate for **q** in the arc-length parameterization problem as follows:

(1) Make a table of the lengths s_i of the polygonal curve defined by the points $p(u_0), p(u_1), \ldots, p(u_i)$.
(2) Given s, find j so that $s_j \le s \le s_{j+1}$.
(3) Set **q** = p(u), where u is the corresponding interpolated value in $[u_j,u_{j+1}]$, that is,

$$u = u_j + \frac{s - s_j}{s_{j+1} - s_j}(u_{j+1} - u_j).$$

On the other hand, to get a more accurate values we must do more work. We start with the arc length problem. Our first good approximation to arc length based on equation (14.22) relies on getting a **good** polygonal approximation to the curve and then using the length of that polygonal curve. Any good polygonal approximation will work. One such is the de Figueiredo algorithm (Algorithm 14.3.1) described in Section 14.3, where one subdivides recursively until the curve is relatively flat according to some criterium. Algorithm 14.8.1 computes arc length based on the polygonal approximation that one gets from this.

Next, note that for smooth functions p(u) the sums $L(\{u_i\})$ in equation (14.22) are Riemann sums which converge to an integral. Specifically,

$$L = L(a,b) = \int_a^b |p'|. \tag{14.23}$$

Therefore, another way to compute L is to use a known numerical approximation to integrals. Now sometimes one is in a situation where one has a fixed curve p(u) and one needs to answer multiple requests for arc lengths of different subarcs p([c,d]) of p(u). Guenter and Parent ([GueP90]) deal with this situation. The authors decided to use a Gaussian quadrature method for evaluating integrals because it is a particularly efficient integration method. They suggested that it is advantageous here to build a table of a certain number of precomputed arc length values in order to speed up the computation of other values.

For any $c,d \in [a,b]$, let GQ(c,d) denote the Gaussian quadrature approximation to the integral

Assume that Flat($\mathbf{p}_1$,$\mathbf{p}_2$,$\mathbf{p}_3$) is a function that returns **true** or **false** depending on whether the three consecutive points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ pass some flatness test.

```
real function Length (curve p; real c, d; point pc, pd)
begin
    real s;
    point ps;

    s := random number in (c,d);
    ps := p(s);
    if Flat (pc,ps,pd)
        then    return |pcpd|
        else    return ( Length (p,c,s,pc,ps) + Length (p,s,d,ps,pd) );
end;
```

Algorithm 14.8.1. Arc length based on polygonal approximation.

$$\int_c^d |p'|. \tag{14.24}$$

Note that since Gaussian quadrature is defined for functions defined over [–1,1], in order to use this method one must reparameterize the integral in (14.24) to

$$\int_{-1}^{1} \left(\frac{d-c}{2}\right) \left| p'\left(\frac{(d-c)t+d+c}{2}\right)\right| dt.$$

In [GueP90] one then builds a table (u_j,s_j), $0 \le j \le n$, using Algorithm 14.8.2. The $\{u_j\}$ form a partition of [a,b] and s_j is a Gaussian quadrature approximation to

$$\int_c^{u_j} |p'|.$$

Using this table, if one wants L(a,u) one first finds the j so that $u_j \le u < u_{j+1}$ and returns the value

```
ε        = some fixed error tolerance
(u0,s0)  = (a,0)
n        = 0;
(un,sn)  = (b,Subdivide (a,b,GQ(a,b),0,ε)

real function Subdivide (real c, d, fullInt, totLength, epsilon)
begin
    real m, lval, rval, lsub;

    m    := (c + d)/2 ;
    lval := GQ (c,m);
    rval := GQ (m,d);
    if  (|fullInt– lval– rval| > epsilon)
        then
            begin
                lsub := Subdivide (c,m,lval,totLength,epsilon/2);
                totLength := totLength + lsub;
                (un,sn) := (m,totLength);
                n := n + 1;
                return  ( lsub + Subdivide (m,d,rval,totLength,epsilon/2);
            end
        else  return  (lval + rval);
end;
```

Algorithm 14.8.2. Building a table for arc length computation.

$$s_j + GQ(u_j, u).$$

Because u lies in $[u_j, u_{j+1}]$, the computation for $GQ(u_j, u)$ will be fast. In general, for any $[c,d] \subseteq [a,b]$, $L(c,d)$ is computed as $L(a,d)-L(a,c)$. Algorithm 14.8.2 is an adaptive algorithm. Basically, one keeps subdividing intervals **I** in half and as long as GQ(**I**) differs by more than some small tolerance ε one uses

$$GQ(\text{left half of } \mathbf{I}) + GQ(\text{right half of } \mathbf{I}).$$

If one subdivided **I**, then the midpoint of **I** and the associated arc length to that point from a gets entered into the table.

Another algorithm for computing arc length in the special case of Bézier curves is described by Gravesen in [Grav95]. It is an adaptive algorithm based on the fact that the control polygon of such curves converges to the curve under subdivision.

Next, we move on to the arc-length parameterization problem. One of the earliest algorithms for finding arc-length parameterizations is described by Sharpe and Thorne in [ShaT82]. Given the curve defined by (14.21) and $s \in [0,L]$ we need to solve

$$s = \int_a^t |p'| \tag{14.25}$$

for t. This is equivalent to solving

$$f(t) = \int_a^t |p'| - s = 0. \tag{14.26}$$

Starting with an initial guess t_0, the standard Newton-Raphson method applied to equation (14.26) generates a sequence of numbers t_n,

$$\begin{aligned} t_n &= t_{n-1} - \frac{f(t_{n-1})}{f'(t_{n-1})} \\ &= t_{n-1} - \frac{f(t_{n-1})}{|p'(t_{n-1})|}, \quad n > 0, \end{aligned} \tag{14.27}$$

that hopefully converges to a root of f(t). The integral in the formula (14.26) is evaluated in [ShaT82] via Romberg integration that worked better than the trapezoidal or Simpson method. Furthermore, the fact that arc length is often roughly proportional to the parametric value motivated the authors to use the following starting value

$$t_0 = a + \frac{\Delta t}{\Delta s} s,$$

where

$$\Delta t = 0.1 \quad \text{and} \quad \Delta s = \int_a^{a+\Delta t} |p'|.$$

This method for computing the arc-length parameterization seemed to work well for [ShaT82] and had no problems with convergence.

Sometimes one needs both arc lengths and arc-length parameterizations. In that case one can use the arc-length algorithm from [GueP90]. The table (u_j,s_j) described earlier not only helps with arc length but also with arc-length parameterization. If we want to find u so that $L(a,u) = s$, then we find the j so that $s_j \le s < s_{j+1}$. This will tell us that $u \in [u_j,u_{j+1}]$ and give us a much better starting point for the Newton-Raphson method applied to (14.26) so that it will run faster.

A problem related to finding the arc length of a curve is to find a curve between two points that has specified tangent lines at the endpoints and also has a specified length. See [RouB96a] and [RouB96b].

14.9 Offset Shapes

14.9.1 Offset Curves

To *offset* a curve or surface traditionally means to move the original a certain distance along a normal vector. Here we are thinking of the curve or surface as lying in $\mathbf{R}^3$ and the normal vector being a vector in $\mathbf{R}^3$. One can define a more intrinsic notion, however, and the engulfing space does not have to be $\mathbf{R}^3$. For example, given a curve in some surface **S**, one can define the offset of that curve **in S**. This would be gotten by moving that curve along geodesics in **S** that are orthogonal to the original curve. Such offsets are called *geodesic offsets*. A good overview of the literature on offset curves and surfaces can be found in [Pham92] and [Maek99]. See also [Brec92].

In general, getting an offset may involve stretching or shrinking the original. Figure 14.34 shows an ellipse **C** and four offset curves $\mathbf{C}_i$. Offsets are interesting also in cases where the original object is not smooth but has differentiability discontinuities. Figures 14.35(a) and 14.36(a) show offset curves when there are cusps and corners. The offsets that are probably desired are shown in Figures 14.35(b) and 14.36(b). To get them one has to do some trimming and blending.

Offset curves are in general computationally more complicated than the original curve. This is basically caused by the fact that one needs to take a square root to get a unit normal vector. For example, except for special cases, the offset of a polynomial or rational curve is not a polynomial or rational curve.

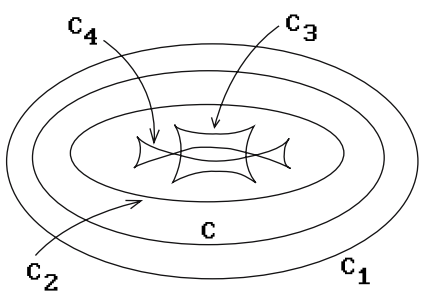


Figure 14.34. Offset curves for an ellipse.

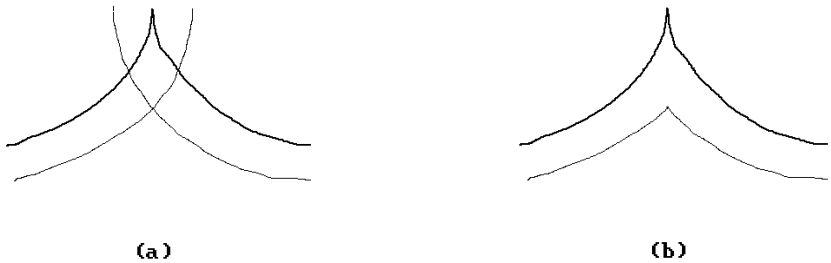


Figure 14.35. Offset curves when there are cusps.

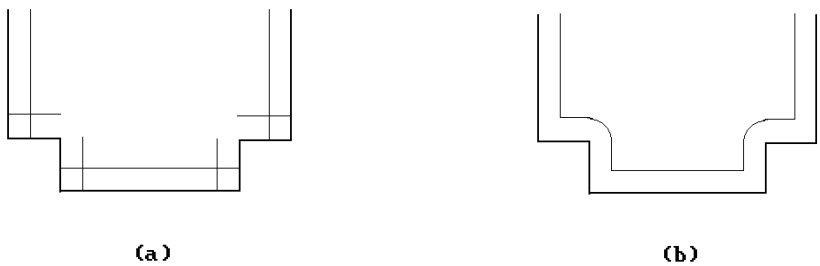


Figure 14.36. Offset curves when there are corners.

This section will look at some of the mathematics behind offset curves and how it can be used to solve practical problems. The reader is assumed to have familiarity with some basic differential geometry. The necessary background can be found in Chapter 9 in [AgoM05]. In particular, see Section 9.7 in that chapter on parallel curves (that is what offset curves are called in differential geometry) for proofs of many of the mathematical assertions made here.

Offset curves are useful tools in a number of practical applications, such as in milling operations, tolerance analysis, and robot path planning. We shall start our discussion of analytic properties of offset curves by considering those of planar curves. This is what is relevant for describing the path of a two-axis NC machine that can move in two orthogonal directions (say the x- and y-direction). Such machines are commonly used to cut two-dimensional outlines in materials. See Figure 14.37. Two serious problems for them are *local overcuts* (*gouging*) or *local undercuts*. Figure 14.38(a) shows a gouging example. As described in [MaeP93], the problem is that in regions where the curvature and the signed offset distance have opposite signs, the cutter radius must be smaller than any radius of curvature of the boundary curve of the part being machined. If we simply eliminate the part of the cutter path that started and ended at the intersection point of the offset curve, then we eliminate the overcut, but will now have produced an undercut. See Figure 14.38(b).

Let $p(u) = (x(u),y(u))$ be a regular parameterization of a curve **C** in the plane. The vector

$$n(u) = \frac{1}{|p'(u)|}(-y'(u), x'(u))$$

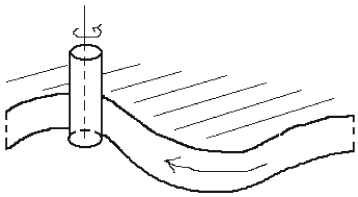


Figure 14.37. Offset curve generated by two-axis NC machine.

(a) Local overcut (gouge)

(b) Local undercut

Figure 14.38. NC cutter over- and undercuts.

is a counter-clockwise oriented **unit** normal vector to this curve.

Definition. The *offset curve* $p_d(u)$ to the planar curve $p(u)$, which is a distance $|d|$ from $p(u)$ for some real number d is defined by

$$p_d(u) = p(u) + dn(u) \tag{14.28}$$

Since

$$p_d'(u) = p'(u)(1 + \kappa(u)d), \tag{14.29}$$

where $\kappa(u)$ is the curvature function of $p(u)$, we see that the tangent vector to $p_d(u)$ is parallel to the tangent vector to $p(u)$. One can also show that the curvature function for $p_d(u)$, $\kappa_d(u)$, is defined by

$$\kappa_d(u) = \frac{\kappa(u)}{1 + \kappa(u)d}.$$

Its derivative with respect to arc-length parameterization is given by

$$\frac{d\kappa_d}{ds_d} = \frac{1}{(1 + \kappa d)^3}\frac{d\kappa}{ds}. \tag{14.30}$$

Note that even if the original curve $p(u)$ has a nice smooth shape, $p_d(u)$ may not be. In Figure 14.34, the offset curves $\mathbf{C}_1$ and $\mathbf{C}_2$ are as nice as the original ellipse. On

the other hand, $\mathbf{C}_3$ and $\mathbf{C}_4$ have cusps and $\mathbf{C}_4$ has self-intersections on top of that. The offset curve may also intersect the original curve. Equation (14.29) points out part of the problem. Even though p(u) may be a regular curve, $p_d(u)$ will not be at those points where

$$1 + \kappa(u)d = 0. \tag{14.31}$$

There may be cusps at those points. This is only a symptom of a bigger problem, which is that a curve and its offset curve usually have different analytic types. For example, the offsets of rational curves are typically not themselves rational curves.

One often wants to talk about offset curves even when the curve is not differentiable everywhere and the normal n(u) may not be defined. See Figure 14.35(a) or 14.36(a), where we are dealing with a piecewise differentiable curve. One can deal with such problems in the case of simple isolated singularities like cusps or corners.

After this litany of problems with offset curves, let us look at approaches to dealing with them. The idea is to subdivide the domain of the curve into segments over which it and the offset behave nicely. The breakpoints of the subdivision correspond to special points on the curve for which there are tests. The points to look for are:

(1) Ordinary cusp of offset:	Defined by equation (14.31) and the condition $\kappa'(u) \neq 0$
(2) Extraordinary cusp of offset:	Defined by equation (14.29) and the conditions $\kappa'(u) = \kappa''(u) = 0$
(3) Turning point:	A point where $p'(u)$ is either vertical or horizontal, that is, either $x'(u) = 0$, $y'(u) \neq 0$, or $x'(u) \neq 0$, $y'(u) = 0$.
(4) Inflection point:	A point where $\kappa(t) = 0$, that is, the curve is locally flat.
(5) Vertex:	A point where $d\kappa/ds = 0$, that is, κ has a local extremum.

The planar evolute of the curve is relevant here because one can show that the cusps of an offset lie on the evolute of the original curve (Theorem 9.7.1 in [AgoM05]). The planar evolute is the locus of the centers of curvature and is defined by

$$q(u) = p(u) + \frac{1}{\kappa(u)} n(u) \tag{14.32}$$

The special points defined by (3–5) above (called *characteristic points* in [FarN90a]) are "intrinsic" properties of curves because they involve solving for zeros of $p'(u)$, $\kappa(u)$, $\kappa'(u)$, $\kappa''(u)$, etc. These points are interesting on **both** curves, but one can show that the turning points, inflection points, and vertices of the offset curve correspond to those of the original curve, except that if $\kappa(t) = -1/d$ at a turning point or vertex of p(u), then the corresponding point on $p_d(u)$ is a cusp or an extraordinary cusp. See Theorem 9.7.2 in [AgoM05].

After the special points (1–5) above have been found one has a segmentation of the curve and its offset. The curve can then be approximated over each primitive

segment in a suitable way, say by some polygonal curves. One is not yet done though because trimming needs to be done if there are self-intersections in the offset curve. This involves first finding the intersection points, an interesting problem in its own right, and then trimming the segments between them. See [FarN90a]. One is then finally ready to generate an appropriate path for the NC machine.

Another approach to getting offset curves is suggested by how milling machines work. One rolls a circle of radius d along the curve. This will sweep out a solid region and the offset curve is one of the boundaries of that region. See Figure 14.39. Now the circle of radius d centered at p(u) can be parameterized by

$$\varphi(\theta) = p(u) + d(\cos\theta, \sin\theta). \tag{14.33}$$

The points on the offset curve are therefore those at which the tangent line to $\varphi(\theta)$ is parallel to the tangent line at p(u) = (x(u),y(u)). In other words, we are looking for those points at which

$$(-\sin\theta, \cos\theta) = c(x'(u), y'(u)), \tag{14.34}$$

that is,

$$\tan\theta = -\frac{x'(u)}{y'(u)}.$$

One has to worry about those places where y′(u) is zero, in particular, where there are cusps. See Figure 14.40.

The fact that equations (14.33) and (14.34) involve transcendental functions may not be desirable. Therefore, one may want to reformulate the envelope into a polynomial form by introducing two variables s and t as follows:

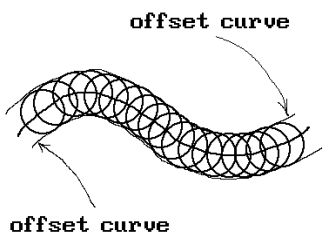


Figure 14.39. Offset curves from milling machine.

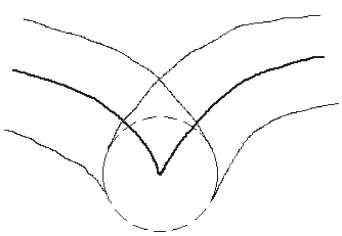


Figure 14.40. A milling machine offset curve problem.

Figure 14.41. Offset curves for multi-axes milling machines.

$$(s - x(u))^2 + (t - y(u))^2 - d^2 = 0. \tag{14.35}$$

Condition (14.34) then translates into the constraint that the tangent vector is orthogonal to the radius vector, namely,

$$p'(u) \bullet (s - x(u), t - y(u)) = 0. \tag{14.36}$$

As a final application involving offset curves, consider the problem of milling free-form surfaces. This sort of operation requires minimally a three-axis milling machine that can move in three orthogonal directions. See Figure 14.41(a). Better yet is a five-axis milling machine that has an additional two degrees of freedom to allow rotations specified by two angles. See Figure 14.41(b). In any case, to carry out the milling, one can first mill along the boundary curves of the patch. Then along an offset curve in the interior of the patch that is an offset of the boundary. We can continue this way until the whole surface is milled. This process involves defining offset curves for space curves. This time we do not have well-defined normals to offset along since there is a whole normal plane at each point of the curve. However, if we look at circles of radius d in these normal planes, what we want is that the offset curve intersects that circle at some point. It is not hard to write down the appropriate equations.

Next, let p(u) be a space curve and (T(u),N(u),B(u)) its Frenet frame. The principal normal N(u) and binormal B(u) form an orthonormal basis for the normal plane to the curve at p(u).

Definition. An *offset curve* $p_d(u)$ to the space curve p(u) at a distance $d > 0$ from p(u) is defined by an equation of the form

$$p_d(u) = p(u) + d(\cos\theta N(u) + \sin\theta B(u)), \tag{14.37}$$

where θ is a function of u in general.

Approaches to computing offset curves for a surface patch parameterized by a function p(u,v) are discussed in [HosL93]. One practical problem with the formulation above is that the principal normal and binormal are not easy to compute.

This concludes our discussion of offset curves. For additional facts see [FarN90a], [FarN90b], or [MaeP93]. Offsets of clothoidal splines are discussed in [MeeW90]. For an overview of a different approach, where one tries to approximate the offset rather than represent it exactly, see [ElLK97]. The motivation is that working with approxi-

mations is not as computationally expensive. The analysis of different algorithms in [ElLK97] showed that the one described in [TilH84] performed best on piecewise quadratic curves. The question of which curves have rational offsets has also been studied in various papers, because otherwise one is basically only left with approximations. For more information about this subject see [Faro92], [Pott95], [Lü95], or [FarS95].

14.9.2 Offset Surfaces

We have already talked about offset-type surfaces when we considered bump mappings in Section 9.8. Here we give a few definitions and state some properties of the basic offset surfaces.

Suppose that p(u,v) is a regular parameterization for a surface **S**. If p(u,v) is differentiable, then

$$N(u,v) = \frac{\partial p}{\partial u}(u,v) \times \frac{\partial p}{\partial v}(u,v) \tag{14.38}$$

is the standard normal vector to the surface. Since N(u,v) does not vanish,

$$\mathbf{n}(u,v) = \frac{N(u,v)}{|N(u,v)|} \tag{14.39}$$

is a well-defined unit normal vector to **S** at p(u,v). Let d be any nonzero real number.

Definition. The *offset surface* $p_d(u,v)$ to p(u,v), which is a distance |d| from p(u,v) is defined by

$$p_d(u,v) = p(u,v) + d\mathbf{n}(u,v). \tag{14.40}$$

Offset surfaces are called parallel surfaces in differential geometry. Just as in the case of offset curves, the reader is assumed to have familiarity with some basic differential geometry. The relevant material for this section can be found in Section 9.14 in [AgoM05] along with proofs of many of the mathematical assertions made here.

There are formulas that express the basic intrinsic geometric properties of an offset surface in terms of the corresponding properties of the original surface. We list them here. First of all, the normal vector

$$N_d(u,v) = \frac{\partial p_d}{\partial u}(u,v) \times \frac{\partial p_d}{\partial v}(u,v)$$

to the surface $p_d(u,v)$ is given by the following formula

$$N_d(u,v) = (1 - 2Hd + Kd^2)N(u,v), \tag{14.41}$$

where K and H are the Gauss and mean curvature of **S**, respectively. Let

$$\mathbf{n}_d(u,v) = \frac{N_d(u,v)}{|N_d(u,v)|}$$

and define σ by $\mathbf{n}_d = \sigma\mathbf{n}$. Notice that σ may equal -1 because the orientation of the offset surface may not be the same as that of the original surface. The principal normal curvatures $(\kappa_i)_d$ for the offset surface are given by

$$(\kappa_1)_d = \frac{\sigma\kappa_1}{1-d\kappa_1} \quad \text{and} \quad (\kappa_2)_d = \frac{\sigma\kappa_2}{1-d\kappa_2}. \tag{14.42}$$

The Gauss curvature K_d and mean curvature H_d of the offset surface can be computed by

$$K_d = \frac{K}{1-2Hd+Kd^2} \tag{14.43}$$

and

$$H_d = \sigma\frac{H-Kd}{1-2Hd+Kd^2}. \tag{14.44}$$

Clearly, all the problems that could arise in the context of offset curves are magnified for offset surfaces. There could again be cusps. We could have ridges and, of course, complicated self-intersections. Regions of high curvature and where the offset distance is close to the minimum concave radius of curvature cause problems. Barnhill and Frost ([BarF95]) analyzed three approaches to offset surfaces via approximations based on uniform bicubic Hermite meshes, NURBS surfaces, and uniform bicubic/biquintic Bézier meshes and then proposed a solution that used triangular Bernstein-Bézier patches. Other approaches for NURBS can be found in [KuSP02] and [KuSP03].

Forsyth ([Fors95]) discusses offsetting and the closely related operation of shelling in a slightly different context. Offsetting is thought of here as an operation on a solid model that adds or removes a uniform layer to its boundary. Shelling comes in two forms. *Closed shelling* is where one removes all of the interior of a solid further than a given distance from its boundary. In *open shelling* one removes all of the solid further than a given distance from a part of its boundary. Figure 14.42 shows two-

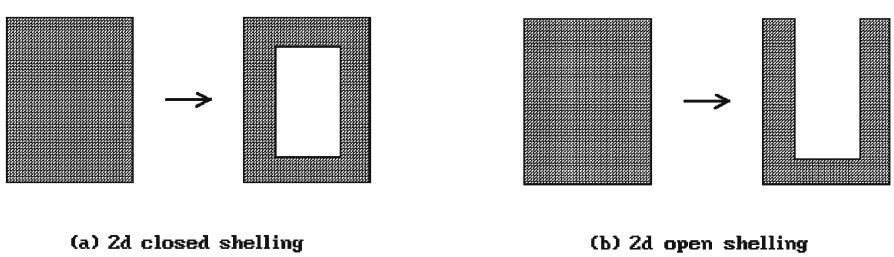


Figure 14.42. Shelling.

dimensional examples of these operations. Clearly, shelling can be defined in terms of standard offset operations. Forsyth describes how to define such offsetting operations for the boundary representation of solids.

14.10 Envelopes

Envelopes are spaces that are a generalization of offset curves or surfaces. These spaces arise from the boundary of regions swept out by moving parts of machinery. Understanding the geometry of envelopes is therefore important in the design of machinery and its operation, such as in the case of NC machines or robots. In particular, it is relevant in making sure that there is adequate clearance of these parts in the work environment. Envelopes have mostly been studied when one is sweeping circles, planes, and spheres. Even there the analysis can get very tricky. A discussion of general envelopes can be found in [Brec92].

Here is a definition for envelopes in the plane.

Definition. Let α_t: $[0,1] \to \mathbf{R}^2$ be a one-parameter family of curves in the plane defined by $\alpha_t(u) = \alpha(u,t)$ for some C^∞ function α: $[0,1] \times [0,1] \to \mathbf{R}^2$. An *envelope* of this family is defined to be a curve p(u) that is **not** a member of this family but which is tangent to some member of the family at every point.

Figure 14.43(a) shows a nicely structured envelope. Figure 14.43(b) shows the envelope of normals to an ellipse whose ends are the centers of the osculating circles. In other words, an envelope can have bad singularities even if we start with nice functions.

One approach to studying the envelope p(u) is to think of p(u) as being the limit as ε approaches 0 of the intersections of p(u) and p(u+ε). Although this has serious problems in general, it seems to work in many cases of interest.

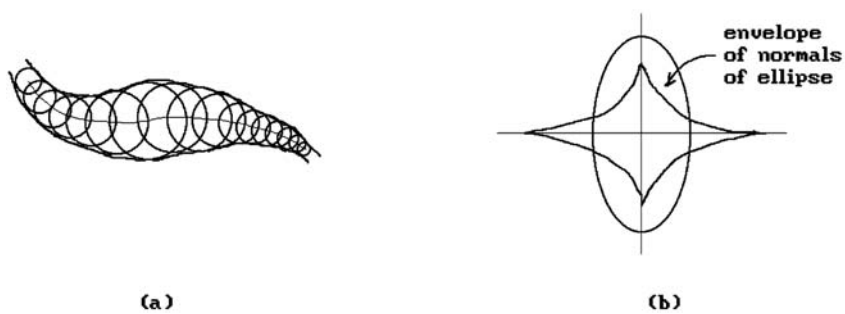


Figure 14.43. Envelopes of curves and normals.

14.11 Exercises

Section 14.2

14.2.1. Carefully describe an algorithm that finds the distance between a point and a polygonal curve.

14.2.2. Find the the point **q** on the curve $p(u) = (u,u^2)$ that is closest to $\mathbf{p} = (-3,1)$.

14.2.3. Find the two nearest points on the curves p: $[-\infty,3] \to \mathbf{R}^2$, $p(u) = (2u,u^2)$, and q: $\mathbf{R} \to \mathbf{R}$, $q(u) = (-u + 1,u + 5)$.

14.2.4. Consider the surface p: $\mathbf{D} \to \mathbf{R}^3$, $p(u,v) = (u,v,u^2 + v^2)$.

(a) Find the the point **q** on the surface that is closest to $\mathbf{p} = (0,3,0)$ if $\mathbf{D} = \mathbf{R}^2$.
(b) Find the the point **q** on the surface that is closest to $\mathbf{p} = (0,3,0)$ if $\mathbf{D} = \mathbf{R}\times(-\infty,-1]$.

14.2.5. Find the two nearest points on the surfaces $p(u,v) = (u,v,u^2+v^2)$ and $q(u,v) = (u,v,2u-6)$.

Section 14.5.1

14.5.1.1 Consider the curve defined implicitly by the equation $y^2 - x^3 = 0$. Resolve the singularity of this curve at (0,0) using the method described in Section 14.5. Work through and explain the details just like we did in Example 14.5.1.2.

Section 14.9.1

14.9.1.1 Consider the planar curve $p(u) = (u,u^2)$. Analyze the offset curve $p_d(u)$ defined by equation (14.28) with respect to cusps, turning points, inflection points, vertices, and self-intersections for the following values of d:

(a) $0 < d < 0.5$
(b) $0.5 < d$
(c) $d = 0.5$

14.12 Programming Projects

Section 14.2

14.2.1. Implement a point-curve distance algorithm for planar

(a) polygonal curves, and
(b) B-spline curves.

Let the user define a curve interactively with a mouse or specify a previously created one. Then let him/her pick a point with the mouse and display both the distance and the point on the curve that is closest to the picked point.

Section 14.4

14.4.1. Implement a trimmed surface algorithm for smooth surfaces with rectangular domains, such as surfaces of rotation, Bézier surfaces, or B-spline surfaces. After a user has selected one, show him/her the domain and let him/her define polygonal curves in that domain. The curves are of two types: those that that define the outer boundary of the final trimmed region and those that specify a hole. After the user has specified the region that is to be trimmed away display the trimmed surface. Develop this program for

(a) polygonal trimming curves, and
(b) B-spline trimming curves with the picked points becoming their control points

Section 14.5.1

14.5.1.1. Implement a marching algorithm for an implicit planar curve that is defined by some predefined polynomial equation.

Section 14.6

14.6.1. Implement a contour program for **graphs** of functions f: $\mathbf{R}^2 \to \mathbf{R}$. After the user defines a polynomial function, display the surface, prompt the user for a value h, and then highlight the contour $f^{-1}(h)$ on the surface.

Section 14.9.1

14.9.1.1. Display offset curves for planar B-spline curves. Allow the user to specify the curve and the d parameter. Check for singularities in the offset curve and mark them for the user to see.

CHAPTER 15

Local Geometric Modeling Topics

Prerequisites: Chapter 4, Sections 8.1–8.5 (parameterization, surface, tangent vectors/planes, manifolds), Sections 9.1–9.4, 9.8, and 9.9 in [AgoM05]

15.1 Overview

This chapter covers a number of applications that involve local properties of surfaces. Section 15.2 discusses some curvature-related topics. Sections 15.3–15.4 deal with geodesics on surfaces. We discuss algorithms for generating both smooth and discrete geodesics and then describe an application to filament winding and tape laying. Section 15.5 is on how one can drop curves onto surfaces, something that is important in robotic applications. Section 15.6 discusses a few blending topics.

15.2 Curvature

Curvature raises its head at many places in CAD and CAGD. It is important to fairing curves and surfaces. We have also seen it in the context of finding efficient polygonal representations for curves and surfaces. For example, if a parametric surface has many relatively flat spots, then using a polygonization of the surface that is obtained by evaluating points at a uniform grid in parameter space would be very wasteful. To minimize the space for the data structure one would want a relatively coarse subdivision for flat regions and only use a much finer subdivision at the more curved parts. Clearly, flatness is a function of curvature. This section presents a sample of the curvature topics that one finds in the literature by summarizing the findings of several technical papers that have been published on the subject. The interested reader can find other references in those papers.

To begin with, there is the notion of curvature itself. In the case of curves, there is really only one concept of curvature and computing it involves the second derivative of the curve. In the case of surfaces, the situation is not as simple because there are a number of different curvature related concepts. The most important is Gauss curvature, but principal and mean curvatures are also useful. To compute these one needs second order partial derivatives. Formulas for computing the various curvatures can be found in Chapter 9 in [AgoM05].

We consider curves first. The fairness of a curve is defined in terms of curvature. See Section 11.12. Typically one seeks curves whose curvature functions are appropriately piecewise monotone. Sapidis ([Sapi92]) describes a simple geometric condition so that a quadratic Bézier curve segment has a monotone curvature function. It is furthermore shown how to move the middle control point of that segment to correct any bad curvature plot it may have initially.

Just because we have an approximation that is within a given tolerance of an object does not mean that the shape of the object has been approximated very well. A curve that wiggles about a straight line would not be a good approximation of the shape of the line. As was indicated in earlier chapters, the choice of metric with respect to which an approximation is defined matters. Wolters and Farin ([WolF97]) describe a metric based on total curvature that does a better job in approximating shape.

Miura ([Miur00]) proposes a new type of curve whose curvature is easier to manipulate than that of the more traditional curves. The method is based on integrating tangent vectors, specifically unit tangent vectors. In order to make this easier for the user to define such vectors, the author's system asks the user to pick points on the unit sphere in an interactive way. The selected vectors are interpolated by thinking of them as unit quaternions. This construction is the analog of the clothoid construction that has been used to manipulate plane curves in a way that controls curvature properties. Miura calls his curve a *unit quaternion integral curve*.

Next, consider surfaces. Krsek et al. ([KrLM98]) describe methods for computing curvature quantities from discrete data. The approach is to approximate the data by second order curves or surfaces. Higher orders did not seem to lead to much improvement. They describe various methods but analyze

The circle fitting method: This turns out to be the fastest.

The paraboloid fitting method: This is slower than the circle fitting method but more accurate on noisy data.

The Dupin cyclide method: This is the slowest but is usually more accurate than the other two methods.

Wollmann ([Woll00]) also tries to estimate curvature values for a discrete surface. The method is based on getting estimates to the curvature of curves and using Euler's and Meusnier's theorem.

Meek and Walton ([MeeW00]) analyze the accuracy of various approaches to the problem of getting approximations to surface normals and Gauss curvature given a surface defined by a set of discrete points. The assumption is that one has accurate data for a smooth surface such as one would get from sampling points on a real object. They analyzed the following methods for finding an approximation to the surface normal and/or the Gauss curvature at a point $\mathbf{p}$:

(1) Fitting a quadratic surface to the given data near **p** and using its normal and Gauss curvature as approximations.
(2) Approximating the surface at **p** by a set of triangles incident to **p** and using various types of averages of their normals.
(3) Approximating the Gauss curvature at **p** by discretizing its definition based on the Gauss map where one thinks of it as a limit of the quotient of small areas containing **p** and the area of their images on the unit sphere. See equation (9.42) in Chapter 9 in [AgoM05].
(4) Approximating the Gauss curvature at **p** by means of the angle deficit method.

One surprising conclusion was that the popular method (4) is not always very accurate.

Andersson ([Ande93]) discusses how one could design a surface by modifying its curvature. This is carried out in terms of solutions to boundary value problems for partial differential equations.

Ye ([Ye96]) points out how a color-coded Gauss curvature map can be used to judge the fairness of a surface. A smoothly varying map is good and rapidly varying ones are bad. Ye answers the following question about the fairness of a surface where two patches meet:

> **Question:** Can the curvature continuity between the patches be visualized by means of the Gauss curvature? Alternatively, if two patches are tangent-plane continuous along their common edge and they have the same Gaussian curvature along the common edge, are they curvature continuous there?

There is a similar question for mean curvature. Let $\kappa_n(\mathbf{C},\mathbf{S},\mathbf{p})$ denote the normal curvature at a point **p** of a curve **C** lying in surface **S**.

Definition. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be surfaces that are tangent-plane continuous along a curve **C**. The surfaces are said to be *curvature continuous* along **C** if for all curves $\mathbf{C}_1$ and $\mathbf{C}_2$ on $\mathbf{S}_1$ and $\mathbf{S}_2$, respectively, that meet and are tangent at a point **p** on **C** we have that $\kappa_n(\mathbf{C}_1,\mathbf{S}_1,\mathbf{p}) = \kappa_n(\mathbf{C}_2,\mathbf{S}_2,\mathbf{p})$.

Ye gives an answer to the question in terms of Dupin indicatrices, principal curvatures, and Gauss and mean curvatures. Mean curvatures turn out to be a better way to measure curvature continuity.

Kaklis and Ginnis ([KakG96]) address the problem of constructing shape-preserving C^2 surfaces that interpolate point sets lying on parallel planes. Call the planar curves $p_i(u)$ interpolating the data of a given plane a *skeletal line*. We are basically looking for a skinning surface $p(u,v)$ for the skeletal lines $p_i(u)$. Kaklis and Ginnis describe how one can get a skinning surface that has the property that if the curvature of adjacent curves $p_i(u)$ and $p_{i+1}(u)$ has the same sign over an interval $[u_j,u_{j+1}]$, then the curvature of all the curves $p(u,v)$, $v \in [v_i,v_{i+1}]$ also has the same sign over the interval $[u_j,u_{j+1}]$.

Wolter and Tuohy ([WolT92]) describe how to compute curvatures for degenerate surface patches.

Other aspects of surfaces that are sometimes interesting are their lines of curvature. Analyzing these involves solving differential equations. See [BeFH86]. Lines of curvature are used to define principal patches.

Finally, is there a notion of curvature in the case of polygonal objects? Such a notion would be defined at vertices and would be a function involving the angle between adjacent edges for curves and the sum of the angles of the faces meeting at a vertex for surfaces.

15.3 Geodesics

15.3.1 Generating Smooth Geodesics

The mathematics of geodesics for surfaces in $\mathbf{R}^3$ is discussed in Section 9.10 in [AgoM05]. The mathematical definition of a geodesic is that it is a **function** (a parameterized curve) defined by second-order differential equations. This is what we shall mean by the term "geodesic," but it is sometimes used more loosely, for example, to refer to the underlying set that is traced out by a geodesic. A common statement is that a straight line in the plane is a geodesic, but one needs to remember that a line can be parameterized in many ways and only some of those parameterizations would actually fulfill the mathematical definition of a geodesic. Two other definitions that are given sometimes (and that consider geodesics as sets rather than maps) are:

The Kinematic Definition. A *geodesic* is a curve traversed by a particle whose acceleration vector at a point lies in the plane spanned by the velocity vector and the normal to the surface at that point. There is no "side-to-side" acceleration. Any acceleration that there is, is used to keep the particle in the surface or to speed it up or slow it down in the direction of the path.

The Static Force Definition. On a convex surface, a curve is called a *geodesic* if a thread stretched along the path it traces out on the surface is in static equilibrium with respect to any sideways tension on it.

A true geodesic would satisfy both of these criteria. However, a geodesic in the kinematic or static force sense would not necessarily be a real geodesic since its acceleration vector might not be orthogonal to its velocity vector. Nevertheless, by Theorem 9.10.11 in [AgoM05] it does trace out a geodesic path.

Note. The boundary of a surface causes technical problems for the definition of a geodesic because one often needs derivatives to be defined in open neighborhoods of a point. To avoid such problems, we shall assume throughout this section that either our surfaces have no boundary or that all the curves being defined are well away from the boundary.

Consider a surface patch $\mathbf{S}$ in $\mathbf{R}^3$ parameterized by

$$\varphi : [c,d] \times [e,f] \to \mathbf{S}$$

Any curve $\gamma(t)$ in $\mathbf{S}$ can be expressed in the form $\gamma(t) = \varphi(\alpha(t))$, where

$$\alpha : [a,b] \to [c,d] \times [e,f].$$

(If we were given γ first we could define $\alpha = \varphi^{-1} \circ \gamma$.) See Figure 15.1. Let

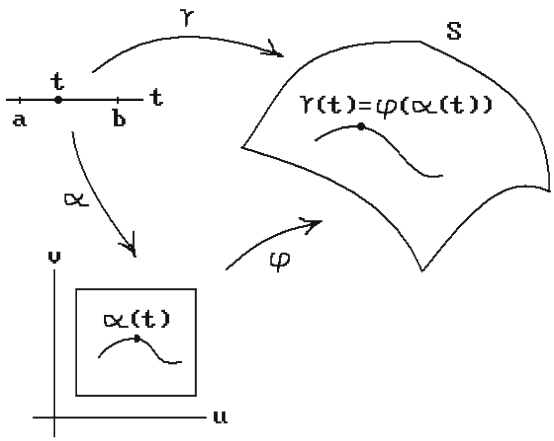


Figure 15.1. Curve in parameterized surface.

$$\varphi(u,v) = (\varphi_1(u,v), \varphi_2(u,v), \varphi_3(u,v))$$

and

$$\alpha(t) = (\alpha_1(t), \alpha_2(t)).$$

We know that

$$\frac{\partial\varphi}{\partial u} = \left(\frac{\partial\varphi_1}{\partial u}, \frac{\partial\varphi_2}{\partial u}, \frac{\partial\varphi_3}{\partial u}\right) \quad \text{and} \quad \frac{\partial\varphi}{\partial v} = \left(\frac{\partial\varphi_1}{\partial v}, \frac{\partial\varphi_2}{\partial v}, \frac{\partial\varphi_3}{\partial v}\right)$$

form a basis for the tangent space at every point of the surface and

$$\mathbf{n} = \frac{\partial\varphi}{\partial u} \times \frac{\partial\varphi}{\partial v}$$

is a normal vector at those points. The chain rule implies that

$$\begin{aligned}
\gamma'(t) &= D\varphi(\alpha'(t)) \\
&= \alpha'(t) J\varphi(u,v)^T \\
&= (\alpha_1'(t) \quad \alpha_1'(t)) \begin{pmatrix} \frac{\partial\varphi_1}{\partial u}(u,v) & \frac{\partial\varphi_2}{\partial u}(u,v) & \frac{\partial\varphi_3}{\partial u}(u,v) \\ \frac{\partial\varphi_1}{\partial v}(u,v) & \frac{\partial\varphi_2}{\partial v}(u,v) & \frac{\partial\varphi_3}{\partial v}(u,v) \end{pmatrix}
\end{aligned} \tag{15.1}$$

where $J\varphi$ is the Jacobian matrix for φ. It follows that

$$\gamma'' = \alpha'' J\varphi^T + \alpha'(J\varphi)'^T \tag{15.2}$$

and

$$(J\varphi)' = \begin{pmatrix} \frac{\partial^2 \varphi_1}{\partial u^2}\alpha_1' + \frac{\partial^2 \varphi_1}{\partial u \partial v}\alpha_2' & \frac{\partial^2 \varphi_1}{\partial u \partial v}\alpha_1' + \frac{\partial^2 \varphi_1}{\partial v^2}\alpha_2' \\ \frac{\partial^2 \varphi_2}{\partial u^2}\alpha_1' + \frac{\partial^2 \varphi_2}{\partial u \partial v}\alpha_2' & \frac{\partial^2 \varphi_2}{\partial u \partial v}\alpha_1' + \frac{\partial^2 \varphi_2}{\partial v^2}\alpha_2' \\ \frac{\partial^2 \varphi_3}{\partial u^2}\alpha_1' + \frac{\partial^2 \varphi_3}{\partial u \partial v}\alpha_2' & \frac{\partial^2 \varphi_3}{\partial u \partial v}\alpha_1' + \frac{\partial^2 \varphi_3}{\partial v^2}\alpha_2' \end{pmatrix}.$$

The condition on $\gamma''(t)$ which makes the curve $\gamma(t)$ into a geodesic is that

$$\gamma''(t) \bullet \frac{\partial \varphi}{\partial u}(\alpha(t)) = 0, \tag{15.3a}$$

$$\gamma''(t) \bullet \frac{\partial \varphi}{\partial v}(\alpha(t)) = 0. \tag{15.3b}$$

This would lead to having to solve a second order differential equation for α. Instead, by introducing a new variable $\alpha'(t) = \beta(t)$, one turns the system (15.3) into a system of first order differential equations, which is the usual preferred approach. See [PFTV86], for example, for ways to solve such systems of equations.

Next, assume that we only want a geodesic in the kinematic or static force sense. In this case, equations (15.3) get replaced by the single equation

$$\gamma''(t) \bullet \mathbf{b}(t) = 0 \tag{15.4}$$

where

$$\mathbf{b}(t) = \mathbf{n}(\gamma(t)) \times \gamma'(t).$$

Of course, as was pointed out earlier, such a curve $\gamma(t)$ may not mathematically be a geodesic since $\gamma''(t)$ may not be a normal to the surface, but it will trace out a geodesic path. Now, assume that the curve $\alpha(t)$ above is an arc-length parameterization of a curve, so that $|\alpha'(t)| = 1$. Let us parameterize the unit vectors $\alpha'(t)$ by the turning angle $\theta(t)$. In other words, write

$$\alpha'(t) = (\cos\theta(t), \sin\theta(t)). \tag{15.5}$$

It follows from (15.5) and the chain rule that

$$\alpha''(t) = \theta'(t)(-\sin\theta(t), \cos\theta(t)). \tag{15.6}$$

If we replace γ'' in equation (15.4) by the right-hand side of equation (15.2) and also replace α'' by the right hand side of equation (15.6), then one can solve for $\theta'(t)$. In fact, it is easy to show that solving (15.4) is equivalent to solving the following system of equations:

$$\begin{aligned}
\theta' &= \frac{-(\cos\theta, \sin\theta)(J\psi)'^{T} \bullet \mathbf{b}}{(-\sin\theta, \cos\theta)(J\psi)^{T} \bullet \mathbf{b}} \\
\alpha_1' &= \cos\theta \\
\alpha_2' &= \cos\theta.
\end{aligned} \tag{15.7}$$

We shall return to this equation in Section 15.4.

Now what we have described so far are just local conditions for geodesics. The curves must satisfy the differential equations shown above in a neighborhood of any point through which they pass. Furthermore, there are of course many solutions to these equations and to get unique solutions one needs to specify additional constraints. The most common constraints, and the ones handled most easily with standard numerical techniques for solving differential equations, are initial conditions. Typical initial conditions would be a start point of the desired curve and a direction vector. However, this does not solve the problem of finding a shortest curve between two points because we would not know the initial direction of the curve. The shortest curve problem is a boundary value problem and much more difficult. A solution to the discrete version of this problem is described in the next section.

One area where one has to deal with geodesics is in the design and manufacture of composite materials. See Section 15.4 below. In this case one wants to generate geodesics given a start point and an initial direction. A common approach is to tessellate the surface and generate geodesics on the resulting polygonal surface. The paper [KSHS03] describes differential equations for a geodesic obtained from a variational approach and compares the numeric solution to these equations to the discrete geodesics one can generate on the approximating polygonal surface using two different algorithms. It turns out that the deviation of the discrete geodesics from the smooth geodesic is not always proportional to the error caused by the tessellation but depends also on the complexity of the surface.

We finish this section with an example. Unfortunately, just as very few curves have a simple formula for their length, very few geodesics have a simple formula. Nevertheless, the following may help the reader understand the mathematics.

15.3.1.1 Example. Consider the paraboloid of revolution **S** defined by the equation

$$f(x,y,z) = z - x^2 - y^2$$

and parameterization

$$\varphi(u,v) = \left(u, v, u^2 + v^2\right), \quad (u,v) \in \mathbf{R}^2.$$

We want to compute the equations that define the geodesics for this surface.

Solution. Let $\alpha(t) = (u(t),v(t))$ be a curve in the domain of φ. First of all, observe that

$$\frac{\partial\varphi}{\partial u} = (1,0,2u) \quad \text{and} \quad \frac{\partial\varphi}{\partial v} = (0,1,2v),$$

so that

$$J\varphi = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2u & 2v \end{pmatrix} \quad \text{and} \quad (J\varphi)' = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 2 & 2 \end{pmatrix}.$$

If $\gamma(t) = \varphi(\alpha(t))$, then

$$\begin{aligned} \gamma'' &= (u'' \; v'')\begin{pmatrix} 1 & 0 & 2u \\ 0 & 1 & 2v \end{pmatrix} + (u' \; v')\begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 2 \end{pmatrix} \\ &= (u'', v'', 2uu'' + 2vv'' + 2u' + 2v'). \end{aligned}$$

It follows from equation (15.3) that the geodesics of **S** are defined by curves $\alpha(t)$ satisfying

$$\begin{aligned} (1+4u^2)u'' + 4uvv'' + 4uu' + 4uv' &= 0 \\ 4uvu'' + (1+4v^2)v'' + 4vu' + 4vv' &= 0. \end{aligned}$$

We shall not solve that system of differential equations, but will show that certain longitudinal curves (meridians) are geodesics in the kinematic sense.

Let $\mathbf{e} = (\cos\theta, \sin\theta, 0)$ be a fixed unit vector in the plane and consider the curve $\gamma(t)$ in **S** defined by

$$\gamma(t) = t\mathbf{e} + t^2(0,0,1) = (t\cos\theta, t\sin\theta, t^2).$$

The curve traces out the intersection of the vertical plane through the origin and **e** with our surface **S**. See Figure 15.2. It is easy to check that

$$\gamma'(t) = (\cos\theta, \sin\theta, 2t) \quad \text{and} \quad \gamma''(t) = (0,0,2).$$

Now,

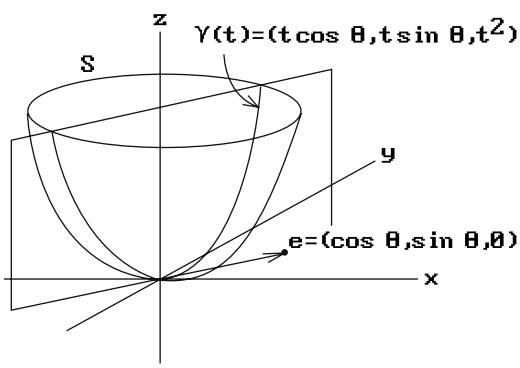


Figure 15.2. Longitudinal curves that are kinematic geodesics.

$$\mathbf{n}(\gamma(t)) = \frac{\partial\varphi}{\partial u}(t\cos\theta, t\sin\theta) \times \frac{\partial\varphi}{\partial v}(t\cos\theta, t\sin\theta) = (-2t\cos\theta, -2t\sin\theta, 1)$$

and

$$\mathbf{b}(t) = \mathbf{n}(\gamma(t)) \times \gamma'(t) = \left(1+4t^2\right)(-\sin\theta, \cos\theta, 0).$$

Clearly, $\mathbf{b}(t) \bullet \gamma''(t) = 0$, so that $\gamma(t)$ is a kinematic geodesic (but not a geodesic since $\gamma''(t)$ is not a normal to the surface).

15.3.2 Generating Discrete Geodesics

The last section dealt with smooth geodesics and had to do with smooth curves lying in smooth surfaces. What if our surface is not smooth but polygonal? See [KSHS03] for a discussion of two algorithms that generate discrete geodesics given a start point and a start direction. It is obvious how to proceed here except in the case where the geodesic passes through a vertex (we preserve angles when crossing edges just like in the plane). The decision made at a vertex is especially critical if the surface came from a tessellation of a smooth surface and we are trying to approximate a smooth geodesic. One needs to take into account the discrete curvature at the point that could be defined in terms of the sum of angles around the point or the normals to the faces that meet at the point.

In this section we discuss the other and much harder variant of the geodesic problem where we are given two endpoints and we want to find the locally shortest polygonal curves connecting the two points. Before establishing that this is a meaningful question and getting to the main results, we need to introduce some terminology. In particular, we pause to make the definition of a polygonal curve (to be called a piecewise linear curve) more precise. It is hoped that the reader will not despair because of the rather large number of detailed definitions of what may seem like obvious terms, but that is the cross that one has to bear when one tries to be unambiguous. The requirement that piecewise linear curves are defined by a sequence of at least **two** points is motivated by the fact that other definitions are then less complicated.

Definition. A *piecewise linear curve* or *pwl curve* p in $\mathbf{R}^n$ is a sequence $(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_k)$, $k > 0$, of points of $\mathbf{R}^n$. The *length* of p, Length(p), is defined using the standard Euclidean metric by

$$\text{Length}(p) = |\mathbf{p}_0\mathbf{p}_1| + |\mathbf{p}_1\mathbf{p}_2| + \ldots + |\mathbf{p}_{k-1}\mathbf{p}_k|.$$

The curve p is said to be *closed* if $\mathbf{p}_0 = \mathbf{p}_k$. The *path* or *underlying space* of p, $|p|$, is defined by

$$|p| = [\mathbf{p}_0, \mathbf{p}_1] \cup [\mathbf{p}_1, \mathbf{p}_2] \cup \ldots \cup [\mathbf{p}_{k-1}, \mathbf{p}_k].$$

If $\mathbf{X} \subseteq \mathbf{R}^n$ and $|p| \subseteq \mathbf{X}$, then we say that p is a pwl curve *in* $\mathbf{X}$. The pwl curve p is said to be *simple* if

(1) all the points $\mathbf{p}_i$, except possibly the first and last, are distinct, and
(2) no segments $[\mathbf{p}_i,\mathbf{p}_{i+1}]$ and $[\mathbf{p}_j,\mathbf{p}_{j+1}]$, $0 \le i < j < k$, intersect unless

(a) $j = i + 1$, in which case they intersect in the point $\mathbf{p}_{i+1}$, or
(b) the curve is closed, $i = 0$, and $j = k - 1$, in which case they intersect in the point $\mathbf{p}_0 = \mathbf{p}_k$.

If $\mathbf{q} \in [\mathbf{p}_i,\mathbf{p}_{i+1}]$, then the pwl curve $(\mathbf{p}_0,\mathbf{p}_1, \ldots ,\mathbf{p}_i,\mathbf{q},\mathbf{p}_{i+1}, \ldots ,\mathbf{p}_k)$ is called an *elementary subdivision* of the pwl curve p and a *proper elementary subdivision* if $\mathbf{q}$ lies in the interior of the segment $[\mathbf{p}_i,\mathbf{p}_{i+1}]$. A pwl curve that is obtained from p by a sequence of (proper) elementary subdivisions is called a (*proper*) *subdivision* of p.

If we think of pwl curves as defining a path that one walks along, then in the case of a simple pwl curve there is no backtracking or self-intersection. In particular, a simple pwl curve traces out a one-dimensional manifold. (See Lemma 15.3.2.1 below.) Clearly, proper subdivisions of simple curves are again simple.

It is easy to parameterize the path of a pwl curve.

Definition. Let $p = (\mathbf{p}_0,\mathbf{p}_1, \ldots ,\mathbf{p}_k)$ be a pwl curve and let

$$\rho_i : \left[\frac{i}{k}, \frac{i+1}{k}\right] \to [\mathbf{p}_i, \mathbf{p}_{i+1}] \quad , \quad 0 \le i < k,$$

be the standard linear map that, using barycentric coordinates, has the form

$$\frac{i}{k}s + \frac{i+1}{k}t \to s\mathbf{p}_i + t\mathbf{p}_{i+1}, \quad s, t \ge 0, \quad s + t = 1.$$

The map

$$\rho : [0,1] \to |p|,$$

where

$$\rho \,|\left[\frac{i}{k}, \frac{i+1}{k}\right] = \rho_i,$$

is called the *standard parameterization* of the pwl curve p. If $s, t \in [0,1]$ and $s \le t$, then define a pwl curve q, called the pwl curve *induced by [s,t]* with respect to the standard parameterization, as follows:

Case $s = t$: Set $q = (\rho(s), \rho(t))$.

Case $s < t$: Define i and j by the condition that $s \in \left[\frac{i}{k}, \frac{i+1}{k}\right)$ and $t \in \left(\frac{j}{k}, \frac{j+1}{k}\right]$. If $i = k - 1$, then $q = (\rho(s), \rho(t))$. Otherwise, $q = (\rho(s),\mathbf{p}_{i+1},\mathbf{p}_{i+2}, \ldots ,\mathbf{p}_j,\rho(t))$.

15.3.2.1 Lemma. Let $p = (\mathbf{p}_0,\mathbf{p}_1, \ldots ,\mathbf{p}_k)$ be a pwl curve and ρ its standard parameterization.

(1) The map ρ is continuous.

(2) If p is simple, then $\rho|(0,1)$ is a homeomorphism between $(0,1)$ and $|p| - \{\mathbf{p}_0, \mathbf{p}_k\}$. If p is simple and not closed, then ρ is a homeomorphism between $[0,1]$ and $|p|$.

Proof. Easy.

Definition. Let p be a pwl curve and ρ its standard parameterization. Let **p** and **q** be two points in the path of p. Choose s and t, so that $s \le t$, $\mathbf{p} = \rho(s)$, and $\mathbf{q} = \rho(t)$. The set $\rho([s,t])$ is called *the part of the path of p from **p** to **q***.

The part of a path of a pwl curve is not well-defined in general because there may be many choices for the parameters s and t (the curve may backtrack on itself, for example). However, for simple curves it is well-defined unless the curve is closed and one of the points is the first or last point of the curve.

After these preliminary definitions, we come to the first basic fact about shortest curves, namely, that they exist.

15.3.2.2 Theorem. Let **S** be a connected compact polygonal surface.

(1) Any two points of **S** can be connected by a shortest pwl curve, meaning that any other pwl curve between the points will have a length that is larger than or equal to the length of that curve. In fact, a shortest pwl curve between two points will have length less than or equal to the length of any rectifiable curve between those two points, not just pwl curves.

(2) Every shortest pwl curve between two points of **S** is a simple curve but there may be more than one shortest curve between two points.

(3) There is a $\delta > 0$, so that, for any two points **p** and **q** in **S** with $|\mathbf{pq}| < \delta$, there is a unique shortest pwl curve from **p** to **q**.

Proof. By cutting along edges one can flatten the whole surface out in the plane, so that one can study curves on the surface by studying curves in planar polygons. It is a well-known fact that the shortest parametric curve between two points in the plane traces out the segment between the points.

Just like in the smooth case, shortest pwl curves are a special case of a more general type of pwl curve.

Definition. A pwl curve p in a polygonal surface **S** is called a *discrete geodesic* if it is locally the shortest pwl curve. More precisely, there is a $\delta > 0$, so that, for any s, $t \in [0,1]$, $s \le t$, with $|s - t| < \delta$, the pwl curve induced by $[s,t]$ with respect to the standard parameterization ρ of p is a shortest pwl curve between $\rho(s)$ and $\rho(t)$.

Intuitively, a discrete geodesic has the property that if two points **p** and **q** in its path are sufficiently close, then the part of the path from **p** to **q** is the path of a shortest pwl curve from **p** to **q**. Just like in the smooth case, geodesics are not necessarily the shortest curves between points. For example, on a cube a geodesic between two points may pass those points more than once as it wraps around the cube multiple times. On the other hand, it is obvious that every shortest pwl curve is a discrete geodesic.

The discrete geodesic problem: Given two points **p** and **q** on a polygonal surface **S**, find a shortest pwl curve in **S** from **p** to **q**.

In the rest of this section we shall assume the following:

(1) All surfaces **S** will be compact, connected, triangulated, and without boundary.
(2) All pwl curves in a surface **S** will have the property that if they meet a vertex of **S**, then that vertex is a vertex for the curve.
(3) All pwl curves are assumed to be **simple**. All subdivisions will be proper.

Surfaces are assumed to be triangulated because triangular faces simplify arguments. Fortunately, a polygon with v vertices and no holes or self-intersections can always be triangulated in time O(v) (see Section 17.6). Furthermore, if a surface has e edges, then it will have O(e) edges after triangulation. However, there is one point that needs to be kept in mind with respect to the triangulation hypothesis. The number of faces may have increased substantially. Therefore, if the big-O complexity of an algorithm depends on the number of faces, then the triangulation has to be taken into account.

Surfaces are not allowed to have a boundary because, like in the last section, the boundary would cause problems that will not be addressed here. A typical surface would therefore be the surface of a polyhedron in $\mathbf{R}^3$. The assumption about pwl curves being simple is justified by the fact that our goal is to find shortest curves and these are all simple. Finally, note that subdividing either a curve or the surface has no effect on the length of the curves or the geodesics in the surface.

The $O(n^2)$ solution to the discrete geodesic problem for polygonal surfaces with n edges presented by Chen and Han in [CheH90] is too complicated to describe in its entirety here, but we shall sketch the main steps for the convex case. Whether or not a surface is convex has a great influence on the complexity of a shortest curve algorithm. When a surface is not convex, one has to handle some complicating special cases although the basic idea is the same as in the convex case.

The specific shortest path problem that we address is the "single source" shortest path problem, namely, we seek an algorithm for finding a shortest pwl curve from a fixed "start" point **s** of the surface **S** to all other points of **S**. We shall assume that **s** is a vertex of the triangulation of **S**. The Chen and Han algorithm is actually not the most efficient solution to the problem. Kapoor ([Kapo99]) has described an $O(n \log^2 n)$ algorithm, but it is more complicated to describe. A number of other papers prepared the ground for these algorithms. In particular, [MiMP87] is worthwhile reading because it helps one see what the problems are when one is looking for shortest paths. Other references can be found in that paper and in [CheH90]. The solution in [MiMP87], which is not as efficient, is based on an argument that is similar to Dijkstra's single-source shortest path algorithm for graphs.

Definition. Two faces of **S** are *edge-adjacent* if they meet in a common edge. An *edge-adjacent sequence* of faces is a sequence $F = (\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k)$ of faces $\mathbf{f}_i$ with $\mathbf{f}_i$ edge-adjacent to $\mathbf{f}_{i+1}$. If $\mathbf{e}_i$ is the common edge of $\mathbf{f}_i$ and $\mathbf{f}_{i+1}$, then the sequence $E = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{k-1})$ is called the *edge sequence* defined by the sequence F. The sequences F or E are said to be *simple* if no face or edge, respectively, appears more than once in the sequence. Let $\mathbf{q}_i$ be a point in the **interior** of the edge $\mathbf{e}_i$. The pwl curve $q = (\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{k-1})$ is said to *connect* the edge sequence E. A pwl curve q in **S** of the form $(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{k-1}, \mathbf{q}_k)$, where $\mathbf{q}_0$ and $\mathbf{q}_k$ are arbitrary points in $\mathbf{f}_1$ and $\mathbf{f}_k$, respectively, is said to *define* the edge sequence E.

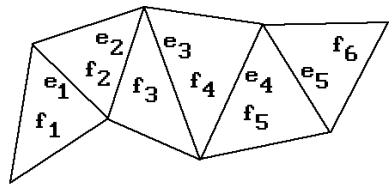


Figure 15.3. An edge-adjacent sequence of faces.

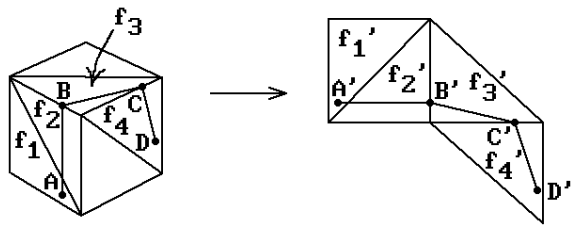


Figure 15.4. Unfolding a curve.

Figure 15.3 shows an edge-adjacent sequence of faces for k = 6 and its associated edge sequence. Given an edge-adjacent sequence of faces $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k$, it is clear that such a sequence can be "unfolded" into the plane determined by the last face. One simply successively rotates the plane of $\mathbf{f}_i$ about $\mathbf{e}_i$ into the plane of $\mathbf{f}_{i+1}$, $i = 1, 2, \ldots, k-1$, so that the image of $\mathbf{f}_i$ under this rotation R_i of $\mathbf{R}^3$ and $\mathbf{f}_{i+1}$ lie on opposite sides of the edge $\mathbf{e}_i$. Define motions M_i by

$$M_i = R_{k-1} \circ R_{k-2} \circ \cdots \circ R_i, \quad i = 1, 2, \ldots, k-1,$$
$$M_k = \text{identity map},$$

and define a map

$$\nu : \bigcup_{i=1}^{k} \mathbf{f}_i \to \text{plane of } \mathbf{f}_k$$

by $\nu|\mathbf{f}_i = M_i$. Next, set $\mathbf{f}_i' = \nu(\mathbf{f}_i) = M_i(\mathbf{f}_i)$. The triangle $\mathbf{f}_i'$ is just the image in the plane of $\mathbf{f}_k$ of the face $\mathbf{f}_i$ under this "unfolding." See Figure 15.4.

Definition. The sequence of triangles $\mathbf{f}_1', \mathbf{f}_2', \ldots, \mathbf{f}_k'$ is called the *planar unfolding* of the edge-adjacent sequence of faces $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_k$. If $\mathbf{X}$ is any subset of the union of the faces $\mathbf{f}_1, \mathbf{f}_2, \ldots,$ and $\mathbf{f}_k$, then the set $\nu(\mathbf{X})$ is called the *unfolding* of $\mathbf{X}$ and the map ν is called the *unfolding map* with respect to the sequence $\mathbf{f}_1, \mathbf{f}_2, \ldots,$ and $\mathbf{f}_k$.

In Figure 15.4, the pwl curve (**A**,**B**,**C**,**D**) unfolds to (**A′**,**B′**,**C′**,**D′**). The unfolding map ν is clearly one-to-one on each face $\mathbf{f}_i$ but may not be globally one-to-one because the faces may wrap around each other. The next two lemmas state some interesting properties of geodesics and their unfoldings.

15.3.2.3 Lemma. A geodesic pwl curve that connects the edge sequence of an edge-adjacent sequence of faces unfolds to a straight line with respect to that sequence.

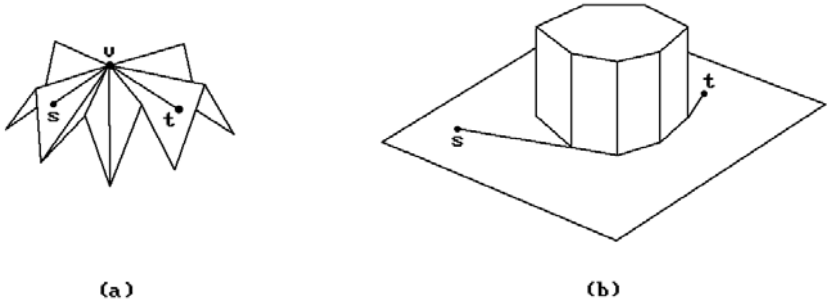


Figure 15.5. Shortest curves on nonconvex surfaces.

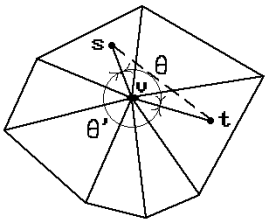


Figure 15.6. The angle at a vertex.

Proof. See [MiMP87]. The fact that the curve intersects the edges in the edge sequence in their interior is needed here.

If our surface is convex, then one can show that, except for possibly its first and last point, a discrete geodesic will not go through any vertices of the surface. Unfortunately, things get more complicated in the nonconvex case. See Figure 15.5. However, a discrete geodesic that passes through a vertex has to satisfy an interesting geometric condition there.

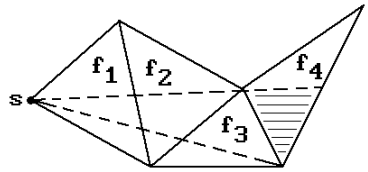
Definition. The *angle* of a face at one of its vertices **v** is the angle between the two edges of the face that meet in **v**.

Suppose that a simple pwl curve goes through a vertex **v**. Let **e** and **e′** be the two distinct adjacent edges of the curve that meet in **v**. By momentarily dividing a face into two, if necessary, we may assume that both **e** and **e′** lie in edges of faces that meet in **v**. Clearly, we can now divide the faces that meet in **v** into two edge-adjacent sequences whose first and last faces meet in edges containing **e** and **e′**. For each of these two sequences add up the angles of their faces at **v**.

Definition. The smaller of the two sums is called the *angle* that the curve makes at **v**.

In Figure 15.6, θ is the angle of the curve at **v**. The reader should not assume that the faces adjacent to a vertex can be flattened out in the plane. Figure 15.5(a) shows an example where this would not be possible.

Figure 15.7. Regions with the same edge sequence.



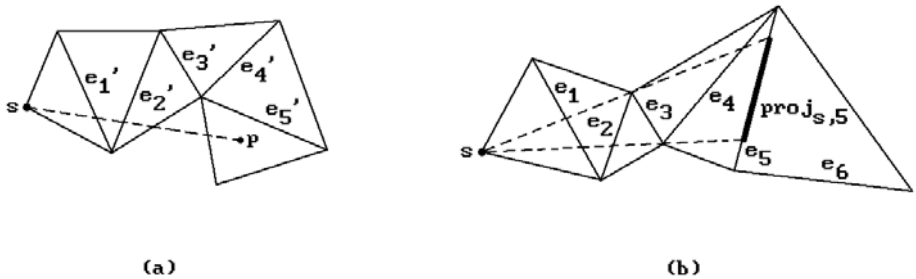


Figure 15.8. Impossible and possible edge sequences for shortest curves.

15.3.2.4 Lemma

(1) A discrete geodesic is a curve that alternates between vertices and (possibly empty) edge sequences such that the unfolded path along any edge sequence is a straight line segment. If the curve passes through any surface vertex, then the angle that the curve makes at that vertex is greater than or equal to π.

(2) If a discrete geodesic between two points is actually the shortest curve between those points, then no edge can appear in more than one edge sequence and each edge sequence must be simple.

Proof. See [MiMP87]. Figure 15.5(b) shows how a sequence of vertices could be part of a geodesic.

The basic idea that will lead to our main theorem (Theorem 15.3.2.6) is to divide the surface into regions, so that the shortest pwl curves from the start point **s** to all the points within a region define the same edge sequence. See the shaded region in Figure 15.7.

Definition. If p is a shortest pwl curve from **s** to a point **p** that defines an edge sequence, then that edge sequence will be called a *shortest edge sequence for* ***p***.

Not every edge sequences is a shortest edge sequence. For example, if we are dealing with a convex surface in Figure 15.8(a), then the unfolded edge sequence $\mathbf{e}_1'$, $\mathbf{e}_2'$, $\mathbf{e}_3'$, $\mathbf{e}_4'$, $\mathbf{e}_5'$ cannot possibly come from a shortest edge sequence. (It could be without the hypothesis of convexity as one can see from Figure 15.5(b).) In the next stage of our discussion we shall assume that surfaces are **convex**. The following facts hold in that case:

15.3.2.5 Lemma. Assume that **S** is a convex surface with n faces.

(1) A shortest path cannot traverse more than n faces.
(2) Two shortest paths from the start point **s** can only intersect in **s** and their endpoint.

Proof. See [ShaS86].

We now look for a criterion that rejects edge-adjacent sequences of faces that cannot possibly come from a shortest curve.

Definition. Given a point **p** and a set **X**, define infCone(**p**,**X**), the *infinite cone on **X** from **p**,* by

$$\inf\mathrm{Cone}(\mathbf{p},\mathbf{X}) = \{\mathbf{p} + s\mathbf{px} \mid s \in [0,\infty), \mathbf{x} \in \mathbf{X}\}.$$

Definition. Given a point **p** and sets **X** and **Y**, define proj(**p**,**X**,**Y**), the *projection of **X** on **Y** from **p**,* by

$$\mathrm{proj}(\mathbf{p},\mathbf{X},\mathbf{Y}) = \inf\mathrm{Cone}(\mathbf{p},\mathbf{X}) \cap \mathbf{Y}.$$

Now let $F_k = (\mathbf{f}_1,\mathbf{f}_2, \ldots ,\mathbf{f}_{k+1})$ be an edge-adjacent sequence of faces in **S** with associated edge sequence $E_k = (\mathbf{e}_1,\mathbf{e}_2, \ldots ,\mathbf{e}_k)$ and unfolding map ν_k. Recall that ν_k maps all faces $\mathbf{f}_i$ into the plane of $\mathbf{f}_{k+1}$. Assume that the start point **s** is the vertex of $\mathbf{f}_1$ opposite the edge $\mathbf{e}_1$. Let $\mathbf{s}'_k$ and $\mathbf{e}'_i$ be the unfolding of **s** and $\mathbf{e}_i$ with respect to ν_k, respectively.

Definition. Define the subsets $\mathrm{proj}_{\mathbf{s},i}$, $1 \le i \le k$, of $\mathbf{e}_i$ recursively as follows:

$$\mathrm{proj}_{\mathbf{s},1} = \mathbf{e}_1,$$
$$\mathrm{proj}_{\mathbf{s},i} = \mathrm{proj}(\nu_i(\mathbf{s}), \nu_i(\mathrm{proj}_{s,i-1}), \mathbf{e}_i), \quad i > 1.$$

Call $\mathrm{proj}_{\mathbf{s},i}$ a *shadow* of the start point **s** on edge $\mathbf{e}_i$. We shall call $\mathbf{f}_{i+1}$ a face *shadowed* by the edge $\mathbf{e}_i$ with respect to the sequence F_i.

Figure 15.8(b) shows what the shadow $\mathrm{proj}_{\mathbf{s},5}$ would look like. Because our surface is convex, we could stop our edge-adjacent sequence of faces at that point since $\mathbf{e}_6$ cannot belong to a shortest edge sequence. In fact, $\mathrm{proj}_{\mathbf{s},6}$ would be the empty set. Basically, $\mathrm{proj}_{\mathbf{s},i}$ defines a cone from **s** in which all shortest paths with edge sequence $(\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3,\mathbf{e}_4,\mathbf{e}_5)$ would have to lie. The algorithm for finding shortest paths for convex surfaces uses these shadows to control the search that one has to perform. Algorithm 15.3.2.1 is an outline of an algorithm for finding shortest edge sequences. One builds a tree whose nodes, other than the root, consist of quadruples of the form $N = (\mathbf{e},\mathbf{f},\mathbf{p},\mathrm{proj}_{\mathbf{s},\mathbf{e}})$, where

e is an edge of the face **f** in the surface **S**,
p is a point in the plane of **f** that is an unfolding of **s**, that is, $\mathbf{p} = \nu(\mathbf{s})$ where ν is an unfolding map to the plane of f, and
$\mathrm{proj}_{\mathbf{s},\mathbf{e}}$ is a subset of **e** that is a shadow of **s**.

Note that a face can be shadowed by each of its three edges.

Definition. The set $\mathrm{cone}(\mathbf{p},\mathrm{proj}_{\mathbf{s},\mathbf{e}}) \cap \mathbf{f}$ is called the *shadow of **s** in N*.

```
Input:     a convex surface S with n faces
           a vertex s of S called the start point
Output:    a node tree T

node = quadruple (e,f,p,proj_{s,e}), where
           e is an edge of the face f in the surface S,
           p is a point in the plane of f that is an unfolding of s,
           proj_{s,e} is a subset of e that is a shadow of s

node tree T;
integer    i;
edge       e′;
point set  X;

Initialize T to consist of simply a root that is a dummy node;
for all faces f that have s as vertex do
    for all edges e of f do Insert (e,f,s,e) as child of root of T;

for i:=1 to n do
    for all leaves N=(e,f,p,proj_{s,e}) of T at ith level do
        begin
            Let f′ be the other face (≠ f) containing the edge e;
            Let e1 and e2 be the edges of f′ other than e;
            Unfold p to p′ in the plane of f′;
            for e′:=e1,e2 do
                begin
                    X := proj (p′,proj_{s,e},e′);
                    if X ≠ ϕ then Insert the node (e′,f′,p′,X) into T as a child of N;
                end
        end;
return T;
```

Algorithm 15.3.2.1. A convex surface edge sequence algorithm.

The tree built by Algorithm 15.3.2.1 is called the *edge sequence tree* for **S** with respect to the start point **s**. The reason for the name is that every path from the root to a node in the tree defines an edge sequence for geodesics from **s** to a point **p** in the node's shadow of **s**. These geodesics are not necessarily shortest paths but there is a unique geodesic with that edge sequence from **s** to every point in the node's shadow of **s**. To find the shortest path from **s** to an arbitrary point **p** in **S** one would first find all the nodes N in the tree, so that **p** is in the shadow of **s** in N. These nodes would define unique geodesics to **p**. We would simply select the shortest of those and that would be the shortest path from **s** to **p**.

We now have a correct algorithm for finding shortest paths. The only problem is that the edge sequence tree might be exponential in size. Figure 15.9(a) shows how

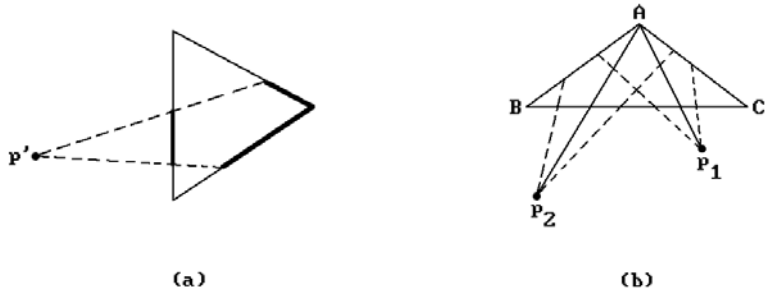


Figure 15.9. Limiting the number of children of a node.

shadows can cover angles opposite an edge and this would give rise to two children to a node. However, our task was to find shortest paths and not all geodesics. Because of this it turns out that the initial edge sequence tree can be pruned. The key observation is that at most one node whose shadows cover the angle needs to be given two children if one is only looking for shortest paths. See Figure 15.9(b). In the figure nodes N_1 and N_2 with the same edge **BC** gave rise to unfolded start points $\mathbf{p}_1$ and $\mathbf{p}_2$ with respect to face **ABC**. If $|\mathbf{p}_1\mathbf{A}| < |\mathbf{p}_2\mathbf{A}|$, then the shorter paths to **s** from points on edges **AB** and **AC** sufficiently close to **A** come from the edge sequence defined by N_1. This means that only N_1 needs to be given two children. If $|\mathbf{p}_1\mathrm{A}| = |\mathbf{p}_2\mathrm{A}|$, then both N_1 and N_2 need only one child. Using this "one angle, one split" criterion leads to a new edge sequence tree generation algorithm that produces a tree with O(n) leaves at each stage. One can reduce the total space used to O(n) by the following trick: if a leaf in the current tree generates only one child, we delete that node and replace it with the child. One can also show that the new tree can be generated in time $O(n^2)$. Once the tree is built, one can find shortest paths in time O(n). To do better, one needs to make some more improvements.

It turns out that the essential information one needs to store is the shortest paths to vertices. Therefore, the authors in [CheH90] create additional "vertex nodes" while they build the edge sequence tree. This can also be done using O(n) space and, given the nature of the tree, one can now find the shortest path to a **vertex** in time O(k), where k is the number of edges in the path.

Now the edge sequence tree certainly has to be built once. This takes time $O(n^2)$. However, after that, what slows down answering shortest path queries to O(n) is finding path information in the tree. By storing that information more efficiently one can speed up multiple queries. An appropriate Voronoi diagram and the subdivision it induces on the surface turns out to do the trick.

First, one cuts the surface along the shortest paths to its vertices. The cut surface can be flattened out into the plane (there may be overlaps). The authors in [CheH90] call the layout one gets the *inward layout* to distinguish it from the one used in [ShaS86]. The start point **s** will map to O(n) vertices $\mathbf{s}_i'$. A Voronoi diagram is built with respect to these image points in the plane and this induces a subdivision of the layout with the property that points that belong to the same region, say the one associated to $\mathbf{s}_i'$, are closer to $\mathbf{s}_i'$ than to any other $\mathbf{s}_j'$, $i \neq j$, and their shortest paths have the same edge sequence. This leaves the question of the complexity of building the Voronoi diagram. Lemma 15.3.2.5(2) implies that all vertices other than the start point

can be listed in a circular order based on the angle that a shortest path to that vertex makes with a fixed line at the start point. Using such an order, connect the flattened vertices by edges in the order in which they are listed. This generates a circular loop called the *equator* that cuts the flattened surface into two regions called the *arctic region* (the one containing the start point) and the *antarctic region*. One can show that the Voronoi diagram is a tree with $O(n)$ edges. Therefore, it can be built in time $O(n \log n)$ and space $O(n)$. The subdivision of the surface induced by the subdivision of the layout can be built in time $O(n^2)$.

This completes the sketch of how one finds shortest paths in convex surfaces. If the surface is not convex, the steps in the algorithm have to be modified but fortunately the complexity of the steps does not change in the end. The final result for surfaces, convex or not, is stated below.

15.3.2.6 Theorem. Let **S** be an arbitrary polygonal surface with n edges and let **p** be a fixed point on **S**. After a preprocessing time of $O(n^2)$ and using space $O(n^2)$, the Chen and Han algorithm answers any query about the shortest distance from a point **q** on **S** to **p** in time $O(\log n)$ and the shortest path from **p** to **q** can be generated in time $O(k + \log n)$, where k is the number of edges intersected by the path.

Proof. See [CheH90].

The Chen and Han algorithm improved on the one in [MiMP87], which was an $O(n^2 \log n)$ algorithm, but it and the better $O(n \log^2 n)$ Kapoor algorithm are too complicated and slow to be practical if one wants accurate answers since one would have to represent computations with a number of bits that is an exponential function of the number of bits in the coordinates of a vertex. For that reason, algorithms have been developed that only give an approximate answer but which are much more efficient. See, for example, the paper by Agarwal et al. ([AgHK00]).

15.4 Filament Winding and Tape Laying

The previous two sections described the mathematics involved in generating geodesics on surfaces. One area of manufacturing where geodesics play a role is in winding filaments or tapes around mandrels to create composite materials. Filament winding is a reinforcement method for creating composites that have high strength and low weight. These materials are created by encasing filaments or fibrous tapes in a resin matrix. The matrix holds the fibers in place, transfers stresses between them, and also seals them from mechanical or environmental damage. The filament or tape and matrix combination is wound around a mandrel that corresponds to the desired shape of the finished material. Surfaces of revolution, or cylindrical surfaces as a special case, are common shapes for these mandrels. At the end, after the material is cured, the mandrel may become part of the finished product or be discarded. The method is widely used in aerospace, hydrospace, military, and many other applications. It is the mathematical aspect of the subject that interests us here, in particular, the role that geodesics play. For manufacturing and other details the

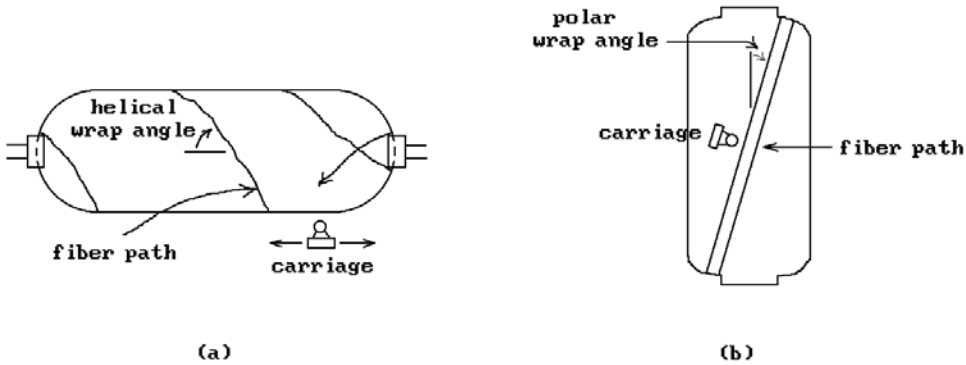


Figure 15.10. Helical and polar filament winding.

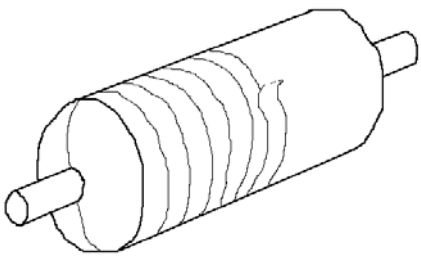


Figure 15.11. Laying tape on a rotating mandrel.

reader should consult appropriate engineering books such as [RosG64] and [Schw97] and related journals.

First, consider the mathematics involved in filament winding. Surfaces of revolution or similar surfaces whose diameter is small compared to the length are good target objects here. Figure 5.10 shows the two common types of filament winding, helical and polar. It is clear why filament paths need to stay close to geodesics. In fact, on a frictionless convex surface it would be impossible to lay a filament in any way other than along a geodesic since those curves correspond to a state of static equilibrium. In the presence of friction the filament becomes "sticky" and its path can deviate from geodesics. The amount of possible deviation would depend on the amount of "stickiness." In practice, the filaments need to be kept tight to minimize slippage, something that is not a problem with tape. Additional problems arise with concave regions in the surface that can cause "bridging." Concave regions are also less of a problem with tape.

Next, we consider tape laying and its mathematics. Figure 15.11 shows the laying of tape along a rotating mandrel. The basic problem with laying a tape along an object is that it may not unroll smoothly, but, depending on the curvature of the surface, may form folds or crinkles in the process. To understand what is going on, one needs to analyze the problem using differential geometry. A piece of tape is a developable surface because it is isometric to a subset of the plane $\mathbf{R}^2$. It follows from Theorem 9.15.5 in [AgoM05] that the only surface on which one can lay a tape in a problem-free manner is a surface whose Gauss curvature is zero at every point. For example, a cylinder is such a surface. On the other hand, since the Gauss curvature of a sphere is nonzero, it is not possible to lay a tape smoothly onto a sphere.

Clearly, if a tape can be laid on a surface **S**, then one can specify a particular placement or *tape path* with a map

$$\varphi : [-d,d] \times [0,L] \to \mathbf{S}, \tag{15.8}$$

where 2d is the width of the tape, L is its length, and

$$\gamma : [0,L] \to \mathbf{S}, \quad \gamma(t) = \varphi(0,t), \tag{15.9}$$

is the curve traced out by the center line of the tape parameterized by arc-length. Now if φ is an isometry, then $\gamma(t)$ is a geodesic in **S** because $0 \times [0,L]$ lies in $[-d,d] \times [0,L]$. This corresponds to using a tape that is "stiff laterally." There is a kind of converse, namely, the centerline of a laterally stiff tape follows a geodesic path. Fix t and define the curve $\eta_t(s)$ by

$$\eta_t(s) = \varphi(s,t).$$

15.4.1 Theorem. If the curve $\eta_c(s)$ is a kinematic geodesic and orthogonal to the curve $\gamma(t)$ at the point $\gamma(c)$ for all c, then $\gamma(t)$ is a kinematic geodesic.

Proof. We need to show that $\gamma''(c)$ is a linear combination of $\gamma'(c)$ and $\mathbf{n}(\gamma(c))$, where $\mathbf{n}(\mathbf{p})$ denotes a nonzero normal vector to **S** at **p**. Since $\eta_c(s)$ is orthogonal to $\gamma(t)$, the vectors $\eta_c'(0)$, $\gamma'(c)$, and $\mathbf{n}(\gamma(c))$ form an orthogonal basis for $\mathbf{R}^3$ at $\gamma(c)$. Next, the assumption that $\eta_c(s)$ is a kinematic geodesic implies that $\eta_c''(0)$ is a linear combination of $\eta_c'(0)$ and $\mathbf{n}(\gamma(c))$. This fact and with the orthogonality of $\eta_c'(0)$, $\gamma'(c)$, and $\mathbf{n}(\gamma(c))$, shows that

$$\eta_c''(0) \bullet \gamma'(c) = 0.$$

Finally, differentiating the equation

$$\eta_c'(0) \bullet \gamma'(c) = 0$$

leads to the following string of equalities:

$$\eta_c'(0) \bullet \gamma''(c) = -\eta_c''(0) \bullet \gamma'(c) = 0.$$

In other words, $\gamma''(c)$ is a linear combination of $\gamma'(c)$ and $\mathbf{n}(\gamma(c))$ and we are done.

Definition. A *natural path* for a tape on a surface **S** is a tape path φ where the center line $\gamma(t)$ is a geodesic and the Gauss curvature at every point of **S** in the image of φ is zero.

Natural paths are the ideal situation, but there are instances where one is willing to put up with less:

(1) There are many surfaces on which one may want to lay tape that have nonzero Gauss curvature. One would have a problem even if one follows a geodesic.
(2) Even on flat surfaces, one may sometimes want to lay tape along lines that are not quite straight.

For these reasons, one is willing to put up with some crinkling or folding; however, the amount of such crinkling or folding that is allowed is something that the user should be able to specify. Therefore, the ability to

(1) specify the actual path and direction angle of the tape, and
(2) predict any gaps, overlaps, and laminate thickness,

would be important to anyone doing off-line programming of the process. In an automated environment a robotic arm that is laying the tape needs to be able to allow for sideways movement and turning.

Note that in the case of filament winding only the path $\gamma(t)$ is important so that the flatness of objects is not so critical. However, filament tows often consists of several filaments and in that case one runs into problems similar to those with tapes because, if a surface is not flat, then adjacent filaments may bulge and move apart if they do not travel equal distances.

From this discussion we see that both in the case of tapes and filaments (those that are sticky or those that are part of multi-filament tows) one is looking for curves that are close, but not necessarily equal to geodesics. Let us show how, given a coefficient of friction μ, it is possible to define *generalized geodesics* ([Crai88]) that are controlled by a "steering function" $s(t)$ satisfying

$$-\mu < s(t) < \mu$$

which will do the job. To find these generalized geodesics we can use the mathematics developed in Section 15.3.1. All we have to do is replace the function $\mathbf{b}(t)$ in equation (15.7) by

$$\mathbf{b}(t) - s(t)\frac{\mathbf{n}(\gamma(t))}{|\mathbf{n}(\gamma(t))|} \tag{15.10}$$

and solve these new differential equation. We can also think of the function $s(t)$ as allowing for stickiness of a filament.

15.5 Dropping Curves on Surfaces

Drawing curves on surfaces is important in robotics applications that involve generating paths for tools to follow. In fact, in those cases one also wants to generate frames which are tangent to the surface along the path. These would be needed by the robot arm for orientation purposes. Mathematically, the easy part of the problem is finding the point where a ray pierces a surface. This is something we already discussed in Section 13.4.1.

Let $\mathbf{S}$ be a surface parameterized by a function $p(u,v)$. Picking points $\mathbf{p}_i$ on $\mathbf{S}$ with a mouse would also give us a sequence of points $\mathbf{q}_i = (u_i,v_i)$ in the parameter space $\mathbf{X}$ of $\mathbf{S}$, where $p(\mathbf{q}_i) = \mathbf{p}_i$. The goal is to generate the frames $F_i = (\mathbf{u}_{1i},\mathbf{u}_{2i},\mathbf{u}_{3i},\mathbf{p}_i)$. See Figure 15.12. From the points $\mathbf{q}_i$ we can generate a curve

$$\sigma : [0,1] \to \mathbf{X} \subset \mathbf{R}^2.$$

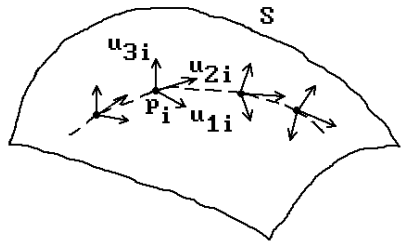


Figure 15.12. Dropping frames on surfaces.

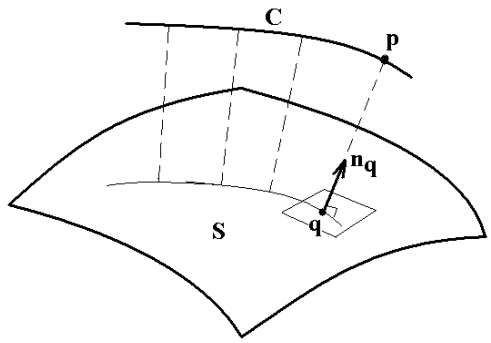


Figure 15.13. Projecting a curve onto a surface.

Then $\gamma : [0,1] \to \mathbf{S}$ defined by $\gamma(t) = p(\sigma(t))$ is a curve on the surface. How should σ be defined from the $\mathbf{q}_i$? If we are not careful then, depending on the parameterization $p(u,v)$, γ may wiggle in undesired ways on the surface. The point is that the user was outlining a curve on the **surface** not in its parameter space. Most likely, the curve should go along a geodesic from one point to the next. Therefore, one way to try to generate the curve from $\mathbf{q}_i$ to $\mathbf{q}_{i+1}$ is to follow the geodesic which starts at $\mathbf{q}_i$ in the "direction from $\mathbf{q}_i$ to $\mathbf{q}_{i+1}$." As an approximation to this direction we can take the vector $\mathbf{q}_i\mathbf{q}_{i+1}$. Now we can apply the methods discussed in Section 15.3.1. Unfortunately, the complete curve may now have corners at the dropped points.

In the case of polygonal surfaces one wants to find a polygonal curve that pass through the dropped points. Connect the dropped points by discrete geodesics.

A related problem is to project a given curve **C** orthogonally to a surface **S**. What this means is that each point **p** on the curve **C** should project to the point **q** on the surface **S** that is closest to it. In the smooth case a necessary condition is that

$$(\mathbf{p} - \mathbf{q}) \times \mathbf{n}_\mathbf{q} = \mathbf{0},$$

where $\mathbf{n}_\mathbf{q}$ is a normal to **S** at **q**. See Figure 15.13. The problem of finding the point on a smooth surface that is closest to a single point was already considered in Section 14.2. In our case here we could pick points $\mathbf{p}_i$ on the curve and find their closest points $\mathbf{q}_i$ on the surface by solving equations like equations (14.4) or (14.5). Connecting the points $\mathbf{q}_i$ by paths on the surface would give us a curve that is an approximation to the projection curve. Alternatively, if the curve **C** and surface **S** are parameterized by functions $\gamma(t)$ and $\varphi(u,v)$, respectively, then the equation above is equivalent to the equations

$$(\gamma(t) - \varphi(u,v)) \bullet \varphi_u(u,v) = 0$$

$$(\gamma(t) - \varphi(u,v)) \bullet \varphi_v(u,v) = 0,$$

which one can try to solve for t, u, and v by some Newton-Raphson method using the points $\mathbf{p}_i$ and $\mathbf{q}_i$ as start points.

Once one has a curve in a surface, frames at points along it would be obtained from the tangent vector to the curve, the normal to the surface, and their cross product.

15.6 Blending

One of the important features of a good modeling system is the ability to blend curves or surfaces. On its simplest level blending simply means to "round" corners and edges or to smoothly connect two or more curves or surfaces to each other. This is an integral part of many manufacturing processes. Much work has and is being done on that subject. Woodwark ([Wood87]), Hoschek and Lasser ([HosL93]), and Vida et al. ([ViMV94]) have a nice overview of the field. We shall only be able to discuss some of the basic elements of it here.

We begin with a comment about terminology. Two terms that are often used in the context of blending are "fillet" and "chamfer." We shall use the "definitions" given in [Wood87]. The size of a blend influences the terminology. Large blends could be considered as a fairing of the surface. On the other hand, fairing a surface along an area where two surface patches meet in an almost tangential way is in general a quite different procedure from the process of blending that we have in mind now. A *fillet* is a mid-sized blend that blends a **concave** vertex, edge, or region in the "internal" part of a nonconvex solid. A good real-life example of this is where a person puts modeling clay along a concave corner and runs his/her thumb along it to create a smooth blend that meets the adjoining surfaces in a tangential manner. A *chamfer* is another mid-sized blend that blends a vertex, edge, or region in the "external" or **convex** part of a solid. One way that the difference between a fillet and a chamfer is expressed sometimes is to say that a fillet is a place where material is added and a chamfer a place where material is subtracted. Finally, it should be pointed out that the terms "blending," "rounding," "smoothing," "filleting," and "chamfering" are at times used interchangeably. In mechanical design and manufacturing blending is usually called filleting or chamfering.

Next, before we get into details, it is useful to try and impose some structure to the broad topic of blending. Varady et al. ([VaVM89] and [VaMV89]) divide blending approaches at the top level into four general types.

Superficial blending: The blending takes place at production time and there are no explicit mathematical formulas associated to the operation but rather there is an instruction of the type "round off with radius R."

Surface blending: One desires a surface that blends two or more other surfaces that are defined either parametrically or implicitly and may or may not intersect.

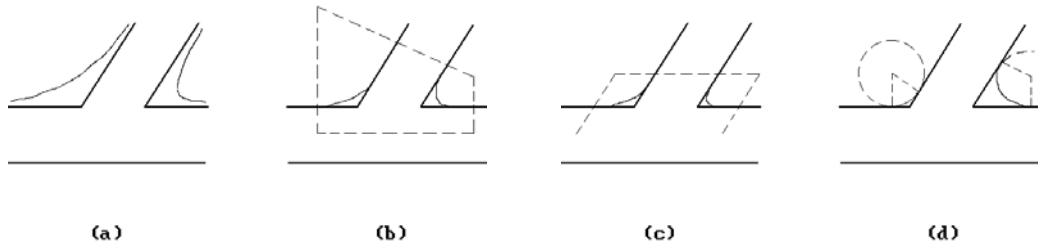


Figure 15.14. Examples of blends.

Polyhedral blending: Here objects are defined by polyhedra and one wants either a polyhedral blending surface or a procedure that, via recursive subdivision, generates a polyhedral blending surface.

Volumetric blending: This approach assumes that we have modeling system based on solids. The system takes care of the blending and provides the user with appropriate operations that carry out the blending automatically. Typically the systems that support this are CSG or b-rep systems where the blending operations are carried out via set operations on the solids. The blending here tends to be of a global nature, whereas the other types of blending are more local, in that they apply only to specific parts of an object.

At the computational level one can make some further distinctions. Is one dealing with implicit or parametric surfaces? Are we using a subdivision algorithm? Are we treating blending as a boundary value problem that is then solved numerically? Additionally, [Wood87] separates blending operations into four types depending on the extent of the blend, that is, how much or in what way the surfaces being blended are modified. Figure 15.14 shows examples of the four types.

(1) There is no constraint and the blend has a global influence on the objects (Figure 15.14(a)).
(2) The blend is constrained to lie in a volume (Figure 15.14(b)).
(3) The blend is constrained to lie in a given range in terms of distances from the edges of surfaces (Figure 15.14(c)).
(4) The blend is constrained by specifying a minimum radius of curvature (Figure 15.14(d)).

We shall look first at several approaches to blending based on implicit surfaces and begin with an example of **global blends**. Blinn ([Blin82]) was interested in displaying molecules and wanted to get away from the "ball-and-stick" approach. He wanted to blend the atoms. His idea was to think of an atom not as a sharply defined ball but rather as a more nebulous object that had a high density near the center of the object but whose density fell off to zero in an exponential fashion. The density function for the atom with center at (x_1,y_1,z_1) was therefore assumed to be of the form

$$D(x,y,z) = e^{-ar},$$

where

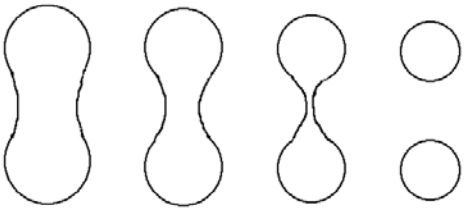


Figure 15.15. Blending with density functions.

$$r = \sqrt{(x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2}.$$

Given a collection of n atoms, we sum the densities for the collection to get a density function of the form

$$D(x,y,z) = \sum_{i=1}^{n} b_i e^{-a_i r_i}, \tag{15.11}$$

where r_i is the distance of (x,y,z) to the center of the ith atom. Actually, for efficiency reasons, r_i^2 was used in the exponents of equation (15.11) rather than simply r_i. One then used a cutoff value c and only displayed those points for which D(x,y,z) > c. See Figure 15.15. By changing the constants b_i and a_i one could achieve different effects. One problem with trying to apply this type of blending in a CAGD program is that the objects would get bigger or smaller because the density function would modify any original sharp boundaries.

Next, [Wood87] attributes an early example of **volume bounded blends** to M.A. Sabin. Given two surfaces defined by F = 0 and G = 0, define

$$H = F(1 - u^2) + Gu^2 + cu^2(1 - u^2), \tag{15.12}$$

where

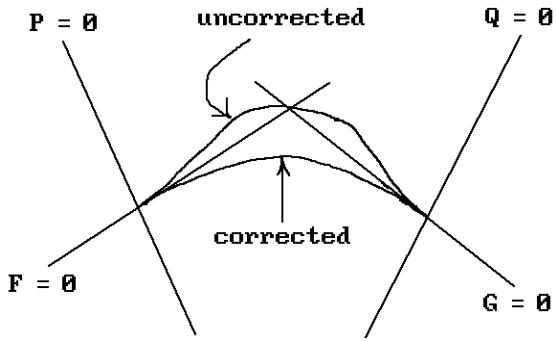
$$u = \frac{P}{P + Q} \tag{15.13}$$

for some auxiliary surfaces defined by P = 0 and Q = 0. The surfaces defined by P and Q define what are usually called the *contact curves*, *link curves*, or *trim (ming) curves* on the surfaces defined by F and G, respectively. These curves define the boundaries of the blending surface. See Figure 15.16. The equation H = 0 then defines the blended surface. The term $cu^2(1 - u^2)$ in equation (15.12) was needed to prevent the blend from passing through the intersection of F = 0 and G = 0 and creating a "bump" at that point.

A related approach originated by Liming ([Limi44]) depends on projective properties of conics. The problem Liming was concerned with was designing airplane fuselages with conic cross-sections. First, note that if we have two conics defined by F = 0 and G = 0, then

$$(1 - t)F + tG = 0 \tag{15.14}$$

Figure 15.16. Volume-bounded blends.



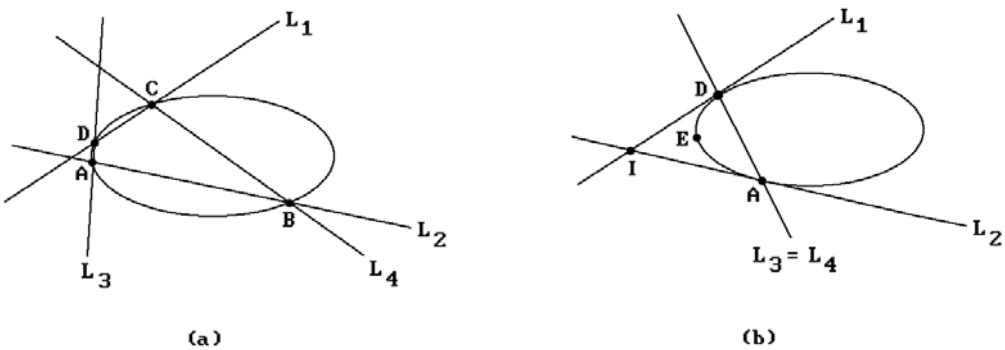


Figure 15.17. Blending with conics.

defines a family of conics which starts with F = 0 and ends with G = 0 as t ranges from 0 to 1. We can apply this to the degenerate conics

$$F = (a_1x + b_1y + c_1)(a_2x + b_2y + c_2)$$

and

$$G = (a_3x + b_3y + c_3)(a_4x + b_4y + c_4),$$

which correspond to two pairs of lines $\mathbf{L}_1$,$\mathbf{L}_2$ and $\mathbf{L}_3$,$\mathbf{L}_4$, respectively. Then the family of conics defined by equation (15.14) passes through the four intersection points **A**, **B**, **C**, and **D** of the lines. See Figure 15.17(a). If we let **C** approach **D** and **B** approach **A**, that is, we let the lines $\mathbf{L}_3$ and $\mathbf{L}_4$ move toward each other, then we shall find that in the limit when $\mathbf{L}_3 = \mathbf{L}_4$, the lines $\mathbf{L}_1$ and $\mathbf{L}_2$ will be tangent to the conics defined by (15.14) at **C** = **D** and **B** = **A**, respectively. See Figure 15.17(b). The conic is uniquely specified by the two points of tangency **A** and **D**, the intersection point **I**, and one other point **F**, which also determines t. This construction obviously gives us blends between two lines in the plane. Furthermore, it extends to 3-space and can be used to find cylindrical and conical blends between two planes.

An example of a **range constrained blend** is the superelliptic blend described in [RocO87], where the cross-sections of the blending surface are superellipses. Another, which we shall describe briefly, is the so-called **potential method** of Hoffmann and Hopcroft in [HofH87]. If F, G, . . . : $\mathbf{R}^3 \to \mathbf{R}$, let V(F,G, . . .) denote the set of simultaneous zeros of the functions F, G, . . . , that is,

$$V(F,G,\ldots) = \{\mathbf{p} \mid 0 = F(\mathbf{p}) = G(\mathbf{p}) = \ldots\}.$$

Consider a set V(F). This set partitions space into three sets: The points **p** where $F(\mathbf{p}) > 0$, which we shall call the *outside* of V(F), the points **p** where $F(\mathbf{p}) < 0$, which we shall call the *inside* of V(F), and the set V(F) itself. For any nonzero value s, the set V(F – s) is either inside or outside of V(F), depending on the sign of s. For example, if

$$F(x,y,z) = x^2 + y^2 + z^2 - 1,$$

then V(F – 3) is the sphere of radius 2 that lies entirely outside the unit sphere V(F).

Now

$$V(G,H) = V(G) \cap V(H)$$

and in general such a set corresponds to the intersection of two surfaces and represents a curve in space. Consider V(G – s,H – t). This describes a family of space curves parameterized by s and t. Next, suppose we constrain the parameters s and t to lie on a curve **C** in the plane defined by an equation

$$f(s,t) = 0.$$

Define F by

$$F(x,y,z) = f(G(x,y,z),H(x,y,z)). \tag{15.15}$$

Then

$$V(F) = \bigcup_{f(s,t)=0} V(G-s, H-t) \tag{15.16}$$

is a surface. Figure 15.18 shows how V(F) is the union of points **p** that are the intersections V(G – s,H – t) of offsets of V(G) and V(H). If f is chosen appropriately, then V(F) becomes our blending surface between G and H.

15.6.1 Theorem. If **C** is tangent to the s-axis at (a,0), then V(F) is tangent to V(H) along the curve V(G – a,H). Similarly, if **C** is tangent to the t-axis at (0,b), then V(F) will be tangent to V(G) along V(G,H – b).

Proof. See [HofH85].

Since we clearly want a blending surface to be tangent to the surfaces between which it is a blend, the important consequence of Theorem 15.6.1 is that we have

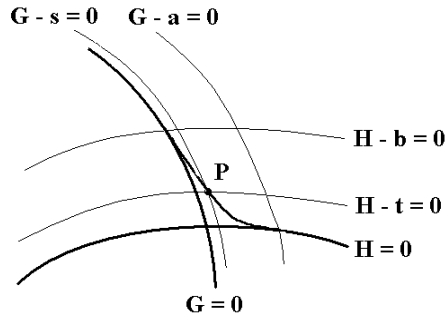


Figure 15.18. Blending with intersections of offsets.

Table 15.6.1

	Type of conic
$\lambda = -\infty$	A pair of lines: s = 0 and t = 0
$-\infty < \lambda < -ab$	Hyperbola
$\lambda = -ab$	Parabola
$-ab < \lambda < ab$	Ellipse: a circle if a = b and $\lambda = 0$
$\lambda = ab$	The line bs + at − ab = 0, counted double

reduced the blending problem for surfaces to the problem of finding a blending function f(s,t) for the coordinate axes in parameter space that is much easier. Furthermore, if the functions we are dealing with are polynomials, then we would like f to have as low a degree as possible. The natural candidate for f is therefore a conic. In view of the tangency conditions, the general form for f is

$$f(s,t) = b^2s^2 + a^2t^2 + a^2b^2 - 2ab^2s - 2a^2bt + 2\lambda st, \tag{15.17}$$

where a, b, and λ are parameters we are free to choose. The parameters a and b specify the points of tangency and λ determines the type of conic we are using (see Table 15.6.1). With this choice of f, the blending surface F will have degree 2(m + n) if G and H have degree m and n, respectively. If G and H are quadratics, then F will be a quartic.

15.6.2 Example. Consider the cylinder V(G) and plane V(H), where

$$G(x,y,z) = x^2 + y^2 - 25 \quad \text{and} \quad H(x,y,z) = y - 2.$$

We would like a blending surface that meets V(G) and V(H) in the vertical lines through (4,3) and (6,2), respectively.

Solution. See Figure 15.19(a). If we set a = 11, b = 1, and $\lambda = -11$ in equation (15.17), then

$$f(s,t) = s^2 + 121t^2 + 121 - 22s - 242t - 22st.$$

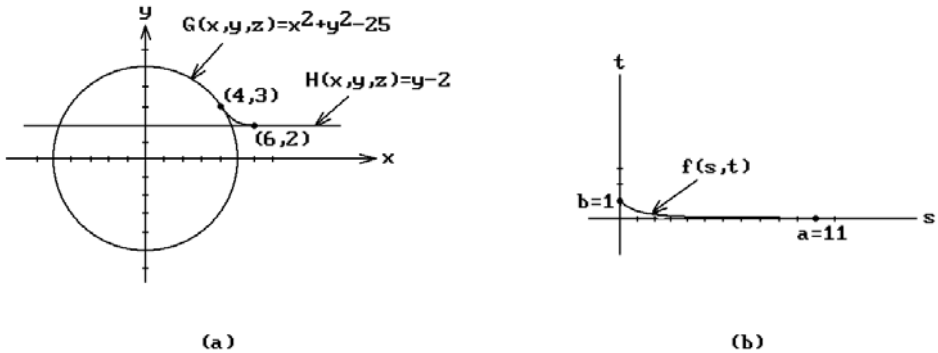


Figure 15.19. Blending with potential surfaces.

The curve f(s,t) = 0 defines a parabola (see Figure 15.19(b)) and satisfies the hypotheses in Theorem 15.6.1, so that V(F) will be the surface we want, where

$$F(x,y,z) = x^4 + 2x^2y^2 - 22x^2y + y^4 - 22y^3 - 28x^2 + 93y^2 + 176y + 1164$$

is defined by equation (15.15). See Figure 15.19(a), which shows the horizontal slice of our surfaces in the xy-plane.

Our construction created a blending surface by a sweeping operation of curves parameterized by a curve f in a plane. We can think of this as a three-dimensional construction by thinking of f as defining a conic cylinder in x-y-z space.

Another well-known implicit blending approach is the **rolling ball–type blend**. As the name suggests, one rolls a ball along the two surfaces one is trying to blend. The ball will touch the surfaces in a tangential way. Mathematically, the surface generated by the rolling ball is a canal surface. One problem is that the blending surface that is obtained in this way is defined by complicated equations even for a blend between relatively simple surfaces such as cylinders. For that reason one has sometimes used approximations ([RosR84]). Klass and Kuhn ([KlaK92]) describe a unified approach to finding a fillet surface based on a rolling ball approach. After determining the intersection curves that are needed for trimming the original surfaces, they end up defining a bicubic Bézier blending surface.

More complicated yet are **variable radius rolling ball blends**. This leads us to cyclides. The definition and geometry of these surfaces are discussed elsewhere in this book (see Section 12.13 in this book and Section 9.13 in [AgoM05]). The reason for the renewed interest in them is precisely because of their usefulness for blending. Figure 15.20 shows how a cyclide can be a blending surface between a cylinder and a plane. More generally, cyclides work well for blending between the basic quadrics, namely, planes, spheres, cylinders, and cones. Allen and Dutta ([AllD97a] and [AllD97b]) define the problem carefully and show that cyclides can achieve singularity-free variable radius rolling ball blends. They give necessary and sufficient conditions for the existence of certain blends and describe constructions for them. They also indicate limitations with single cyclide blends. The paper [AllD97c] extends their results to supercyclides. The supercyclides allow more freedom in the shape of blends

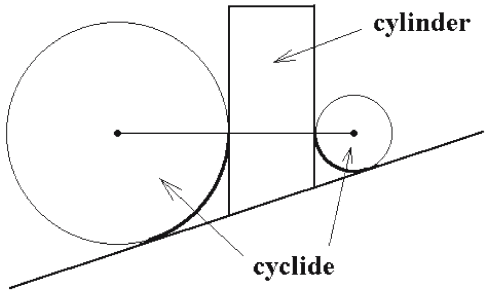


Figure 15.20. Cross-section of cylinder/plane blending with a cyclide.

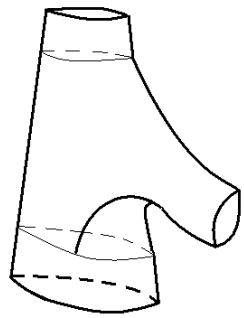


Figure 15.21. The cone/torus blending problem.

but are also more complex computationally. Shene ([Shen98]) gives a complete analysis of possible blendings of two cones with a cyclide. The paper [Shen00] discusses a solution to problems encountered with a common construction for blending two cones with a cyclide. A well-known unsolved blending problem is the so-called Cranfield object that involves a cone/torus blend. See Figure 15.21. The actual Cranfield object is a real part of an oil rig. Although it is possible to model the part in commercial systems, it is not easy. As Pratt ([Prat90]) and others have pointed out, modeling systems should have built in capabilities that would allow a user simply to specify the cone and torus and let the system do a robust blending operation on its own. Other references for cyclides can be found in the bibliography.

Next, we look at blending based on parametric surfaces. A good reference is [ViMV94]. It will be worthwhile to start with some terminology as summarized by Vida et al.

Base surfaces: The surfaces that are being blended. See Figure 15.22(a).

Blending surface: The surface that does the blending. See Figure 15.22(a).

Trimline: A curve along which the base and blending surfaces meet. Such a curve can be considered as a curve that trims the base or blending surface. See Figure 15.22(a).

Range parameter: Parameters that specify the extent of the blend such as how far from the intersection of the base surfaces the trim line should be. They could be computed automatically or specified by the user. See Figure 15.22(b).

Spine curve: In those cases where a blending surface is defined by a sweeping-type operation, a spine curve could be the trajec-

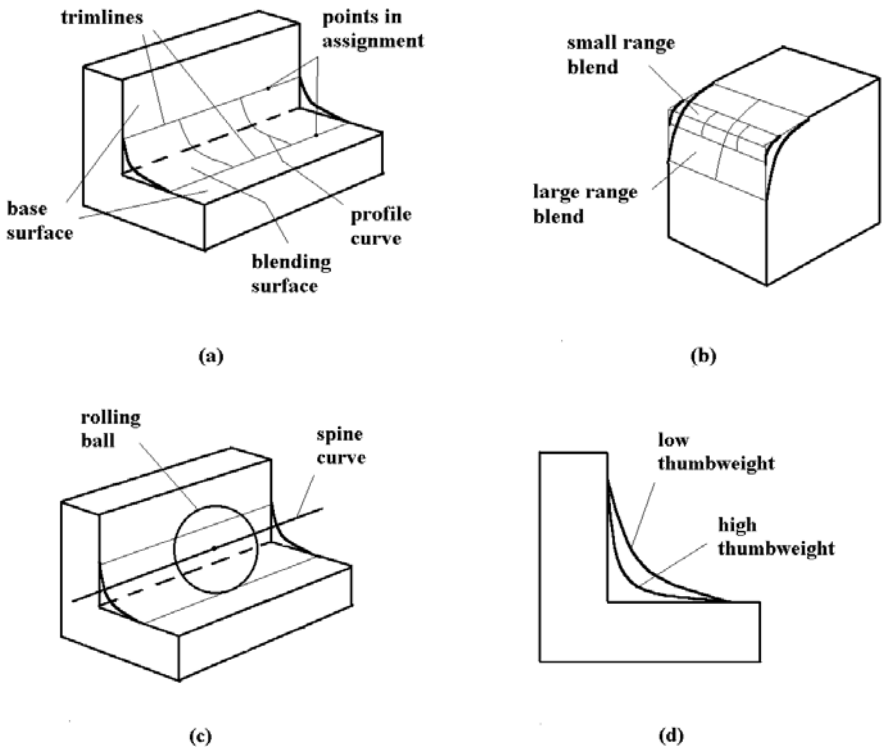


Figure 15.22. Parametric blending terminology.

tory of the sweep, but it is more generally other related curves such as the intersection of the base surfaces or some offsets of those. See Figure 15.22(c).

Profile curve: A cross-sectional curve of the blending surface associated to each point of its spine. It is usually a planar curve. Other terms that have been used for this curve are "blending arc," "generator," or "crossing curve." See Figure 15.22(a).

Thumbweight: This is a quantity associated to the shape of a profile curve and measures its "fullness." The closer the curve is to the base surfaces, the higher one says that its thumbweight is. See Figure 15.22(d).

Assignment: One can define profile curves without spine curves, but then one has to say which points correspond to which on the trim lines. This correspondence is called assignment. See Figure 15.22(a).

Vida et al. divide parametric blending approaches into five categories:

(1) Rolling ball blends
(2) Spine-based blends
(3) Trimline–based blends

(4) Polyhedral methods
(5) Other methods

Rolling ball blends have already mentioned. There is a natural spine curve, namely, the curve swept out by the center of the ball, and the trimlines are just where the ball meets the base surfaces. See the paper by Vida et al. for a discussion and references for how the blending surface can be parameterized. For example, Choi and Ju ([ChoJ89]) use quadratic rational Bézier curves for the cross-sections of the parameterizations. Another approach is described in the paper [KlaK92] mentioned earlier.

In spine-based methods, the spine has to be defined. In the case of rounding edges, the edge can be used as the spine. More generally, it could be the intersection of two surfaces, such as offset surfaces from the base surfaces. If sweeping is involved, then the sweep lines can be so used. Rolling ball blends can be considered as a special case of spine-based methods. The modeler described by Chiyokura in [Chiy88] basically does spline-based blending. Trimlines are usually computed automatically. For example, the trimlines could be defined in terms of the points on the base surfaces closest to the spine curve. The profile curves are usually required to be curves that lie in the planes determined by the points on the spine curves and their two assigned points on the trimlines.

For trimline–based methods there are many ways that those curves could be specified. One can define trimlines as the intersection of a base surface and another surface, such as an offset surface for the other base surface. One can also define trimlines by specifying curves in the parameter spaces of the base surfaces if they are parametric surfaces. Once one has the trimlines, the next step is to define profile curves. This can be done with or without an explicit spine curve. Quite a few approaches to blending are trimline based. Again see [ViMV94] for more details. An approach using trimmed tensor product surfaces is described in [ElbC97].

Most parametric blending tends to use rectangular or triangular patches. The reader interested in blends for n-sided patches should see [HsuT98], where one can find a number of additional references for this topic.

Blending methods based on polyhedral objects fall into two types. In both cases one starts with an initial polyhedral object that defines the general shape of the final object. One can then either use recursive subdivision of the facets or a local rounding operation to achieve smoothing. Recursive subdivision surfaces were described in Section 12.17. Chiyokura and Kimura ([ChiK83]) suggested using local rounding operations that are built into a modeler. Their idea was that one would first define a wireframe object by means of Euler operations whose edges would be tagged as needing rounding or not. The initially straight edges were then replaced by appropriate curved edges. The last step was to fill in faces with smooth surface patches. The approach was extended in [Chiy87] and [Chiy88]. A user would first specify fancier appropriate rounding data for edges and vertices of the original polyhedral model. (Such data essentially corresponds to defining trimlines.) This would define a curved mesh and Gregory patches would then be generated for the actual rounding. Another direct approach is described in [Szil91]. Szilvasi-Nagy uses automatically generated rectangular Ferguson-type bicubic patches for the blending. Cylindrical surfaces are used to round edges and, at the vertices where they meet, a rectangular patch blends them. A user can vary the shape of the blend by modifying parameters such as rounding radii.

The two last approaches to blending that we shall mention are quite different from all the others. The first is due to Roach and Martin ([RoaM92]), who use Fourier transform methods. Roughly speaking, one starts with a rectangular array of points chosen from the relevant region in the initial surface. The Fourier transform is then applied to these points, the high-frequency part of this data is attenuated, and then mapped back by means of the inverse Fourier transform. The motivation for this is that high frequencies correspond to sharp changes in shape, such as, for example, at vertices and edges, and the mathematical steps taken are a way to smooth these changes. The other blending approach is due to Bloor and Wilson ([BloW89a] and [BloW89b]), who argue that a blending surface can be thought of as a solution to the following type of problem:

> Given a region Ω with boundary $\partial\Omega$, is there a function defined on that region that satisfies some given conditions on $\partial\Omega$?

This is a well-known type of problem in analysis and leads naturally to Laplace-type partial differential equations satisfying certain boundary conditions. We sketch the example in [BloW89b]. That example involves blending a vertical cylinder of radius 1 centered on the z-axis and the horizontal xy-plane z = 0. In our current context, a parameterization

$$p(u,v) = (x(u,v), y(u,v), z(u,v))$$

for the blending surface, should therefore be interpreted as a solution to the partial differential equations

$$\frac{\partial^2 x}{\partial u^2} + b^2 \frac{\partial^2 x}{\partial v^2} = 0, \quad \frac{\partial^2 y}{\partial u^2} + b^2 \frac{\partial^2 y}{\partial v^2} = 0, \quad \text{and} \quad \left(\frac{\partial^2}{\partial u^2} + a^2 \frac{\partial^2}{\partial v^2}\right)^2 z = 0, \tag{15.18}$$

where a and b are constants that have been introduced to allow for more control over the solution. The trimlines were taken to be the circle

$$z = h, \quad x^2 + y^2 = 1$$

at height h on the cylinder and the circle

$$z = 0, \quad x^2 + y^2 = R^2.$$

in the plane. Given the boundary conditions

$$\begin{gathered} x(0,v) = \cos v, \quad y(0,v) = \sin v, \quad z(0,v) = h, \\ x(1,v) = R\cos v, \quad y(1,v) = R\sin v, \quad z(1,v) = 0, \\ x_u(0,v) = 0, \quad y_u(0,v) = 0, \quad z_u(0,v) = s, \\ z_u(1,v) = 0, \quad x_u(1,v) \text{ and/or } y_u(1,v) \neq 0, \end{gathered} \tag{15.19}$$

it turns out that one can write down an explicit solution to equations (15.18) that depends on the two parameters a and s. (The parameter b ends up being fixed for this

Figure 15.23. Blending as a boundary value problem.

solution.) Varying these parameters generates different blends and their effect can be described geometrically. Figure 15.23 shows cross-sections of the solutions corresponding to s = –0.1 and –10.0 for a fixed parameter a.

Finally, as is the case for most solutions to geometric modeling problems, after one has a basic idea for a solution one usually has to deal with a number of "details." In the case of blending, aside from the usual issues of robustness and efficiency, there are some topological issues. If one is blending an edge, what does one do at the ends of the edge? A vertex by itself can be dealt with by defining a trimline and filling in a patch. Ends of edges are more complicated. There may be several edges ending in the same vertex. See [ViMV94] for this nontrivial problem and ways to deal with it.

15.7 Programming Projects

Section 15.3.1

15.3.1.1 Let a user pick two points on a surface. Use the second point to define a direction and then generate a smooth geodesic starting at the first point and going in that direction.

Section 15.5

15.5.1 Have the user pick a surface and then define a space curve whose data was read from a file. Drop this curve onto the surface.

Section 15.6

15.6.1 Implement one of the blending techniques for curves.

CHAPTER 16

Intrinsic Geometric Modeling

Prerequisites: Chapters 7–9 in [AgoM05] (for Sections 16.3 and 16.4)

16.1 Introduction

The modeling that we have discussed so far in this book dealt with objects imbedded in $\mathbf{R}^3$ (or $\mathbf{R}^n$ more generally). Euclidean space was ever present, even if implicitly, when we talked about spaces whose points were tuples of real numbers. The curves and surfaces that we discussed were always presented as subspaces of Euclidean space. This is of course not surprising because most people when they talk about geometric objects, even when they talk about them in abstract terms, usually do so in the context of a particular imbedding of them in $\mathbf{R}^3$ rather than thinking of them intrinsically. However, computers have become powerful enough to start doing more of the latter. The motivation for this is not just theoretical. For example, to understand our universe we need to try to represent space in a global way. Consider the analogy with a hypothetical two-dimensional world that is actually a sphere or a torus. Even though the view locally would make it look like $\mathbf{R}^2$, to understand it in its entirety we need to think of it as a **collection** of "flat" patches. This world exists on its own without any "outside" that a three-dimensional person could give it using some imbedding of it in $\mathbf{R}^3$. When we come to our own three-dimensional universe, is it like $\mathbf{R}^3$ or like the three-dimensional unit sphere in $\mathbf{R}^4$? We do not hypothesize any "outside" surrounding space in which our universe is imbedded when we think about the world around us.

In this chapter we want to suggest that we have come to a point in time where there is a place for modeling programs that representing spaces intrinsically and do graphics **inside** a space (manifold) rather than pretend that everything we want to look at and understand lies in a single view. Although the idea of doing this is not entirely new, mathematicians have developed a few programs of this type for research purposes, such a goal has not received much attention in the broader computer graphics community up to now. Virtual reality programs only partially accomplish what we have in mind. Also, note that although we have used the word "global" in previous chapters, as in the discussion of local and global illumination models in Chapter 9 and in the title of Chapter 14, we are taking the meaning of "global" a step further

here. In particular, the global illumination models discussed in the earlier chapter still were implemented in a local space.

Finally, this brings us to another point. Current modeling systems (other than those involved with volume rendering) only represent objects. Systems should really also model the **entire** space ($\mathbf{R}^3$) that these objects exist in. Since the objects are really imbedded objects, if one modeled the complimentary space one would have a lot more information. For example, visible surface determination would theoretically become less expensive because one could basically "walk" from the viewpoint to an object to see if something blocked it rather than having to check a ray against all the objects in a scene. The fact that one can do precisely that in volume rendering makes that approach very attractive and guarantees that it will become more important in the future, especially since the hardware issues that have made this approach impractical are going away.

Section 16.2 is a slight digression from the main topic of this chapter, although the material is related. We discuss current virtual reality programs that try to immerse a user into computer generated worlds which are representations of either actual or totally artificial environments. Section 16.3 describes what geometrically intelligent modeling systems might look like. Section 16.4 describes the idea behind the SPACE program that accompanies this book. Section 16.5 touches on some software that currently exists and which implements some aspects of what we have been talking about here, suggesting future directions for geometric modeling.

16.2 Virtual Reality

The term "virtual reality" (VR) is used to mean many things, ranging from visions that one may never be able to achieve to more down-to-earth applications that are starting to have a profound influence. If one wants to create a virtual, meaning computer-generated, world, one clearly has to be able to model this world. It is therefore appropriate to say something about it in a book on geometric modeling; however, there is much more to VR than just modeling and so we shall limit ourselves to some general comments. The interested reader is referred to [Broo99] and [VFLL00], where one can find lots of other references to work in the field. Those papers and others in the *IEEE Computer Graphics and Applications* journal also describe many examples of applications in VR. We shall not describe them here.

The idea of virtual environments (VE) using head-mounted displays (HMD) goes back to the early 1960s ([HalM63]). Ivan Sutherland ([Suth65]) considered the computer screen as a window into a virtual world. However, actually getting anything to work took some time because it was harder than initially expected. Before 1990 the results were very limited and existed mainly at a few laboratories. The first two IEEE conferences on VR took place in 1993. One was the Virtual Reality Annual International Symposium (VRAIS '93) and the other was a symposium at the Visualization '93 conference. As Brooks ([Broo99]) points out, between 1994 and 1999 the technology moved from "it almost works" to "it barely works." Vehicle simulators and entertainment applications were not counted in that analysis. Although very successful, the former were not really built on VR technology and the latter made lower demands and the experience achieved was an end in itself and not thought of as a tool.

Sometimes it is convenient to distinguish between two flavors of VR, namely, passive (nonimmersive) versus immersive virtual reality (IVR). An example of the former would be running a flight simulator on a computer screen. In the case of IVR, a user would feel surrounded by the computer-generated environment and would be able to walk through it and interact with it. There are two standard implementations of IVR. One uses a head-mounted display and the other puts the user in a "cave." The original cave environment was the CAVE system described in [CrSD93]. IVR can be further distinguished by how much freedom a user has and what the constraints are. What mobility does a user have? Can one "walk" around the environment? What is the field of view?

LaViola ([VFLL00]) divides three-dimensional user interaction in a VE into three categories:

Navigation. This includes physical movement (e.g., actual walking, walking in place on a treadmill, riding a stationary vehicle like a bicycle), manual viewpoint manipulation (e.g., hand motions determine movement), steering (e.g., gaze directed motion), target-based travel (user specified destination), and route planning (e.g., the user specifies a path to follow by drawing on a map).

Selection and Manipulation. A basic approach here would be a virtual hand or a cursor that tracks one's hand. One can also implement indirect control via widgets (e.g., handle widgets that allow rotation, translation, . . .). A third way is via physical props.

Application Control. This can be achieved, for example, with graphical menus, voice commands, gestural interaction, or tools.

Brooks ([Broo99]) argues that four technologies are needed for VR to be successful:

(1) To get multiple sensory information, one needs visual, auditory, haptic, and tactile displays to immerse a user in the virtual world while at the same time blocking out any contradictory sensory impressions from real world.
(2) Graphics rendering systems need to be able to sustain 20–30 frames per second motions.
(3) Tracking systems need to be capable of continually reporting a user's position and orientation.
(4) One has to be able to create and maintain large databases of models in the virtual world.

Resolution will have to increase before VR looks real. A user would want to get the same sensations from the virtual environment as from a real one. In particular, there should be force feedback. A big early problem was latency. In 1994 it tended to be 250–500 ms, which was much too large because flight simulators have shown that a latency of more than 50 ms is perceptible. The latency problem is especially noticeable for head rotations. Motion capture can be achieved via wireless optical and magnetic tracking systems with or without wires or by using an exoskeleton. Virtual humans are very difficult because their motions a very complex. Models for a virtual environment are not easy to come by. They usually come either from a CAD systems

or sensing devices. In general, what is the best way to populate virtual environments? A recent additional level of complexity arises in the context of distributed and multiuser environments. See [CMBZ00] for some comments on networked virtual environments.

An alternative to VR is augmented reality (AR). This is a hybrid approach where one displays three-dimensional virtual objects in a user's natural environment. Typically, a user would see the world via a head-mounted camera and a display with virtual images superimposed onto the real scene. Some applications of AR are showing proposed buildings in their actual setting, interacting with machinery, and medical data for patient. For example, AR has been used to superimpose internal ultrasound data onto a patient.

There are of course many connections between VR and geometric modeling. One area where VR has stimulated research in geometric modeling has to do with the fact that one often deals with large models that have to be displayed quickly in interactive environments. Decimation algorithms come in handy here, that is, one wants to remove data that is not really needed for accurate display. See the end of Section 14.3 for a few more details and some references.

It seems that currently, in addition to the standard architectural walkthroughs, virtual prototyping, and medical, therapy, and entertainment applications, IVR is having its biggest impact on scientific visualization where the huge amounts of data one has to cope with makes it hard to grasp its meaning. The current successful applications are rather specialized however and one is left with the question of when IVR will be used in the more general purpose sort of way that its strongest advocates dream about.

16.3 Geometrically Intelligent Modeling Systems

Modeling systems have come a long way. A variety of paradigms have been developed. They were discussed in Chapter 5, but it is clear that overall what we have is a lot of ad hoc approaches, some of which work reasonably well within their domain. A unifying structure is missing in CAD/CAM/CAGD. Of course, the ad hoc nature of current state of geometric modeling is not surprising, given the short time that computers have existed. Furthermore, it is important to have something that "will get the work done" now. The niceties of having a unifying "theory" are things that can be indulged in at some future point in time. Besides, there will always be special problems that are solved in special ways. Nevertheless, it is the author's dream to see a truly general and complete modeling system that will understand all the geometry that mathematics understands at the time.

There are really two basic parts to CAD/CAM/CAGD systems: the underlying geometry and the interface between it and the user. Since the geometric coverage is adequate for current users and industry, if one overlooks the constant efforts to make algorithms and data structures more efficient, it is not surprising that most of the new work is related in one way or another with the user interface. The book [ShaM95] explains how, by looking at the design process, one can create feature libraries and automate some common model manipulations. The authors refer to features as being based on an **extensional view** of design, where one records what practitioners commonly do. They argue that one also needs to capture the **intentional aspect** of design,

where one tries to establish in advance what a user may want to do. Much more work needs to be done in this area. The then-current state of feature-based design and key problems are discussed by Mäntylä et al. in [MäNS96]. Developing expert systems in the geometric context is much harder than for the traditional areas where expert systems have been successful. Ways of using a blend of artificial intelligence and computational geometry algorithms to help here are described by Requicha in [Requ96]).

Amato's ([Amat96]) view of geometric modeling was that while a lot has been done with respect to **building** models, much more work remains to be done when it comes to **analyzing** and **manipulating** models. She specifically addresses the problem of virtual prototyping in industrial design. There are many more stages to product development than just modeling and one would like to automate them also. Two of the problems she mentions are:

Design for Maintenance. After individual parts of a larger object have been designed and assembled, one would like an understanding of how they fit "together." Can a part be removed without removing other parts? Is there room enough to insert a tool and for a mechanic to do this? Amato points out that the Boeing 777 was completely designed in a CAD/CAM system, but the maintenance issues were studied with physical models. A problem such as part removal is really a special case of the difficult motion planning problem.

Assembly Sequencing. If an object is made up of parts, what sequence of steps would allow one to put the whole object together? This is related to the part removal problem.

By modeling the entire prototyping process, one can create manuals and also virtual environments for training. Computational geometry is a major component in a solution because many of the problems that have to be solved belong to that field.

From now on we shall use the term "geometrically intelligent modeling system" to mean a system that understand geometric invariants associated to spaces and can answer questions such as whether or not two spaces are homeomorphic or isometric. We are talking about systems that are knowledgeable about the topology **and** differential geometry of an object. Current commercial systems (basically those outside university research departments) only have a superficial knowledge of the geometry of objects. Most of what they know is **local** information.

For a modeling system to be geometrically intelligent, it will minimally have to maintain whatever is necessary to compute the invariants described in Chapters 6–9 in [AgoM04]. These invariants enable us to distinguish between surfaces. Although they are insufficient for classifying spaces in higher dimensions, they are at least a start towards that goal. The following two steps are necessary to carry out this program:

(1) We need to maintain an adequate cell structure for all of its objects, one suitable for computing at least the standard algebraic topology invariants such as the homology groups.
(2) We need to maintain the appropriate differentiable structure to compute differentiable geometry invariants. In the discrete case one only needs the attaching maps of the cells.

The ability to classify spaces is especially important for computer vision and robotics. A lot of work has been done over the years to determine the parts of a digitized scene, the goal being to understand what is really in the scene in the same sense that humans would. For example, one would like to be able to recognize whether we are looking at a sphere or a doughnut (torus). Much still needs to done to achieve such a higher level of recognition.

As a concrete example of what the author has in mind when it comes to geometrically intelligent programs, the next section describes the author's attempt at such a program. It is hardly a definitive answer to the discussion above but should be taken simply as an indication of how manifolds can be treated intrinsically.

16.4 Exploring Manifolds

This section describes some features and capabilities of the SPACE program that the author is currently implementing. It is a work in progress and not everything has yet been implemented. The reader will have to consult the documentation on the accompanying CD to find out the latest state of the program. Nevertheless, we want to indicate the author's approach to the development of geometrically intelligent modeling programs. The idea of the program is to enable users to define and edit arbitrary three-dimensional manifolds (with or without boundary), to "fly" through them, and to query them about topological and geometric properties. The user interface is described in more detail in the document SpaceGUI on the CD. The code for the program and additional documentation can also be found on the CD. There is a "$2^1/_2$"-dimensional mode in the program for studying surfaces. If $\mathbf{S}$ is a surface, we use the three-dimensional space $\mathbf{S} \times [0,1]$ to study $\mathbf{S}$. The surface $\mathbf{S}$ is thought of as the "floor" $\mathbf{S} \times 0$ of the "room" $\mathbf{S} \times [0,1]$ along which we can walk. In this way the study of surfaces can be made into a special case of the study of three-dimensional manifolds, which does not involve much extra work.

Internally, the SPACE program represents a manifold by means of a cell decomposition. It could easily be generalized to represent arbitrary spaces that admit cell decompositions because the only essential property of the cell decomposition is that a space is built by a sequence of steps, each of which involves attaching a cell to the previous part via some attaching map. We also deal only with polygonal manifolds (manifolds obtained by gluing rectilinear cells together via linear maps), so that questions of differential geometry are moot, but one could handle discrete curvature or other discrete geometry questions. Manifolds $\mathbf{M}$ are represented as the union of a sequence of 3-dimensional faceted disks $\mathbf{D}_1, \mathbf{D}_2, \ldots,$ and $\mathbf{D}_k$, so that if

$$\mathbf{M}_i = \bigcup_{j=1}^{i} \mathbf{D}_j,$$

then $\mathbf{M} = \mathbf{M}_k$ and each $\mathbf{D}_i$ is attached to the boundary of $\mathbf{M}_{i-1}$ by identifying a collection of facets in the boundary of $\mathbf{D}_i$ with facets in the boundary of $\mathbf{M}_{i-1}$ via a map f_i.

Manifold Creation. Users can create new manifolds in a variety of ways ranging from low-level to high-level operations such as

(1) by choosing from a collection of well-known predefined manifolds that topologically are homeomorphic to spaces such as $\mathbf{D}^3$, $\mathbf{S}^3$, or $\mathbf{P}^3$,
(2) by gluing two parts of the boundary of an already constructed manifold together,
(3) by attaching one manifold to another along regions in their boundary,
(4) by a connected sum operation on two existing manifolds,
(5) via a spherical modification (surgery) operation on an existing manifold or by adding a handle to the boundary, and
(6) as a quotient of $\mathbf{D}^3$ by the action of certain transformation groups.

Visualization. One can visualize a manifold **M** using one of two views.

The "global" view: This is the intrinsic view where the observer is moving through **M**.

The "local" view: This shows a perspective view of the individual cells in $\mathbf{R}^3$ that were used in the construction of **M**. There is an associated orthographic view, which is simply the orthographic projection of the local view onto the x-y plane.

One can think of these views as corresponding to two worlds – a global and local world. Looking at this another way, there is an abstract cell structure which is the same in the global and local world but their point data is different (the two worlds correspond to different realizations of the abstract cells). Basically, the observer is inside the manifold in the global world and outside looking at its parts in the local world. Users can move the cells in the local view without changing the topological type of the manifold. The local view has a mode where the identification of boundary polygons is indicated. Both the global and perspective local view can be seen simultaneously. The observer is indicated by means of a cone in the perspective and orthographic local view. The observer can be moved in various ways and all the representations of the observer in the different views move in parallel.

Initially, when a manifold is defined, the boundaries of the cells that make it up are shown when one is in the global view. It is convenient to think of the boundary polygons as "walls." At this point the user can start collapsing the cell structure, either by piercing individual walls or by letting the program automatically collapse all the walls as far as possible to a "reduced" state. The observer would then see only walls that are left and that cannot be collapsed any further. One can make these also invisible, but in that case, assuming that the manifold did not have any boundary, there would be nothing to see (actually the user can specify how far one can look in terms of the number of cells that one can look through). To make things more interesting one can define (and move) objects in the cells of the manifold, so that the topology of the manifold will then influence the way and the number of times that one sees them as one looks straight ahead. One can also make marks on the walls.

Figures 16.1–16.3 (the color version of these figures are .GIF files that can be found on the accompanying CD) are examples of what one can see in the SPACE program. Figure 16.1 shows the ($2^1/_2$)-dimensional (global) view from inside the surface $\mathbf{S}^2$. In fact, we see how the entire screen on the computer monitor would look. The top of the screen shows some status information. On the right are menu items. Immediately

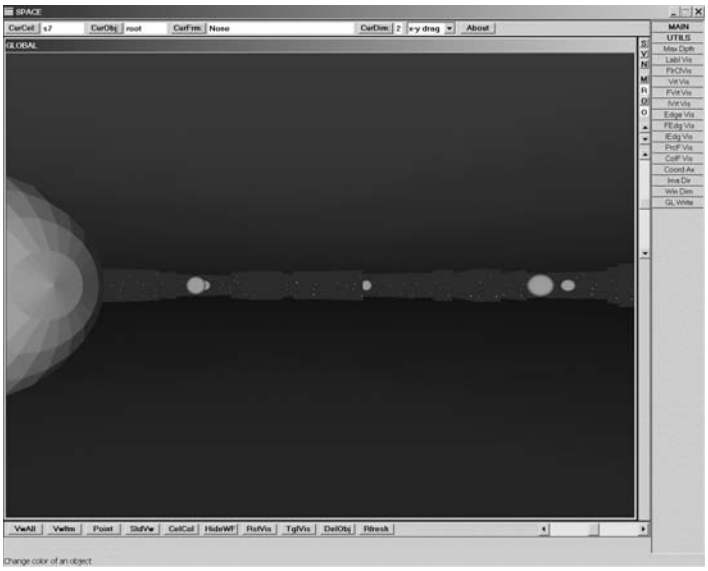


Figure 16.1. A $2^1/_2$-dimensional view from inside $\mathbf{S}^2$.

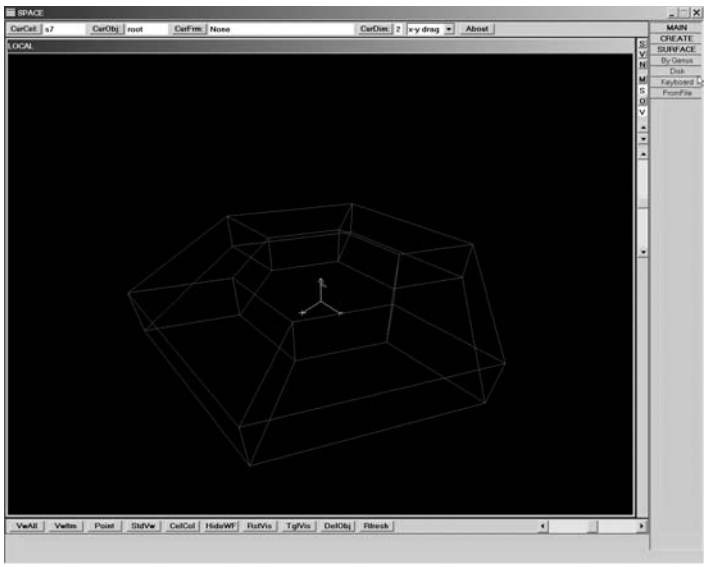


Figure 16.2. A local view of the cells making up the view in Figure 16.1.

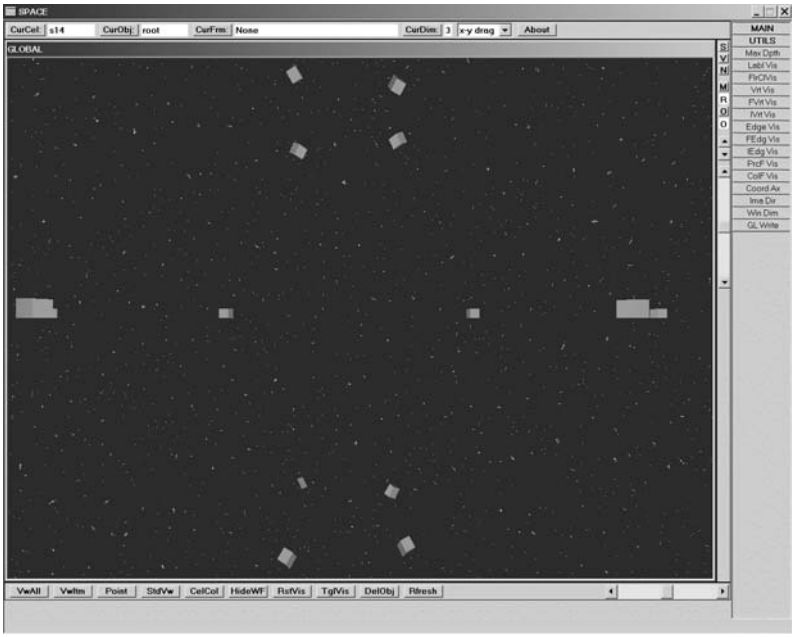


Figure 16.3. A view from inside $\mathbf{S}^3$.

to the right and below the graphics area are various buttons that perform view manipulation actions, such as zooming, spinning, changing the rendering mode, etc. As mentioned earlier, we are really showing $\mathbf{S}^2 \times [0,1]$. The floor $\mathbf{S}^2 \times 0$ and ceiling $\mathbf{S}^2 \times 1$ are shown in brown and blue, respectively. The (abstract) cell structure of the sphere has been collapsed to consist of one 2-cell and one 0-cell. There is an option to show the 0-cell, which would then show up as a column. As a 3-dimensional manifold the world consists of a collection of 3-cells with identifications along their boundary and to display a view we simply look through these cells. We have allowed ourselves to look straight ahead through six cells. The "walls" in the distance are boundary walls of the last cell we see through. They are colored to look like a sky with stars and this is intended to indicate that we could look further. There is also a **single** yellow sphere in our world. The fact that one sees more than one is caused by the fact that a line of sight goes several times around the sphere. Although a person looking in a real empty sphere would only see the back of their head, the reason Figure 16.1 shows multiple spheres in staggered positions and deviates from the theoretical view is that our polygonal representation of the sphere has modified the geometry and produced different geodesics. Figure 16.2 is the wireframe local view of the cell structure that is used to represent the world in Figure 16.1. The red cone represents the observer.

Figure 16.3 shows the view from inside the three-dimensional manifold $\mathbf{S}^3$. The cell structure of the sphere has again been reduced and consists of one 3-cell and one 0-cell. The starry sky in the distance indicates the edge of our specified viewing distance. The yellow boxes are what we see of the **single** box that we created in our world.

The cell structure of a manifold gives us all the information that one needs to define homology groups and any derived algebraic invariants. One could also define discrete geometry concepts such as the discrete curvature at edges and vertices from the attaching maps, although this has not yet been implemented.

Current Available Queries. One can ask for the homology groups.

16.5 Where to from Here?

The SPACE program is obviously only a start on a general purpose three-dimensional manifold program. For one thing, there are a lot more interesting and important invariants associated to three-dimensional manifolds other than homology groups. Some of these and algorithms for computing them can be found in [Matv03].

Although mathematicians at various university research centers have developed software over the years to study aspects of the intrinsic geometry of spaces along the lines that we have been discussing here, the programs tended to be more specialized than is the intent of the SPACE program and restricted to predefined classes of manifolds. There is no room to go into all these interesting programs here, but we mention several below.

Higher Dimensions. Four dimensions have been of particular interest. A simple example of a four-dimensional space is the *hypercube* $[0,1]^4$. Getting a good visual understanding of such a space boils down to using good projections of it to $\mathbf{R}^3$ (and then to $\mathbf{R}^2$). Thomas Banchoff and others pioneered the visualization of surfaces in $\mathbf{R}^4$ in the 1970s at Brown University. Showing the surfaces as they rotated helped give the viewer a feeling of depth. See [Banc95], [HanH92], and [HaMF94].

Special Task Programs. One research center (shut down in 1998) that seems to have produced an extraordinary number of modeling programs was the Geometry Center, a National Science Foundation Science and Technology center at the University of Minnesota. Gunn's MANIVIEW program ([Gunn93]) is probably the program that comes closest to what the SPACE program is trying to do. Here the viewer is also inside a three-dimensional manifold (one defined by a discrete group in this case). The program was an external module for the well-known surface visualization program GEOMVIEW. Another program, also from the Geometry Center, is Jeff Weeks' SNAPPEA program for studying hyperbolic 3-manifolds. (A *hyperbolic manifold* is a Riemannian manifold with constant negative sectional curvature. In a hyperbolic plane, given a line and a point not on the line, there are an infinite number of lines through the point that are parallel to the line. The sum of the angles of a triangle is less than π.). Such manifolds are important to the classification of 3-manifolds. The program is also an important tool for studying knot theory. See [Week85].

For a discussion of some further visualization programs see [HaMF94].

PART III

More on Special Computer Graphics Topics

CHAPTER 17

Computational Geometry Topics

17.1 Introduction

This chapter considers a few selected problems from the field of computational geometry. Section 17.2 discusses range trees and Section 17.3, interval and segment trees. These data structures are then used in Section 17.4 to answer questions about when bounding boxes intersect. Section 17.5 describes a simple approach to convex hull intersections. The polygon triangulation problem is considered in Section 17.6. Finally, Voronoi diagrams and the associated Delaunay triangulations are discussed in Sections 17.7 and 17.8, respectively.

17.2 Range Queries

This section describes data structures that are useful in answering range queries. Although the problem of responding to such queries in an efficient manner is interesting in its own right, the main reason for including it here is that some of the material is needed in the next section, which is on efficient bounding box intersections. In general, though, it is worth pointing out that range queries arise not only in geometry, where one might want to know which of a collection of points lie in a certain range, but also in other areas. For example, in the context of an employee database one might one to find out which employees are between 20 and 30 years old, make a salary of between \$40,000 and \$60,000, and have between one and three children.

Since it will be convenient to interpret the term "box" in a generic n-dimensional sense and to restrict ourselves to boxes whose sides are parallel to coordinate axes, we begin with a definition.

Definition. An *axis-parallel* n-dimensional box in $\mathbf{R}^n$ is any set of the form

$$[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n],$$

where $a_i \leq b_i$.

In the next section our object will be to describe quick intersection algorithms for axis-parallel line segments, rectangles, and blocks (the axis-parallel one-, two-, and three-dimensional boxes, respectively). Right now we concentrate on the problem of determining which points (0-dimensional boxes) from a collection of points fall into an axis parallel box.

Problem 1. Assume that $\mathbf{S} = \{r_1, r_2, \ldots, r_n\}$ is a **fixed** set of n real numbers. Given numbers a and b, which of the r_i lies in the segment [a,b]?

If the question in Problem 1 is only asked once, then there is a straightforward O(2n) solution. We simply compare each of the r_i with the endpoints of the segment [a,b]. However, if the question is asked many times with different segments, then it turns out that we can do better.

Step 1. We first build a balanced binary search tree T from the r_i. (For the discussion here, the term "balanced" means that all the leaves of the tree lie on adjacent levels.) To do this we sort the numbers in **S**, put the median M of the numbers in **S** at the root of T, put the median of all the numbers in **S** that are less than M at the root of the left subtree of T, put the median of all the numbers in **S** that are larger than M at the root of the right subtree of T, and so on. The actual numbers are stored at the leaves, but we use the values at the interior nodes to guide the search. See Figure 17.1.

Step 2. Search for a in T. Assume that the search ends at the leaf $\mathbf{n}_a$. Search for b in T. Assume that the search ends at the leaf $\mathbf{n}_b$. The numbers of **S** that lie in [a,b] are the leaves of T between nodes $\mathbf{n}_a$ and $\mathbf{n}_b$. For example, Figure 17.1 shows the case [a,b] = [25,72]. The leaves corresponding to the numbers that lie in the interval are shown as black rectangles.

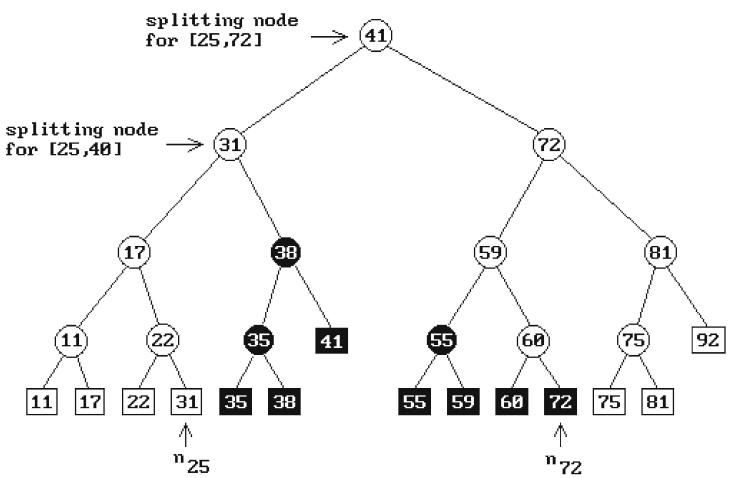


Figure 17.1. Range searching with balanced binary search trees.

Incidentally, the left and right subtrees of each interior node **n** of the tree T can be thought of as representing ranges. If the node has the value s and its parent the value t, then $[-\infty,s]$ and $[s,t]$ are the ranges of the left and right subtree of **n**, respectively.

By linking all the leaves of T together, it is easy to see that the time complexity of Problem 1 is $O(\log_2 n + k)$, where k is the number of elements of **S** that lie in [a,b]. The links take up additional space and one can do without them using a recursive algorithm that traverses all the subtrees "between" leaves $\mathbf{n}_a$ and $\mathbf{n}_b$. In Figure 17.1, the nodes that have to be traversed, other than those on the path to $\mathbf{n}_a$ and $\mathbf{n}_b$, are shown in black. Because the ideas behind implementing this recursive algorithm will be needed later, we shall describe the algorithm. An important location we have to find is the node where the paths from the root of T to $\mathbf{n}_a$ and $\mathbf{n}_b$ split. We shall call this the *splitting node*. For example, the splitting nodes for the intervals [25,72] and [25,40] are indicated in Figure 17.1. Let T_r denote the number stored at the root of a tree T. Algorithm 17.2.1 finds the splitting node. After we find the splitting node for our range problem, then we simply make one pass from that node to $\mathbf{n}_a$, printing the leaves of all the **right** subtrees of the nodes we encounter, and another pass from that node to $\mathbf{n}_b$, printing the leaves of all the **left** subtrees of the nodes we encounter. The leaves $\mathbf{n}_a$ and $\mathbf{n}_b$ themselves have to be checked separately to see if the values they hold are part of our range. Algorithm 17.2.2 describes the whole range query process.

Summarizing we get

17.2.1 Theorem. Given a set of n real numbers, these numbers can be stored in a balanced binary search tree that can be constructed in time $O(n\log_2 n)$ using $O(n)$ space. Interval queries can be made in time $O(\log_2 n + k)$, where k is number of real numbers in our set that satisfy the query.

Proof. For more details see [BKOS97].

Next, we look at the corresponding two-dimensional problem for finding points in certain ranges.

Problem 2. Assume that $\mathbf{S} = \{\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_n\}$ is a fixed set of n points in $\mathbf{R}^2$. Given a rectangle (or "window") $[a,b] \times [c,d]$, which of the $\mathbf{p}_i$ lies in this rectangle?

```
binary tree function SplitNode (binary tree T, real a, b)
begin
    while NotALeaf (T) and ((b ≤ T_r ) or (a > T_r )) do
        if b ≤ T_r
            then T := LeftSubtree (T)
            else T := RightSubtree (T);
    return T;
end;
```

Algorithm 17.2.1. Finding the splitting node.

```
procedure 1dRangeQuery (binary tree T, real a, b)
begin
    binary tree splitT;

    splitT := SplitNode (T, a, b);
    if  IsLeaf (splitT)
        then  check if we need to print splitT_r
        else
            begin
                T := LeftSubtree (splitT);
                while  NotALeaf (T)  do
                    if  a ≤ T_r
                        then
                            begin
                                PrintSubTree (RightSubtree (T));
                                T := LeftSubtree (T);
                            end
                        else  T := RightSubtree (T);
                Check if we need to print T_r;

                T := RightSubtree (splitT);
                Do a similar traversal as we did for the left subtree, except that
                    we print leaves of left subtrees;
            end
end;
```

Algorithm 17.2.2. A 1d range query algorithm.

We reduce the solution to Problem 2 to two one-dimensional problems. Let $\mathbf{p}_i = (x_i, y_i)$. Assume that all the x- and y-coordinates of our points are distinct. We need this assumption for the binary search trees we build, but shall show how to get rid of this assumption later.

Step 1. Create a balanced binary search tree T for the x-coordinates x_i like in the solution to Problem 1. This will enable us to find all the points of **S** whose x-coordinates lie in the correct range. Of course, their y-coordinates may not be in range.

Step 2. If **n** is a node of T, let P(**n**) be all the points of **S** that are leaves in the subtree of T with root **n**. Store at **n** a pointer to the balanced binary search tree T(**n**) based on the y-coordinates of the points in P(**n**), except that we also store the points $\mathbf{p}_i$ at the leaves not just simply their y-coordinates.

Definition. The data structure consisting of the tree T together with the associated trees T(**n**) is called a *range tree*.

Algorithm 17.2.3 is a more precise description of how the range tree is built. Algorithm 17.2.4 shows how a range tree is used to answer the query of Problem 2.

We summarize the results on Problem 2:

17.2.2 Theorem. Given a set of n points in $\mathbf{R}^2$, we can store them in a range tree that can be constructed in time $O(n \log_2 n)$ using $O(n \log_2 n)$ space. Axis-parallel rectangle queries can be made in time $O(\log_2^2 n + k)$, where k is number of points in our set that satisfy the query.

Proof. For a proof see [BKOS97].

In the discussion leading up to Theorem 17.2.2, we implicitly assumed that all the x-coordinates and y-coordinates are separately distinct. We shall now show that there is a simple labeling trick that will enable us to drop this assumption. The fact is that

```
range tree function RangeTree (point set P)
begin
    binary tree  T;
    range tree   R, R_L, R_R;
    real         x_mid;
    point set    P_L, P_R;

    T := balanced binary search tree based on y-coordinates of points in P with points
         also stored at leaves;
    if |P| = 1
        then  R := range tree with only a root at which we store the single point of P
                   and to which we associate T
        else
            begin
                x_mid := median of x-coordinates of points in P;
                P_L   := points of P with x-coordinates less than or equal to x_mid;
                P_R   := points of P with x-coordinates larger than x_mid;
                R_L   := RangeTree (P_L);
                R_R   := RangeTree (P_R);
                R     := range tree with only a root at which we store x_mid and
                         to which we associate T;
                Make R_L and R_R the left and right subtree of R, respectively;
            end;

    return R;
end;
```

Algorithm 17.2.3. Building a range tree.

```
procedure 2dRangeQuery (range tree R, real a, b, c, d)
begin
    range tree splitR;

    { Except for the fact that we are dealing with range trees, the SplitNode function
      below is just like the one in Algorithm 17.2.1. }
    splitR := SplitNode (R, a, b);
    if IsLeaf (splitR)
        then check if we need to print point stored at splitR
        else
            begin
                R := LeftSubtree (splitR);
                while NotALeaf (R) do
                    if a ≤ x-value at root of R
                        then
                            begin
                                1dRangeQuery (T (Root (R)), c, d); {query on y}
                                R := LeftSubtree (R);
                            end
                        else R := RightSubtree (R); {query on y}
                Check if we need to print point stored at R;

                R := RightSubtree (splitR);
                Do a similar traversal as we did for the left subtree, except that
                    we print leaves of left subtrees;
            end
end;
```

Algorithm 17.2.4. A 2d range query algorithm.

to be able to use our binary search trees, the only thing we really needed were linear orders on what we called the "x-" and "y-coordinates" of points and any such would do. The following lexicographic type ordering will do the job:

Interpret the phrase "the x-coordinate of (x,y) is less than the x-coordinate of (x′,y′)" to mean $x < x'$ or ($x \le x'$ and $y < y'$).
Interpret the phrase "the y-coordinate of (x,y) is less than the y-coordinate of (x′,y′)" to mean $y < y'$ or ($y \le y'$ and $x < x'$).

Another approach to solving Problem 2 is via Kd-trees. The query time using this data structure is worse, namely, $O(\sqrt{n} + k)$, but it uses only $O(n)$ space. Yet another way is by using a technique called *fractional cascading*. In this case we use more space but can get the query time down to $O(\log_2 n + k)$. See [BKOS97].

17.3 Interval and Segment Trees

We begin with the following problem:

Problem 1. Which of a collection of axis-parallel segments in $\mathbf{R}^2$ intersect a given rectangle (or "window") $[a,b] \times [c,d]$?

Let us divide the segments into four classes:

(1) Those that are contained in the rectangle.
(2) Those that intersect the boundary of the rectangle only **once**.
(3) Those that intersect the boundary exactly **twice**.
(4) Those that contain an edge of the boundary.

The segments belonging to classes (1) and (2) have an endpoint in the rectangle and can be found using our answer to Problem 2 in the last section with respect to the question "Which of the 2n endpoints lie in the rectangle?" Range trees or fractional cascading can be used to provide the answer. This leaves the edges belonging to classes (3) and (4). We concentrate on the edges parallel to the x-axis. Those parallel to the y-axis can be handled in a similar manner. It follows that all we have to do is determine which segments intersect the left vertical edge of the rectangle.

Problem 2. Which of a collection of segments in $\mathbf{R}^2$ that are parallel to the x-axis intersect a given vertical segment $a \times [c,d]$?

Problem 2 is solved in two steps.

Step 1. Determine which of the segments intersect a given vertical **line**?

Solving Step 1 is really a one-dimensional problem.

Problem 2a. Let $I = \{[a_1,b_1],[a_2,b_2], \ldots ,[a_n,b_n]\}$ be a collection of intervals. Which of those intervals contain the query point c?

Our approach to solving Problem 2a will be to use recursion. Let c_{mid} denote the median of the 2n endpoints of these intervals. We shall use c_{mid} to express I as the disjoint union of three sets I_{left}, I_{mid}, and I_{right} and replace the question about which intervals contain c with three questions. Define

I_{left} = the set of intervals in I which lie entirely to the left of c_{mid},
I_{mid} = the set of intervals in I which contain c_{mid}, and
I_{right} = the set of intervals in I which lie entirely to the right of c_{mid}.

See Figure 17.2. The interesting set is I_{mid}. When trying to determine which of its intervals contain c we have extra information in the case of this set that speeds up getting an answer. If $c < c_{mid}$, then we only have to check the left endpoints of the intervals because all the right endpoints are larger than c_{mid}. Similarly, if $c > c_{mid}$, then we only have to check the right endpoints of the intervals because all the left endpoints are smaller than c_{mid}. Therefore, let us store the left endpoints in increasing order and the right endpoints in decreasing order. We can now easily determine the intervals that contain c. If $c < c_{mid}$, then we start at the smallest left endpoint and

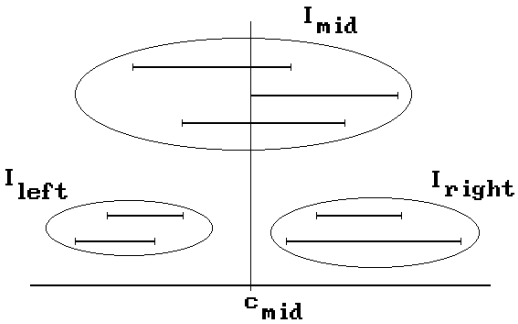


Figure 17.2. The sets I_{left}, I_{mid}, and I_{right}.

move right and report the interval with that endpoint until we find a left endpoint which is larger than c. If $c > c_{mid}$, then we start at the largest right endpoint and move left and report the interval with that endpoint until we find a right endpoint which is smaller than c. There is the case where $c = c_{mid}$, but this can easily be handled in the earlier two cases.

We are ready to describe a data structure that will match the algorithm we just outlined. Construct a binary tree T = T(I) as follows:

(1) If I is empty, let T be the empty tree.

(2) Otherwise, using the notation above, T will consist of a root and a left and right subtree, namely, $T(I_{left})$ and $T(I_{right})$, respectively. To the root node of T we associate the value $c_{mid}(T) = c_{mid}$ and two sorted lists $L_{left}(T)$ and $L_{right}(T)$, where $L_{left}(T)$ is a copy of I_{mid} sorted by the left endpoints of its intervals and $L_{right}(T)$ is a copy of I_{mid} sorted by the right endpoints.

Definition. The data structure T(I) is called an *interval tree*.

Algorithm 17.3.1 shows how an interval tree is used for a query.

17.3.1 Theorem. The interval tree associated to a set of n intervals uses O(n) storage and has height $O(\log_2 n)$. It can be built in time $O(n \log_2 n)$. If there are k intervals which contain the query point, we can report them in time $O(\log_2 n + k)$.

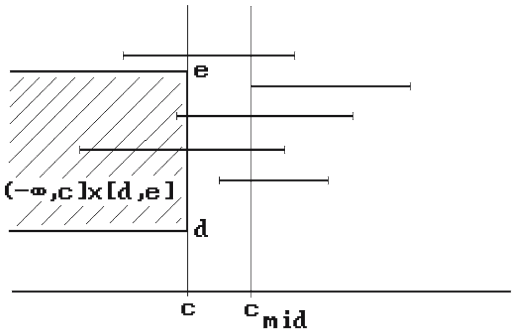
Proof. See [BKOS97]. To speed up finding the medians c_{mid} one needs to sort the whole list of endpoints once and for all.

This finishes Step 1 in the solution to Problem 2.

Step 2. Given the collection of segments that intersect a given vertical line, select those that intersect an interval in that line.

To carry out Step 2 we must modify our interval tree T. Let $c \times [d,e]$ be the vertical interval. At the moment the tree T is only concerned with the x-coordinates of points. We must also look at the y-coordinates. The problem is that the lists $L_{left}(T)$ and $L_{right}(T)$ are inadequate. What we need to do in the case of $L_{left}(T)$ is check if endpoints lie in the rectangle $(-\infty,c] \times [d,e]$. See Figure 17.3. In the case of $L_{right}(T)$ we must check if endpoints lie in the rectangle $[c,+\infty) \times [d,e]$. But we saw in Section 17.2

Figure 17.3. Checking endpoints of $L_{left}(T)$.




```
procedure IntervalTreeQuery (interval tree T, real c)
begin
    if NotALeaf (T) then
        if c < c_mid(T)
            then
                begin
                    Beginning with the leftmost interval in L_left(T) move right
                        reporting all intervals that contain c and stop at the first
                        interval that does not;
                    IntervalTreeQuery (LeftSubtree (T),c);
                end
            else
                begin
                    Beginning with the rightmost interval in L_right(T) move left
                        reporting all intervals that contain c and stop at the first
                        interval that does not;
                    IntervalTreeQuery (RightSubtree (T),c);
                end
end;
```

Algorithm 17.3.1. The interval tree query algorithm.

that a range tree is a good data structure for that. Therefore, we replace the lists $L_{left}(T)$ and $L_{right}(T)$ in T with range trees $R_{left}(T)$ and $R_{right}(T)$ determined from the left and right endpoints of I_{mid}, respectively. The only change that has to be made in Algorithm 17.3.1 for querying an interval tree is that instead of traversing the lists $L_{left}(T)$ and $L_{right}(T)$ we make queries on the corresponding range tree.

Summarizing, we get

17.3.2 Theorem. Let S be a set of n horizontal segments in the plane. If k of these intersect a vertical segment, then they can be reported in time $O(\log_2^2 n + k)$ using a data structure that uses $O(n \log_2 n)$ space and which can be built in time $O(n \log_2 n)$.

Proof. See [BKOS97].

The discussion above can be combined to answer the question in Problem 1.

17.3.3 Corollary. Let S be a set of n axis-parallel segments in the plane. If k of these intersect an axis-parallel rectangle, then they can be reported in time $O(\log_2^2 n + k)$ using a data structure that uses $O(n \log_2 n)$ space and which can be built in time $O(n \log_2 n)$.

So far we have only considered axis-parallel segments. What if our segments are not parallel to the coordinate axes? We could solve this segment-window intersection problem with a trick. Simply enclose these segments in axis-parallel rectangles, so that they become one of the diagonals of the rectangle. We can now apply what we know about axis-parallel rectangle intersections. If a rectangle intersection is found we will have to check if the segment actually intersects the window. Unfortunately, it is easy to construct examples where the rectangles intersect the window but the segments do not, so that our solution is not very efficient. Getting a better solution to this problem leads to our next data structure.

We return to an earlier query problem, Problem 2a. Let $I = \{[a_1,b_1],[a_2,b_2], \ldots, [a_n,b_n]\}$ be a collection of intervals. We want to find those intervals that contain a query point c. Let $e_1, e_2, \ldots, e_m$ be the ordered list of **distinct** left and right endpoints of the intervals in I. These points induce a partitioning of the real line into the following open and closed intervals:

$$(-\infty,e_1),[e_1,e_1],(e_1,e_2),[e_2,e_2],\ldots,(e_{m-1},e_m),[e_m,e_m],(e_m,+\infty) \tag{17.1}$$

Each of these intervals will be called an *elementary interval*. The reason for including the closed intervals that contain a single endpoint is that one may want to distinguish between a query point being an endpoint or in the interior of an interval.

The first step to finding an interval that contains the query point is to find the elementary interval to which it belongs. Therefore we build a balanced binary search tree T whose leaves correspond to the intervals in the list (17.1). If **n** is a leaf node in T, let E(**n**) denote its associated elementary interval. We could store all the intervals of I that contain E(**n**) in a list at **n**, but the only problem with this is a big cost in storage since the same intervals would potentially appear in many different lists because it might contain many elementary intervals. A more efficient approach would be to store the interval in a list at a node that is the root of that subtree of T whose leaves consist of the elementary intervals contained in that interval. We cannot quite do that but we can come close. We define a data structure S = S(I) that accomplishes this:

(1) Let T = T(S) be the balanced binary search tree T whose leaves correspond to the intervals in (17.1). If **n** is a leaf node in T, then E(**n**) will denote its associated elementary interval.
(2) If **n** is an interior node of T, then E(**n**) will denote the union of elementary intervals that are the leaves of the subtree of T with root **n**.
(3) At each node **n** of T we store E(**n**) and the subset I(**n**) of intervals in I defined by

$$I(\mathbf{n}) = \{\alpha \in I \mid E(\mathbf{n}) \subseteq \alpha \quad \text{and} \quad E(\text{Parent}(\mathbf{n})) \not\subseteq \alpha\}.$$

The subset I(**n**) will be called the *canonical subset* of the node **n**.

Definition. The data structure S(I) consisting of the annotated tree T(S) is called a *segment tree*.

Figure 17.4 shows a segment tree. Algorithm 17.3.2 shows how a segment tree is used for a query. We construct the segment tree by first sorting the endpoints of the intervals and use this ordering to build a balanced binary search tree for the elementary intervals. We also compute the intervals E(**n**) for all the nodes **n**. Then the intervals from I are inserted one at a time in the appropriate lists associated to the nodes **n** using Algorithm 17.3.3.

17.3.4 Theorem. The segment tree associated to a set of n intervals uses $O(n \log_2 n)$ storage and can be built in time $O(n \log_2 n)$. If there are k intervals that contain the query point, we can report them in time $O(\log_2 n + k)$.

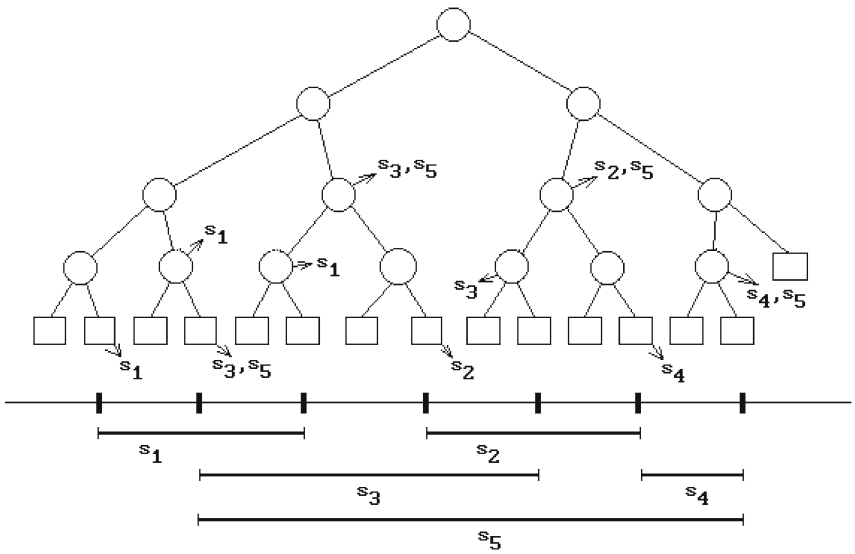


Figure 17.4. A segment tree.

```
procedure SegmentTreeQuery (segment tree S, real c)
begin
    Report all the intervals in I (Root (S));
    if NotALeaf (S) then
        if c ∈ E (Root (LeftSubtree (S)))
            then SegmentTreeQuery (LeftSubtree (S),c);
            else SegmentTreeQuery (RightSubtree (S),c);
end;
```

Algorithm 17.3.2. The segment tree query algorithm.

```
procedure SegmentTreeInsertion (segment tree S, interval α)
begin
    if E (S) ⊆ α
        then Insert α into the set I (Root (S))
        else
            begin
                if E (Root (LeftSubtree (S))) ∩ α ≠ ϕ
                    then SegmentTreeInsertion (LeftSubtree (S), α);
                if E (Root (RightSubtree (S))) ∩ α ≠ ϕ
                    then SegmentTreeInsertion (RightSubtree (S), α);
            end
end;
```

Algorithm 17.3.3. Segment tree insertion algorithm.

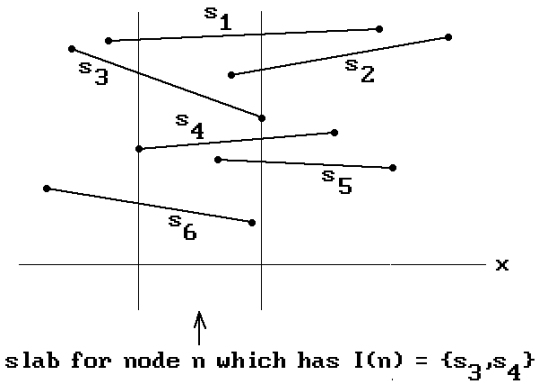
Proof. See [BKOS97]. The key observation about the space requirement is that any interval is stored in the interval list of at most two nodes per level and the height of the tree is $O(\log_2 n)$.

In this section we have described two data structures, the interval tree and the segment tree, both of which speed up queries about which intervals from a set of intervals contain a given point. The interval tree uses only linear storage and is therefore better for simple queries. On the other hand, the segment tree is better when we have in mind more complicated queries, such as testing for intersections with a given window. The reason is that the interval tree will inspect $O(\log_2 n)$ nodes in a query, but not all of the intervals associated to those nodes will contain the query point. In a segment tree search, our answer will consist of the union of each **entire** canonical subset of intervals encountered at the nodes in our search. As an example of the advantage of the segment tree we return to the non-axis-parallel segment intersection problem.

Problem 3. Which of a collection of segments in $\mathbf{R}^2$, not necessarily parallel to the x-axis but not vertical, intersect a given vertical segment $\mathbf{q} = a \times [c,d]$?

Consider the segment tree built from the x-coordinates of the segments. Alternatively, we are talking about the segment tree built on the intervals on the x-axis that are the orthogonal projection of the segments on that axis. We make one slight change in that segment tree, in that we shall store the actual segments in the canonical subsets, not their projections. Then when we encounter a node $\mathbf{n}$ of that tree in a query for a, $E(\mathbf{n})$ corresponds to a **slab** $E(\mathbf{n}) \times \mathbf{R}$ in the two-dimensional problem and so the tree gives us information about which slabs are crossed by our original segments. See Figure 17.5. Furthermore, a segment α intersecting the line $x = a$ will intersect our vertical query segment $\mathbf{q}$ if and only if the top endpoint of $\mathbf{q}$ lies above the line $L(\alpha)$ determined by α and the bottom endpoint lies below it. To find these segments, we make another assumption, namely, we assume that the interiors of all of our segments are disjoint. With that assumption, one can order the segments verti-

Figure 17.5. A slab.



cally, that is, a segment β will be above a segment α if and only if both endpoints of β lie above L(α). If both segments lie in the same line, it does not matter which is considered above which. For example, the segments in Figure 17.5 are listed in decreasing vertical order. These observations lead to the following data structure for solving Problem 3:

(1) Create a segment tree S for the intervals that are the projections of the segments on the x-axis, but store the actual segments in the canonical subsets rather than their projections.
(2) Maintain the canonical subsets I(**n**) at a node **n** as a binary search tree based on the vertical order of the segments.

To make a query with respect to the vertical segment **q**, we search the segment tree S. For each node **n** we encounter we do what amounts to a one-dimensional range query on the canonical subsets with respect to the interval [c,d]. This approach gives us.

17.3.5 Theorem. Let S be a set of n segments with disjoint interiors in the plane. If k of these intersect a vertical segment, then they can be reported in time $O(\log_2^2 n + k)$ using a data structure that uses $O(n \log_2 n)$ space and which can be built in time $O(n \log_2 n)$.

Proof. See [BKOS97].

17.3.6 Corollary. Let S be a set of n segments with disjoint interiors in the plane. If k of these intersect an axis-parallel rectangle, then they can be reported in time $O(\log_2^2 n + k)$ using a data structure that uses $O(n \log_2 n)$ space and which can be built in time $O(n \log_2 n)$.

17.4 Box Intersections

As we have pointed out in the past, computing intersections is often a very expensive operation and it would be better not to start such computations unless one was

reasonably certain that the objects in fact intersected. By associating bounding boxes to objects, a quick test for whether two objects intersect is to check if their associated bounding boxes intersect. Finding the intersection of two boxes is easy, but since bounding boxes get used a lot in computer graphics, it is worthwhile to use some especially efficient algorithms for doing this. This section describes such algorithms using the results from Sections 17.2 and 17.3.

Interval Intersections. Assume that we are given a set I of n intervals. We want to determine those that intersect a given interval [a,b]. This problem can be reduced to two searches:

Step 1: Find those intervals which contain a.
Step 2: Find those intervals whose left endpoint belongs to [a,b].

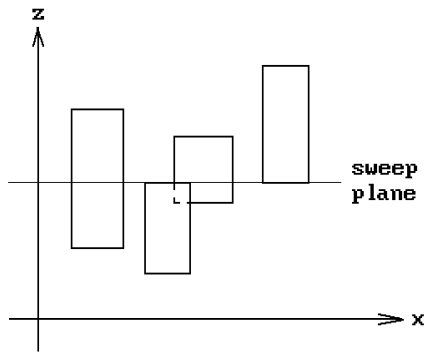
For Step 1 we can use an interval tree. This approach would take O(n) space. It takes $O(n \log_2 n)$ time to build the tree and $O(\log_2 + k)$ time for each query that has k answers. We could also use a segment tree but that takes more space, namely, $O(n \log_2 n)$. Step 2 is just a one-dimensional range query problem that can be solved with a balanced binary search tree built from the left endpoints of the intervals in I. It has the same space and time complexity as the interval tree for Step 1.

Rectangle Intersections. Assume that we are given a set R of n axis-parallel rectangles in the plane. We want to determine those that intersect a given rectangle or window $[a,b] \times [c,d]$. One approach to solving this problem is to use interval trees for the segments that make up the boundary of the rectangles. The only rectangles that will be missed are ones that either contain the window or are disjoint from it. One can check for these rectangles separately in time O(n) by checking if the vertices are either all inside or all outside the rectangle. It follows that the k intersecting rectangles can be reported in time $O(\log_2^2 n + k)$ using a data structure that uses $O(n \log_2 n)$ space and which can be built in time $O(n \log_2 n)$. Segment trees could also be used at the cost of somewhat higher storage requirement ($O(n \log_2^2 n)$).

Box Intersections. Assume that we are given a set B of n axis-parallel boxes in $\mathbf{R}^3$. We want to determine those that intersect a given box $[a,b] \times [c,d] \times [e,f]$. This problem can also be solved by reducing it to a lower-dimensional problem. By considering the rectangle faces of the boxes, we can think of it as three two-dimensional problems with an extra dimension that is handled with a balanced binary search tree. This would lead to a query complexity of $O(\log_3^2 n + k)$. A slightly faster $O(\log_2^2 n + k)$ algorithm is obtained with a sweep algorithm. We sketch this approach next. For more details see [Hoff89].

Box Intersections Based on Sweeping. We sweep a plane parallel to the x-y plane in a positive direction along the z-axis. Every box intersection will then show up as a rectangle intersection in one of these planes. See Figure 17.6. As one moves the sweep plane at height z the only time anything changes with respect to one of the boxes $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ is when one passes the top or bottom face of this box, that is, z passes the values z_1 and z_2. Let R(z) denote the set of rectangles that are the intersections of all the boxes with the sweep plane at level z. Thought of as rectangles in

Figure 17.6. Box intersections with sweeping.



the plane, we need to find the intersection of the rectangles R(z) with the query rectangle [a,b] × [c,d]. Of course, we are only interested in zs in the interval [e,f]. For a very large negative z the set R(z) will be empty. Then as z increases one needs to add and delete rectangles from R(z). We sort the z-values of the bottom and top face of all the boxes to rapidly find the z-values where R(z) changes and use interval trees for the sets R(z).

The sweeping technique can also be used to find intersections of rectangles in the plane.

17.5 Convex Set Problems

This section considers some related problems dealing with convex sets. We first describe a simple algorithm for testing whether two convex linear polyhedra intersect. We then apply this result to the well-known problem of determining the vertices of a convex hull. The results are based on the fact that disjoint convex sets can be separated by a hyperplane (see [Vale64] or [Rock70]) and that there is a straightforward algorithm for determining whether such a hyperplane exists or not.

Notation. Let **p** and **n** be a point and nonzero vector of $\mathbf{R}^n$, respectively. Let $\mathbf{H}(\mathbf{p},\mathbf{n})$ and $\mathrm{i}\mathbf{H}(\mathbf{p},\mathbf{n})$ denote the halfplane and open halfplane, respectively, defined by

$$\mathbf{H}(\mathbf{p},\mathbf{n}) = \{\mathbf{q} \mid \mathbf{n} \bullet (\mathbf{p} - \mathbf{q}) \geq 0\}$$

and

$$\mathrm{i}\mathbf{H}(\mathbf{p},\mathbf{n}) = \{\mathbf{q} \mid \mathbf{n} \bullet (\mathbf{p} - \mathbf{q}) > 0\}.$$

Let $\mathbf{P}(\mathbf{p},\mathbf{n})$ denote the hyperplane

$$\mathbf{P}(\mathbf{p},\mathbf{n}) = \mathbf{H}(\mathbf{p},\mathbf{n}) \cap \mathbf{H}(\mathbf{p},-\mathbf{n}) = \{\mathbf{q} \mid \mathbf{n} \bullet (\mathbf{p} - \mathbf{q}) = 0\}.$$

Definition. Two subsets **X** and **Y** of $\mathbf{R}^n$ are said to be *linearly separable* if there is a hyperplane $\mathbf{P}(\mathbf{p},\mathbf{n})$ so that $\mathbf{X} \subseteq \mathbf{H}(\mathbf{p},-\mathbf{n})$ and $\mathbf{Y} \subseteq \mathbf{H}(\mathbf{p},\mathbf{n})$. The hyperplane is said to *sep-*

arate **X** and **Y**. The sets are said to be *strictly linearly separable* if $\mathbf{X} \subseteq i\mathbf{H}(\mathbf{p},-\mathbf{n})$ and $\mathbf{Y} \subseteq i\mathbf{H}(\mathbf{p},\mathbf{n})$ and the hyperplane is said to *strictly separate* **X** and **Y** in this case.

Spaces that are strictly linearly separable are clearly disjoint. Spaces that are linearly separable may not be, but their intersection will be contained in the boundary of the spaces if they are both n-dimensional. For example, the squares $[0,1] \times [0,1]$ and $[1,2] \times [0,1]$ are linearly separable (using the hyperplane $\mathbf{P}((1,0),(1,0))$) even though their intersection, the line segment $1 \times [0,1]$, is nonempty.

Definition. Let $\mathbf{n} \in \mathbf{R}^n$. If $\mathbf{p} \in \mathbf{R}^n$, then define the *projection of* ***p*** *along* ***n***, $\mathbf{n}(\mathbf{p})$, by

$$\mathbf{n}(\mathbf{p}) = \mathbf{n} \bullet \mathbf{p}.$$

If $\mathbf{S} \subseteq \mathbf{R}^n$, then the *projection of* ***S*** *along* ***n***, $\mathbf{n}(\mathbf{S})$, is the subset of **R** defined by

$$\mathbf{n}(\mathbf{S}) = \{\mathbf{n}(\mathbf{p}) \mid \mathbf{p} \in \mathbf{S}\}.$$

Note that the projection of a convex set is an interval.

Some of the results that follow have a hypothesis that certain affine hulls are n-dimensional (for example, the set S in Theorem 17.5.1). We do this only to keep the statement of the results as simple as possible. If the hypothesis were not to hold we basically have a lower-dimensional problem that could also be handled easily.

17.5.1 Theorem. Let $\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s$ and $\mathbf{q}_1,\mathbf{q}_2, \ldots ,\mathbf{q}_t$ be points in $\mathbf{R}^n$. Let $\mathbf{X} = \mathbf{p}_1\mathbf{p}_2 \ldots \mathbf{p}_s$ and $\mathbf{Y} = \mathbf{q}_1\mathbf{q}_2 \ldots \mathbf{q}_t$. Assume that the affine hull of $S = \{\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s,\mathbf{q}_1,\mathbf{q}_2, \ldots ,\mathbf{q}_t\}$ has dimension n. The following two statements are equivalent:

(1) The convex hulls **X** and **Y** are linearly separable.
(2) There are linearly independent points $\mathbf{r}_1,\mathbf{r}_2, \ldots ,\mathbf{r}_n$ in S so that if **n** is any nonzero normal vector to the hyperplane that they generate, then the intervals $\mathbf{n}(\mathbf{X})$ and $\mathbf{n}(\mathbf{Y})$ intersect in at most one point.

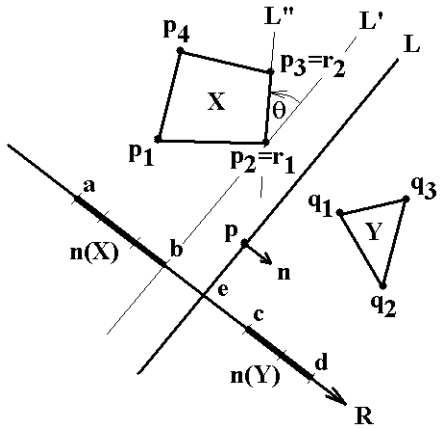
Proof. To show that (2) implies (1), assume that $\mathbf{n}(\mathbf{X}) = [a,b]$ and $\mathbf{n}(\mathbf{Y}) = [c,d]$ intersect in at most one point for some vector **n**. Without loss of generality assume that $b \le c$. Let **p** be a vertex of **X** so that $\mathbf{n}(\mathbf{p}) = b$. Clearly, $\mathbf{P}(\mathbf{p},\mathbf{n})$ is a separating plane for **X** and **Y**.

It remains to show that (1) implies (2). This is the nontrivial part of the theorem. We begin by looking at the special cases of where n is 1 or 2.

Case n = 1: The convex hulls **X** and **Y** in this case are just intervals [a,b] and [c,d]. We need to show that if they are linearly separable by a hyperplane $\mathbf{P}(\mathbf{p},\mathbf{n})$ (a point in this case) then $\mathbf{n}(\mathbf{X}) = [ka,kb]$ and $\mathbf{n}(\mathbf{Y}) = [kc,kd]$ are either disjoint or meet in at most one point, where $k = |\mathbf{n}|$. This is clear.

Case n = 2: This is the interesting case in that its proof will show what one wants to do in general. Suppose that **X** and **Y** are linearly separable via a line $\mathbf{L} = \mathbf{P}(\mathbf{p},\mathbf{n})$. Let $e = \mathbf{n}(\mathbf{p})$. It is easy to see that $\mathbf{n}(\mathbf{X}) \subseteq (-\infty,e]$ and $\mathbf{n}(\mathbf{Y}) \subseteq [e,+\infty)$, so that $\mathbf{n}(\mathbf{X})$ and $\mathbf{n}(\mathbf{Y})$ can have at most the endpoint e in common. If **L** contains two distinct vertices from **X** and **Y**, then we are done. Otherwise, consider the case where **L** is disjoint from **X** and

Figure 17.7. Determining a separating plane.



Y. (It will be obvious from the discussion of this case how to deal with the other case where **L** contains one vertex from **X** and **Y**.) See Figure 17.7. Consider the distances from the vertices of **X** and **Y** to **L**. Assume without loss of generality that it is a vertex $\mathbf{r}_1$ of **X** that is closest to **L**. Let **L′** be the line through $\mathbf{r}_1$ that is parallel to **L**. Clearly, **L′** linearly separates **X** and **Y**. If **L′** contains no vertex of **X** other than $\mathbf{r}_1$, then rotate **L′** about $\mathbf{r}_1$ through an angle θ to a line **L″** that "bumps" into another vertex of **X** or **Y** other than $\mathbf{r}_1$. We can let this vertex be $\mathbf{r}_2$. See Figure 17.7, where $\mathbf{r}_2$ belongs to **X**. For the angle of rotation simply take the minimum of all the angles between **L′** and lines through $\mathbf{r}_1$ and a vertex of **X** or **Y**. (To have well-defined angles orient the line **L′**.)

The General Case. This case is handled similar to the case where n is 2. Instead of lines we now have hyperplanes. If the separating hyperplane is disjoint from **X** and **Y**, then we translate it to pass through a point of **X**. Finally, we rotate it about this point, if necessary, until we have "bumped" into n linearly independent vertices of **X** and **Y**.

The theorem is proved.

In the special case where **Y** is a point, we can strengthen the result in Theorem 17.5.1.

17.5.2 Theorem. Let $\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s$, and **q** be points in $\mathbf{R}^n$. Assume that $\mathbf{X} = \mathbf{p}_1\mathbf{p}_2 \ldots \mathbf{p}_s$ has dimension n. Then **q** is disjoint from **X** if and only if there is a subset $\{\mathbf{r}_1,\mathbf{r}_2, \ldots ,\mathbf{r}_n\}$ of linearly independent points of $\{\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s\}$ so that the hyperplane $\mathbf{H} = \text{aff}(\{\mathbf{r}_1,\mathbf{r}_2, \ldots ,\mathbf{r}_n\})$ strictly separates **X** and **q**.

Proof. Since the point **q** is clearly disjoint from **X** if the hyperplane **H** exists, we only need to prove the converse. If **q** does not belong to **X**, we know that we can strictly separate **X** and **q** by a hyperplane $\mathbf{P}(\mathbf{p},\mathbf{n})$. By translating $\mathbf{P}(\mathbf{p},\mathbf{n})$, if necessary, we may assume like in the proof of Theorem 17.5.1 that **p** is a vertex of **X**. Consider the (n − 1)-dimensional faces of **X** that contain **p** and let $\mathbf{P}(\mathbf{p},\mathbf{n}_1)$, $\mathbf{P}(\mathbf{p},\mathbf{n}_2), \ldots ,$ $\mathbf{P}(\mathbf{p},\mathbf{n}_m)$ be the hyperplanes determined by these faces, where the normals $\mathbf{n}_i$ are

chosen so that $\mathbf{X} \subseteq \mathbf{H}(\mathbf{p},\mathbf{n}_i)$. We claim that one of the hyperplanes $\mathbf{P}(\mathbf{p},\mathbf{n}_i)$ must strictly separate $\mathbf{X}$ and $\mathbf{q}$. If this were not so, then $\mathbf{q}$ would lie in every one of the halfplanes $\mathbf{H}(\mathbf{p},\mathbf{n}_i)$ and hence lie in their intersection. Since this intersection lies in the halfplane $\mathbf{H}(\mathbf{p},\mathbf{n})$, which contains $\mathbf{X}$, it would follow that $\mathbf{q}$ lies in $\mathbf{H}(\mathbf{p},\mathbf{n})$, which is a contradiction. Therefore, without loss of generality suppose that $P(\mathbf{p},\mathbf{n}_1)$ strictly separates $\mathbf{X}$ and $\mathbf{q}$. This hyperplane must clearly contain n linearly independent points of $\{\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s\}$.

Definition. A point $\mathbf{v}$ is said to be a *vertex* or an *extreme point* of a convex set $\mathbf{C}$ if $\mathbf{v}$ does not belong to the interior of any segment whose endpoints lie in $\mathbf{C}$.

Finding the vertices of the convex hull of a finite set of points is the convex hull vertex problem. Theorem 17.5.2 leads to a simple algorithm for finding the vertices of a convex hull.

17.5.3 Corollary. Let $\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_s$ be points in $\mathbf{R}^n$. Assume that $\mathbf{X} = \mathbf{p}_1\mathbf{p}_2 \ldots \mathbf{p}_s$ has dimension n. Let $\mathbf{Y} = \mathbf{p}_1\mathbf{p}_2 \ldots \mathbf{p}_{s-1}$. Then $\mathbf{p}_s$ is a vertex of $\mathbf{X}$ if and only if either

(1) $\mathbf{Y}$ is $(n - 1)$-dimensional, or
(2) there is a subset $\{\mathbf{r}_1,\mathbf{r}_2, \ldots ,\mathbf{r}_n\}$ of linearly independent points of $\{\mathbf{p}_1,\mathbf{p}_2, \ldots ,\mathbf{p}_{s-1}\}$ so that the hyperplane $\mathbf{H} = \mathrm{aff}(\{\mathbf{r}_1,\mathbf{r}_2, \ldots ,\mathbf{r}_n\})$ strictly separates $\mathbf{Y}$ and $\mathbf{p}_s$.

Proof. The corollary easily follows from Theorem 17.5.2 and the fact that if $\mathbf{p}_s$ is a vertex of $\mathbf{X}$, then $\mathbf{p}_s$ does not belong to $\mathbf{Y}$ and can be strictly separated from that set.

Finally, it is easy to see that the proofs of the theorems above lead to an $O(m^{n+1})$ algorithm for determining whether the convex hulls of s and t points, respectively, intersect, where $m = s + t$. We get an $O(s^{n+1})$ algorithm for deciding if a point is a vertex of the convex hull of s points.

17.6 Triangulating Polygons

Polygonal sets are very common. The simplest of these are simplices. By triangulating sets one can sometimes reduce problems to problems on simplices. A good case in point is algebraic topology where triangulations can be used effectively to define and compute homology groups. There are also many places in CAD/CAGD where triangulations are useful. This section discusses triangulation algorithms for (planar) polygons (**without** holes).

If a polygon $\mathbf{P}$ defined by vertices $\mathbf{p}_1, \mathbf{p}_2, \ldots , \mathbf{p}_k$ is convex, then a simple way to triangulate it is to use triangles $\mathbf{p}_1\mathbf{p}_i\mathbf{p}_{i+1}$, $i = 2,3, \ldots ,k$. See Figure 17.8. Of course, a real algorithm would have to be more careful if one wants nondegenerate triangles because vertices might be collinear.

Now consider the general case, but before we get started, it is important that the reader understand what we mean by a polygon because that term is often used very loosely. A polygon (defined carefully in Section 6.3 in [AgoM05]) is a bounded closed planar set whose boundary is a single closed polygonal curve and as such can be

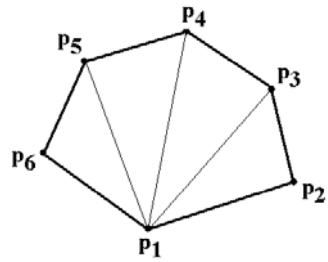


Figure 17.8 Triangulating a convex polygon.

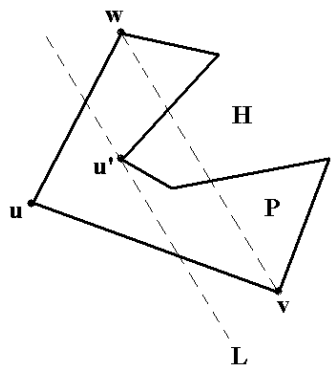


Figure 17.9. Finding a diagonal of a polygon.

thought of as being defined by a sequence of vertices. Polygons have **no** holes and they are not "self-intersecting." Some texts, such as [BKOS97], call such polygons *simple polygons*.

The approach to triangulating an arbitrary (simple) polygon will be to successively add edges between two of its vertices.

Definition. A *diagonal* of a polygon **P** is a segment (1-simplex) **uv** between two distinct vertices **u** and **v** of **P** whose interior is contained in the interior of **P**.

In Figure 17.8 the segment $\mathbf{p}_1\mathbf{p}_3$ is a diagonal, but $\mathbf{p}_1\mathbf{p}_2$ is not. A segment between two vertices that passes outside of the polygon, which could happen if it is not convex, is not a diagonal.

17.6.1 Lemma. Every polygon **P** in $\mathbf{R}^2$ with $n > 3$ vertices has a diagonal.

Proof. Order the vertices of **P** by their x-coordinates and, of the left-most vertices, let **u** be the one with the smallest y-coordinate. Let **v** and **w** be the two vertices of **P** adjacent to **u**. See Figure 17.9. If **vw** is a diagonal, then we are done. Otherwise, let S be the set of all vertices of **P** other than **u**, **v**, and **w** that lie in the triangle **uvw**. Choose a vertex **u′** in S that is farthest from the line through **v** and **w**. We claim that **uu′** is a diagonal. This is because if **L** is the line through **u′** that is parallel to **vw**, then the halfplane **H** defined by **L**, which does not contain **u** is convex and must contain all the points of S and any edges of **P** between these points.

17.6.2 Theorem. Every polygon can be triangulated. If the polygon has n vertices, then each of its triangulations will have n – 2 triangles.

Proof. Let **P** be a polygon in $\mathbf{R}^2$ with $n \geq 3$ vertices. We prove that **P** can be triangulated by induction on n. If n = 3, then we have a triangle and there is nothing to do. Assume that n > 3 and that the theorem is true for all m, $3 \leq m < n$. By Lemma 17.6.1 **P** has a diagonal. This diagonal will divide **P** into two polygons, each having fewer than n vertices. These two polygons will have triangulations by the inductive hypothesis and the union of these triangulations will define a triangulation of **P**.

Next, we show that any triangulation of **P** has n – 2 triangles. This is again proved by induction on n. Pick any diagonal of **P**. Assume that it has vertices **u** and **v** and let $\mathbf{P}_1$ and $\mathbf{P}_2$ be the two polygons into which it divides **P**. Suppose that $\mathbf{P}_i$ has n_i vertices. Clearly,

$$n_1 + n_2 = n + 2$$

since the only vertices that $\mathbf{P}_1$ and $\mathbf{P}_2$ have in common are **u** and **v**. By induction, $\mathbf{P}_i$ has $n_i - 2$ triangles. Therefore, **P** has

$$(n_1 - 2) + (n_2 - 2) = (n_1 + n_2) - 4 = (n + 2) - 4 = n - 2$$

triangles and we are done.

Theorem 17.6.2 leads to an $O(n^2)$ algorithm for a triangulation. Finding a diagonal with the argument in Lemma 17.6.1 takes O(n) steps because taking the left-most vertex, checking if the segment between the two adjacent vertices is a diagonal, and finding **u′** if not, takes at most O(n) steps. As we subdivide the polygon along the diagonal, the worst case is when one of the two polygons always ends up being a triangle. Is this the best we can do? No, we can do better. We sketch the geometric idea behind one improvement and refer the reader to [BKOS97] or [LeeP77] and [GJPT78] for more details. We already pointed out that for convex polygons one needs only O(n) steps. However, partitioning a polygon into convex pieces is itself a difficult problem and so we need a weaker property than convexity.

Definition. A polygon **P** is said to be *monotone with respect to line* **L** if every line **L′** that is orthogonal to **L** intersects **P** in a connected interval. We say that **P** is *y-monotone* if it is monotone with respect to the y-axis.

See Figure 17.10 for a polygon **P** that is y-monotone and one, **Q**, that is not. Our first task will be to divide a polygon into y-monotone pieces. We begin by classifying the vertices of the polygon.

Let **P** be a polygon defined by the vertex sequence $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k$. To simplify the notation define $\mathbf{p}_0 = \mathbf{p}_k$ and $\mathbf{p}_{k+1} = \mathbf{p}_1$. Assume that the vertices have been listed in a counterclockwise order, that is, the polygon **P** lies to the "left" of an edge $\mathbf{p}_i\mathbf{p}_{i+1}$. (The normal $\mathbf{n}_i$ with the property that $(\mathbf{p}_i\mathbf{p}_{i+1}, \mathbf{n}_i)$ induces the standard orientation points into the polygon along the edge $\mathbf{p}_i\mathbf{p}_{i+1}$.)

Definition. Define the *interior angle* θ_i at a vertex $\mathbf{p}_i$ to be the signed angle between the vectors $\mathbf{p}_i\mathbf{p}_{i+1}$ and $\mathbf{p}_i\mathbf{p}_{i-1}$. A vertex $\mathbf{p}_i$ is called a *turn vertex* if it is one of the following type:

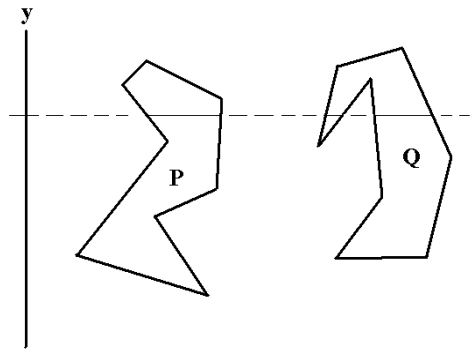


Figure 17.10. Y-monotonicity of polygons.

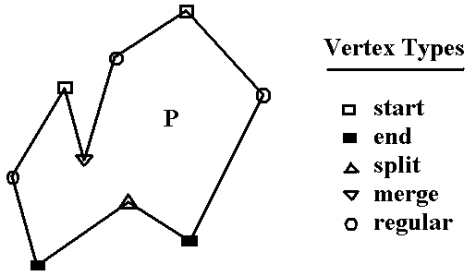


Figure 17.11. Vertex types of a polygon.

start vertex: Its two adjacent vertices lie below it and the interior angle is less than π.

end vertex: Its two adjacent vertices lie above it and the interior angle is less than π.

split vertex: Its two adjacent vertices lie below it and the interior angle is larger than π.

merge vertex: Its two adjacent vertices lie above it and the interior angle is larger than π.

If a vertex is not a turn vertex it is called a *regular vertex*. Here a vertex $\mathbf{p} = (p_x,p_y)$ is *below* another vertex $\mathbf{q} = (q_x,q_y)$ if $p_y < q_y$ or $p_y = q_y$ and $p_x > q_x$. The vertex $\mathbf{p}$ is *above* $\mathbf{q}$ if $p_y > q_y$ or $p_y = q_y$ and $p_x < q_x$.

See Figure 17.11. Note that if the adjacent vertices of a vertex lie either above or below it, then the interior angle cannot equal π. Now it is the split and merge vertices that we want to eliminate because a polygon will be y-monotone if it has no such vertices. To partition a polygon into y-monotone pieces we proceed as follows. We start at a highest vertex and sweep a line downwards. When we hit a split vertex $\mathbf{p}_i$, then we want to choose a diagonal that goes up from $\mathbf{p}_i$ and split the polygon with it. This will give us two polygons and we then work on them one at a time. If we hit a merge vertex, then we choose a diagonal that goes down. Figure 17.12 shows what would have happened to the polygon in Figure 17.11. Polygon $\mathbf{P}$ in Figure 17.12(a) would split into polygons $\mathbf{P}_1$ and $\mathbf{P}_2$ shown in Figure 17.12(b) and finally polygon $\mathbf{P}_2$ would split into polygons $\mathbf{P}_3$ and $\mathbf{P}_4$ shown in Figure 17.12(c). By choosing suitable data

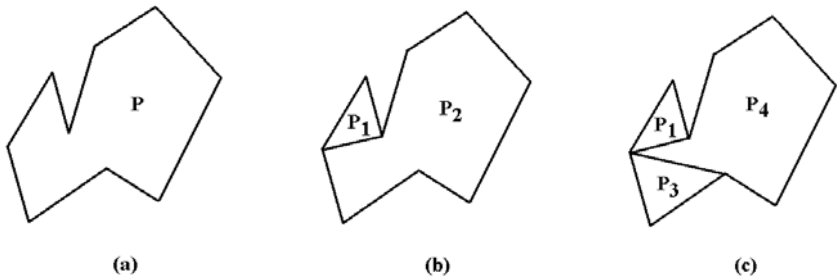


Figure 17.12. Finding y-monotone regions of a polygon.

structures such as a priority queue and balanced search trees to optimize the operations in the algorithm, one can prove the following.

17.6.3 Theorem. A polygon with n vertices can be partitioned into y-monotone regions in $O(n \log n)$ time using $O(n)$ storage.

Proof. See [BKOS97].

The next step is to find an efficient algorithm for triangulating a y-monotone polygon. The idea of such an algorithm is to walk down along the right and left part of the boundary of the polygon starting from a highest vertex, adding diagonals as we go along using a greedy type algorithm. Algorithm 17.6.1 sketches the main steps for the case of a strictly y-monotone polygon.

Definition. A y-monotone polygon is said to be *strictly y-monotone* if it does not have any horizontal edges.

At any rate, the complexity of Algorithm 17.6.1 is clearly linear in the number of vertices of the polygon, so that we get.

17.6.4 Theorem. A strictly y-monotone polygon with n vertices can be triangulated in $O(n)$ time.

Proof. See [BKOS97].

Fortunately, although the y-monotone polygons obtained from Theorem 17.6.3 may not be strictly y-monotone, they are good enough so that Algorithm 17.6.1 works on them also and so Theorems 17.6.3 and 17.6.4 lead to the following result:

17.6.5 Corollary. A polygon with n vertices can be triangulated in $O(n \log n)$ time using $O(n)$ storage.

Corollary 17.6.5 is not the best that one can do. Chazelle ([Chaz91]) has shown that one can triangulate polygons in linear time; however, the argument is rather complicated. Other references for the (simple) polygon triangulation problem are [Orou94] and [NarM95]. In some applications, such as finite element analysis, one is not satisfied with just any triangulation. In particular, skinny triangles or triangles

Inputs: A strictly y-monotone polygon **P** specified by two chains of vertices starting at the top vertex with one chain listing the vertices along the right side of the polygon and the other listing the ones on the left

Outputs: A triangulation of **P** as an appropriately specified list of edges T

```
tagged vertex p_i, p;
stack of tagged vertices S;
integer i;

Order the vertices of P by decreasing y-coordinate with vertices having the same
y-coordinate being ordered by increasing x-coordinate. Let p_1, p_2, ..., p_k be the
vertices listed in that order tagged by whether they are on the right or left side of P.

Initialize S to empty;  Add all edges of P to T;
Push (p_1,S);  Push (p_2,S);
for  i:=3  to  k−1  do
     if  p_i and Top (S) lie on different sides of the polygon
          then
               begin
                    while  |S| > 1  do
                         begin
                              p := Pop (S); Insert a diagonal from p_i to p into T;
                         end
                    Pop (S); Push (p_{i−1},S); Push (p_i,S);
               end
          else
               begin
                    p := Pop (S);
                    while  diagonal from p_i to Top (S) lies in P  do
                         begin
                              p := Pop (S); Insert a diagonal from p_i to p into T;
                         end;
                    Push (p_i,S);
               end;
Insert into T a diagonal from p_k to all the vertices on S except the first and last;
```

Algorithm 17.6.1. The triangulation algorithm for y-monotone polygons.

with angles close to π are often undesirable and a lot of work has been done to get more nicely shaped triangles in the end. As one example, we mention the paper [BeMR94] that describes a way to get a triangulation so that no angle in the triangles is larger than $\pi/2$. It starts by first packing the polygon with disks.

Finally, the triangulation algorithm we described above also applies to polygons with holes and, more generally, for arbitrary planar subdivisions. For polygons with holes see also the trapezoidation algorithm in Section 14.4 and [ZalC99].

17.7 Voronoi Diagrams

Voronoi diagrams are structures that get used in many places in geometric modeling and computational geometry. This section will define them and summarize some of their more important properties. Some good references for more information about them are [Aure91], [BKOS97], [PreS85], and [Edel87].

The general problem that Voronoi diagrams address is:

> For each point **p** in a given a set **S** of points of a space **X**, find all the points of **X** that are closest to **p**.

We make this more precise. We shall only consider Voronoi diagrams for the plane, that is, the case $\mathbf{X} = \mathbf{R}^2$. This is the case that one encounters most often. At the end of the section we shall make some comments about other cases.

Definition. Let $\mathbf{p}, \mathbf{q} \in \mathbf{R}^2$, $\mathbf{p} \neq \mathbf{q}$. Define the halfplane $\mathrm{h}(\mathbf{p},\mathbf{q})$ by

$$\mathrm{h}(\mathbf{p},\mathbf{q}) = \{\mathbf{r} \in \mathbf{R}^2 \mid \quad |\mathbf{pr}| \leq |\mathbf{rq}|\}.$$

It is easy to show that $\mathrm{h}(\mathbf{p},\mathbf{q})$ actually is a halfplane. See Figure 17.13. In fact, it is the halfplane containing **p** determined by the perpendicular bisector of the segment $[\mathbf{p},\mathbf{q}]$, that is,

$$\mathrm{h}(\mathbf{p},\mathbf{q}) = \left\{\mathbf{r} \in \mathbf{R}^2 \mid \mathbf{pq} \bullet \left(\mathbf{r} - \frac{\mathbf{p}+\mathbf{q}}{2}\right) \leq 0\right\}.$$

This fact follows easily from the identity

$$\mathbf{pq} \bullet \left(\mathbf{r} - \frac{\mathbf{p}+\mathbf{q}}{2}\right) = \frac{|\mathbf{pr}|^2 - |\mathbf{rq}|^2}{2}.$$

Now let **S** be a finite set of points in $\mathbf{R}^2$. The elements of **S** will be called *sites*.

Definition. The *Voronoi cell of a point* **p** in **S**, denoted by $\mathrm{V}(\mathbf{p})$, is defined by

$$\mathrm{V}(\mathbf{p}) = \bigcap_{\mathbf{q} \in \mathbf{S} - \{\mathbf{p}\}} \mathrm{h}(\mathbf{p},\mathbf{q}).$$

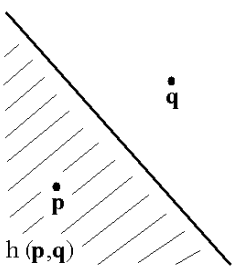


Figure 17.13. A halfplane $\mathrm{h}(\mathbf{p},\mathbf{q})$.

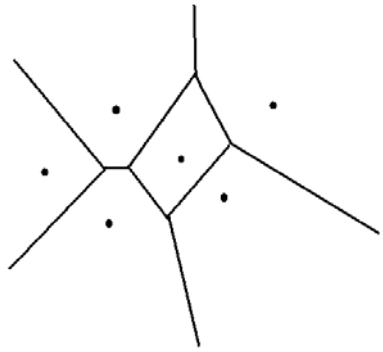


Figure 17.14. A Voronoi diagram.

Figure 17.14 shows a Voronoi diagram for six sites. Here are a few basic facts about the sets V(**p**):

17.7.1 Proposition. Let **S** be a finite set of n points in the plane.

(1) Each V(**p**) is a nonempty convex set because it is an intersection of (convex) halfplanes that contain **p**. It has a nonempty interior because it contains a small neighborhood about **p** consisting of points that belong to no other Voronoi cell.

(2) The boundary of each V(**p**) is the union of at most n – 1 segments or halflines, called the *edges* of V(**p**). (We assume that an edge is a maximal linear subset of the boundary, that is, the lines determined by the edges are distinct, or, alternatively, no two edges lie in the intersection of the same pair of Voronoi cells.) The boundary has at most n – 1 vertices, where a *vertex* is the intersection of two edges.

(3) Each point in the interior of an edge of a V(**p**) is equidistant to **precisely** two sites. Each vertex is equidistant to at least three sites.

(4) If $\mathbf{p} \neq \mathbf{q}$, then $V(\mathbf{p}) \cap V(\mathbf{q})$ is either empty or consists of a single edge.

(5) Every point of the plane belongs to at least one Voronoi cell V(**p**).

Proof. Easy. See [Aure91] or [BKOS97].

Definition. The *Voronoi diagram of the set* **S**, denoted by Vor(**S**), is the set of Voronoi cells V(**p**), $\mathbf{p} \in \mathbf{S}$, together with their edges and vertices.

Proposition 17.7.1(4) and (5) imply that Vor(**S**) defines a kind of partition of the plane. The only reason that we cannot call it a cell complex is that it contains unbounded sets that are homeomorphic to half-open disks. (Cell complexes are defined in Section 7.2.4 in [AgoM05]. Their cells must be homeomorphic to closed disks.)

Definition. Given a point **p**, let $C_{\mathbf{S}}(\mathbf{p})$ denote the largest circle centered at **p** that contains no site of **S** in its interior.

17.7.2 Proposition. Let **S** be a finite set of n points in the plane.

(1) Vor(**S**) contains precisely n "2-cells."

(2) The Voronoi diagram contains no vertices if and only if all sites are collinear.

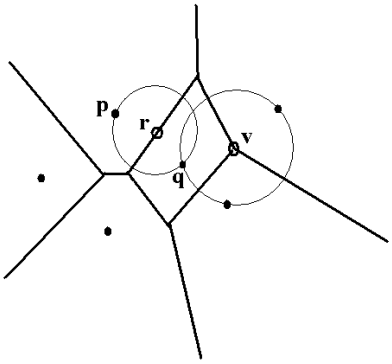


Figure 17.15. Characterizing vertices and edges of a Voronoi diagram.

(3) A point **v** is a vertex of Vor(**S**) if and only if $C_S(\mathbf{v})$ passes through at least three sites in **S**. See Figure 17.15.

(4) Two Voronoi cells V(**p**) and V(**q**) determine an edge of Vor(**S**) if and only if there is a point **r** so that $C_S(\mathbf{r})$ contains **p** and **q** in its boundary but no other site. See Figure 17.15.

(5) If no four points of **S** lie on a circle, then every vertex of Vor(**S**) belongs to precisely three edges of Vor(**S**).

(6) Vor(**S**) contains at most 3n – 6 edges and at most 2n – 5 vertices.

Proof. See [BKOS97]. (3) and (4) characterize the vertices and edges of Vor(**S**).

Next, we consider the question of how to compute the Voronoi diagram for n sites efficiently. Because the problem of sorting n numbers can be reduced to computing a Voronoi diagram, one should not expect any algorithm to be faster than $O(n \log n)$. Fortune's algorithm ([Fort87]) is an algorithm that achieves this best complexity.

17.7.3 Theorem. The Voronoi diagram of a set of n points in the plane can be computed in time $O(n \log n)$ using $O(n)$ space.

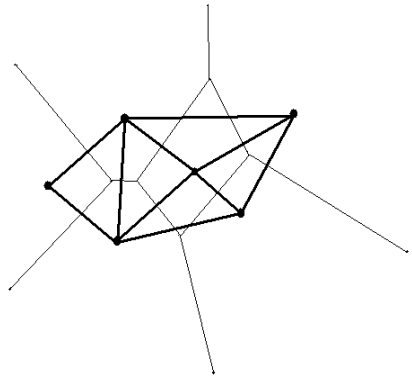
Proof. See [BKOS97].

Voronoi diagrams can be generalized to higher dimensions. It has been shown that Voronoi diagrams for n sites in $\mathbf{R}^m$ can be computed in $O(n \log n + n^{\lceil m/2 \rceil})$ optimal time. The important special case of Voronoi diagrams for three-dimensional linear polyhedra has been studied extensively. A recent algorithm and implementation is described in [EtzR99]. It separates the problem into two parts. First, a symbolic representation called the *Voronoi graph* is computed and then the geometric part, namely, the actual vertices and edges.

17.8 Delaunay Triangulations

Closely related to Voronoi diagrams are Delaunay triangulations. Again we shall restrict ourselves to subsets of the plane.

Figure 17.16. A Delaunay triangulation.



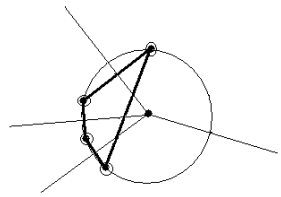


Figure 17.17. A nontriangular face of a Delaunay cell complex.

Definition. Let **S** be a finite set of points (sites) in the plane. The *Delaunay graph* of **S** is the graph with vertex set **S** and whose edges are defined by the condition that two vertices are connected by an edge if and only if the Voronoi cells associated to those vertices share a common edge. The *Delaunay cell complex* of **S** is the two-dimensional cell complex defined by the condition that its vertices and edges are the vertices and edges of the Delaunay graph of **S**. The 2-cells are called the *faces* of the Delaunay cell complex. By subdividing the faces of the Delaunay cell complex into triangles, if they are not that already, one obtains a two-dimensional simplicial complex called a *Delaunay triangulation* of **S**.

Figure 17.16 shows the Delaunay cell complex associated to the Voronoi diagram in Figure 17.14. As we can see, it is in fact a Delaunay triangulation. Figure 17.17 shows a Delaunay cell complex defined by the four sites marked by circles. It consists of a single face that is a quadrilateral and not a triangle. It admits two Delaunay triangulations. Now, the fact that we always do get a real cell complex needs some justification, which is provided by the properties of the Delaunay graph listed in the following theorem.

17.8.1 Theorem. The Delaunay graph for a set **S** satisfies the following properties:

(1) The Delaunay graph G is the "dual graph" of the Voronoi diagram D in the sense that a Voronoi cell in D becomes a vertex in G, an edge **e** in D defines an edge in G between the two vertices in G that correspond to the Voronoi cells that have **e** in common, and a vertex **v** in D defines the (triangular) region bounded by edges of G which connect the Voronoi cells that have **v** in common. (Compare this to the barycentric subdivision of a simplicial complex.)

(2) An edge is orthogonal to the edge the Voronoi cells share but does not necessarily intersect that edge.

(3) A Delaunay graph of a planar set is a planar graph.

(4) If no four points of **S** lie on a circle, then each face of the Delaunay cell complex is a triangle.

(5) Three points of **S** are vertices of the same face of the Delaunay cell complex if and only if the circle defined by them contains no point of **S** in its interior.

(6) Two points of **S** define an edge in the Delaunay graph if and only if there is a closed disk that contains those points in its boundary and which contains no other points of **S**.

(7) A Delaunay triangulation contains at most $3n - 6$ edges and $2n - 4$ faces.

Proof. See [Aure91] or [BKOS97]. Parts (5), (6), and (7) follow from Theorem 17.7.2(3), (4), and (6).

Definition. Let **S** be a finite set of points in $\mathbf{R}^2$. A *triangulation of* ***S*** is a simplicial complex K whose vertex set is **S** and whose underlying space is the convex hull of **S**.

17.8.2 Theorem. Let **S** be a finite set of points in the plane and let K be a triangulation of **S**. Then K is the Delaunay triangulation of **S** if and only if the circle circumscribing any triangle in K does not contain any point of **S** in its interior.

Proof. The theorem is an easy consequence of Theorem 17.8.1(5) and (6).

One of the important uses of Delaunay triangulations is to find a triangulation of the convex hull of a set of points with the property that the triangles have as good a shape as possible. For example, long thin triangles are undesirable in many applications.

Definition. Let **S** be a finite set of points in the plane and let K be a triangulation of **S**. If K has m triangles and if we order the 3m angles θ_i of these triangles in increasing order so that $\theta_i \le \theta_j$ if $i < j$, then the vector $(\theta_1,\theta_2, \ldots ,\theta_{3m})$ is called an *angle vector* of K. The triangulation K is called *angle-optimal* for **S** if $(\theta_1,\theta_2, \ldots ,\theta_{3m}) \ge (\tau_1,\tau_2, \ldots , \tau_{3m})$ for all angle vectors $(\tau_1,\tau_2, \ldots ,\tau_{3m})$ of triangulations of **S**.

17.8.3 Theorem. Let **S** be a finite set of points in the plane.

(1) Any angle-optimal triangulation of **S** is a Delaunay triangulation of **S**.

(2) Any Delaunay triangulation of **S** maximizes the minimal angle of the triangles in any triangulation of **S**. If no four points of **S** lie on a circle, then the Delaunay triangulation of **S** is angle-optimal.

Proof. See [BKOS97].

The property in Theorem 17.8.3(2) can be expressed in another way. Note that there are two ways that a quadrilateral can be triangulated because we can choose either of the two diagonals. Given a triangulation, consider quadrilaterals that consist of two adjacent triangles in that triangulation. A Delaunay triangulation has the

property that, for all such quadrilaterals, the minimum of the six angles in the two triangles that triangulate it will not increase if one switches to the other triangulation and replaces the diagonal that is the common edge by the other diagonal. Although Delaunay triangulations maximize the minimal angle of triangles, it is easy to give examples that it does not minimize the maximal angle.

17.8.4 Theorem. Using a Voronoi diagram a Delaunay triangulation can be computed in O(n) time after a precomputation time of the Voronoi diagram of $O(n \log n)$.

Proof. See [Aure91] or [BKOS97].

Rather than computing a Delaunay triangulation from a Voronoi diagram, which involves computing and storing the vertices of that diagram, there are algorithms that compute it directly. The three main approaches are divide-and-conquer ([GuiS85] and [Dwye87]), sweepline ([Fort87]), and incremental ([GreS77], [ClaS89], [BDST92], [FanP93], [BoiT93], [Lisc94], [BKOS97], [Devi98]). According to tests described in [SuDr95] and [Shew96] the divide-and-conquer algorithms seem to be the fastest with the sweepline algorithms the next fastest and the incremental algorithms the slowest, although the simplest. An algorithm based on the divide-and-conquer approach that also extends to higher dimensions can be found in [CiMS98].

Finally, if one wants to generalize to higher dimensions, one can build on what is known for Voronoi diagrams in that case or again try to deal with the problem directly. For a survey of this topic see [Aure91]. A specific implementation for a Delaunay triangulation for three dimensions can be found in [FanP95].

CHAPTER 18

Interval Analysis

18.1 Introduction

Interval analysis is one approach to deal with numerical errors in computations. To give an oversimplified account, it addresses a common situation in numerical computations. One has some variables that one only knows to a certain accuracy (the values lie in certain intervals) and one wants to perform some operations on them. The result therefore will only be approximate (all one can say is that it lies in some interval of values). Since we cannot assume to ever have exact numbers, we want to replace them by intervals and formalize the idea of functions from intervals to intervals. Accuracy would be specified by changing the size of the input intervals. We no longer guarantee exact numbers but instead guarantee that numbers, either input or output, lie in specified intervals. Geometric computations on a computer tend to always include some epsilons anyway and interval analysis simply formalizes the use of these epsilons.

Although interval analysis has a long history, Moore's monograph [Moor66] began a new era of applications to error analysis for digital computers. There is a large body of literature on the subject. It is not the goal of this chapter to present a thorough development of interval analysis. Rather, we have a much more modest goal to simply present some of its basic elements and indicate its relevance to obtaining robust algorithms in geometric modeling. In particular, one reason for including this material in this book was the author's interest in the generative modeling approach described in [Snyd92]. Interval analysis played a big role in Snyder's GENMOD modeler. A large part of the book [Snyd92] is devoted to showing how interval analysis can be used in algorithms important for geometric modeling. See also [Snyd92a] for a quick overview. Two other general references are [Moor79] and [AleH83]. For extensive bibliographies see [Garl85] and [Garl87].

Sections 18.2 and 18.3 present the basic definitions in interval analysis, list the most important properties of arithmetic with intervals, and discuss inclusion functions. Sections 18.4–6 describe several interval analysis algorithms that lead to robust solutions for many problems in geometric modeling. We finish with a few concluding remarks in Section 18.7.

18.2 Basic Definitions

Definition. Let I(**R**) denote the set of closed intervals in **R**.

By identifying the real number a with the interval [a,a] we shall consider **R** as a subset of I(**R**). Throughout this chapter we shall use capital letters to denote intervals.

Definition. If A = [a,b] ∈ I(**R**), then define

$$\mathrm{lb}(A) = a, \quad \text{and} \quad \mathrm{ub}(A) = b.$$

Next, here are the basic arithmetic operators of addition, subtraction, multiplication, and division on intervals.

Definition. Let $* \in \{+, -, \cdot, /\}$. Let A, B ∈ I(**R**). Define

$$A*B = \{a*b \mid a \in A \text{ and } b \in B\}$$

In the case of /, we shall always assume that 0 does not belong to B. At times we shall abbreviate the product A·B to AB.

18.2.1 Lemma. If A = [a,b] and B = [c,d], then the following holds:

(1) A + B = [a + c,b + d]
(2) A·B = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)]
(3) A − B = [a − d,b − c] = A + [−1,−1]·B
(4) A / B = [min(a/c,a/d,b/c,b/d), max(a/c,a/d,b/c,b/d)]
= [a,b]·[1/d,1/c]

Proof. This is an easy exercise.

18.2.2 Examples.
[−1,3] + [2,5] = [1,8]
[−1,3] − [2,5] = [−6,1]
[−1,3]·[2,5] = [−5,15]
[−1,3] / [2,5] = [−1/2,3/2]

The next lemma summarizes some basic facts that, among other things, show that the operations on I(**R**) act very much like they do on the reals **R**. The main fact that keeps (I(**R**), +, ·) from being a ring is that is does not have additive inverses.

18.2.3 Lemma. Let A, B, C, D ∈ I(**R**). Then

(1) (Commutativity) A + B = B + A and A·B = B·A.
(2) (Associativity) (A + B) + C = A + (B + C) and (A·B)·C = A·(B·C).
(3) (Identity) The intervals [0,0] and [1,1] are the unique additive and multiplicative identities, respectively. More precisely,

$$A = X + A = A + X \quad \text{for all } X \text{ in } I(\mathbf{R}) \text{ if and only if } X = [0,0].$$
$$A = Y \cdot A = A \cdot Y \quad \text{for all } Y \text{ in } I(\mathbf{R}) \text{ if and only if } Y = [1,1].$$

(4) $I(\mathbf{R})$ has no zero divisors.

(5) The only elements $[a,b]$ in $I(\mathbf{R})$ that have an additive or multiplicative inverse are those for which $a = b$. However, we do have

$$0 \in A - A \quad \text{and} \quad 1 \in A/A \quad \text{for all } A \in I(\mathbf{R}).$$

(6) $A \cdot (B + C) + A \cdot B + A \cdot C$ (subdistributivity)
$a \cdot (B + C) = a \cdot B + a \cdot C$, $a \in \mathbf{R}$
$A \cdot (B + C) = A \cdot B + A \cdot C$ if $bc \geq 0$ for all $b \in B$ and $c \in C$

(7) If $A \subseteq C$ and $B \subseteq D$, then $A * B \subseteq C * D$, for $* \in \{+, -, \cdot, /\}$. This is often expressed by saying that the standard interval arithmetic operators are *inclusion isotone* or *inclusion monotonic*.

Proof. This is fairly straightforward. See [AleH83].

18.2.4 Example. The distributive law fails:

$$[1,2] \cdot [(1,1) + (-1,-1)] = [0,0] \quad \text{but} \quad [1,2] \cdot [1,1] + [1,2] \cdot [-1,-1] = [-1,1].$$

Property (7) in Lemma 18.2.3 is particularly important. In the context of computations, it tells us that as new errors creep into computations we can keep control of them.

It is possible to define a metric on $I(\mathbf{R})$.

Definition. Define

$$d : I(\mathbf{R}) \times I(\mathbf{R}) \to \mathbf{R}$$

by

$$d([a,b],[c,d]) = \max(|a - c|, |b - d|).$$

18.2.5 Lemma. The function d defines a metric on $I(\mathbf{R})$.

Proof. This is easy to show directly, but actually it is the well-known Hausdorff metric.

18.2.6 Example. $d([-1,3],[2,5]) = \max(|-1 - 2|, |3 - 5|) = 3$.

18.2.7 Theorem.

(1) $(I(\mathbf{R}), d)$ is a complete metric space.

(2) The operations of addition, subtraction, multiplication, and division defined on $I(\mathbf{R})$ are continuous.

Proof. Easy.

Recall that every continuous function assumes its minimum and maximum value on a closed interval, so that the next definition is well-defined.

Definition. Let $f:\mathbf{R} \to \mathbf{R}$ be a continuous function. The function

$$f_I : I(\mathbf{R}) \to I(\mathbf{R})$$

defined by

$$f_I(A) = [\min_{a\in A} f(a), \max_{a\in A} f(a)], \quad \text{for } A \in I(\mathbf{R}),$$

is called the *induced function*.

18.2.8 Theorem. The induced function of a continuous function is a continuous function on $I(\mathbf{R})$.

Proof. This is easy to prove from the definitions.

With this notation of induced maps we shall free to write expressions such as

$$A^k, e^A, \ln A, \sin A, \text{etc.}$$

Furthermore, from Theorem 18.2.8 all of these are continuous.

18.2.9 Example. If $f(x) = 2x + 3$, then $f_I([a,b]) = [2a + 3, 2b + 3]$.

18.2.10 Example. If $f(x) = x^2$, then

$$\begin{aligned} f_I([a,b]) &= [a^2, b^2], && \text{if } a \ge 0, \\ &= [b^2, a^2], && \text{if } b \le 0, \\ &= [0, \max(a^2, b^2)], && \text{otherwise.} \end{aligned}$$

Definition. Define the *absolute value* of an interval $A = [a,b]$ in $I(\mathbf{R})$, denoted by $|A|$, by

$$|A| = d(A,[0,0]) = \max(|a|,|b|) = \max\{|x| \mid x \in A\}.$$

18.2.11 Example. $|[-1,3]| = 3$.

18.2.12 Lemma. Let $A, B, C \in I(\mathbf{R})$ and $x \in \mathbf{R}$. The absolute value function for intervals has the following properties:

(1) $|A| \ge 0$ and $|A| = 0$ if and only if $A = [0,0]$.
(2) If $A \subseteq B$, then $|A| \le |B|$.
(3) $|A + B| \le |A| + |B|$.

(4) $|xA| = |x||A|$.
(5) $|AB| = |A||B|$.
(6) $d(A,B) = |A - B|$.
(7) $d(xA,xB) = |x|\, d(A,B)$.
(8) $d(AB,AC) \le |A|\, d(B,C)$.

Proof. Straightforward.

Definition. Let $A = [a,b] \in I(\mathbf{R})$. Define the *width* of the interval A, w(A), and the *midpoint* of A, mid(A), by

$$w(A) = b - a \quad \text{and} \quad \mathrm{mid}(A) = (a+b)/2.$$

18.2.13 Lemma. Let $A, B \in I(\mathbf{R})$.

(1) $w(A) = \max_{x,y \in A} |x - y|$.
(2) If $A \subseteq B$, then $w(A) \le w(B)$.
(3) $w(A \pm B) = w(A) + w(B)$.

Proof. Easy.

We generalize to $\mathbf{R}^n$.

Definition. $I(\mathbf{R}^n) = I(\mathbf{R})^n$.

The elements of $I(\mathbf{R}^n)$ are products of intervals in $\mathbf{R}$ and clearly have the form

$$[a_1,b_1] \times [a_2,b_2] \times \ldots \times [a_n,b_n],$$

where $a_i \le b_i$. The elements of $I(\mathbf{R}^n)$ will be called *intervals* of $\mathbf{R}^n$. We extend the standard interval operations +, −, ·, and / to the intervals of $\mathbf{R}^n$ in a coordinate-wise manner.

Definition. If $\mathbf{X} \subseteq \mathbf{R}^n$, then

$$I(\mathbf{X}) = \{A \in I(\mathbf{R}^n) \mid A \subseteq \mathbf{X}\}.$$

Definition. Let $A = A_1 \times A_2 \times \ldots \times A_n$ be an interval in $\mathbf{R}^n$. Define the *width* of A, w(A), and the *midpoint* of A, mid(A), by

$$w(A) = \max(w(A_1), w(A_2), \ldots, w(A_n))$$

and

$$\mathrm{mid}(A) = (\mathrm{mid}(A_1), \mathrm{mid}(A_2), \ldots, \mathrm{mid}(A_n)).$$

Define the *absolute value* of A, |A|, by

$$|A| = \max(|A_1|, |A_2|, \ldots, |A_n|).$$

Define

$$d : I(\mathbf{R}^n) \times I(\mathbf{R}^n) \to \mathbf{R}$$

by

$$d(A,B) = |A - B|.$$

18.2.14 Lemma. The map d is a metric and $(I(\mathbf{R}^n),d)$ is a complete metric space. The maps

$$w : I(\mathbf{R}^n) \to \mathbf{R} \quad \text{and} \quad \text{mid} : I(\mathbf{R}^n) \to \mathbf{R}^n$$

are continuous functions with respect to that topology.

Proof. Easy.

In the future we shall assume that $I(\mathbf{R}^n)$ is a topological space with the topology derived from the metric d. It is just the product topology defined from the topology of $I(\mathbf{R})$. Therefore, it makes sense to talk about topological notions such as convergence. Also, we note that the properties we stated earlier for intervals in $\mathbf{R}$ have obvious generalizations and hold for intervals in $\mathbf{R}^n$. We shall not repeat them here.

Here is one last operation on intervals. It defines the smallest interval that contains two intervals.

Definition. If $A = [a,b]$ and $B = [c,d]$ are two intervals in $\mathbf{R}$, define

$$A \vee B = [\min(a,c), \max(b,d)].$$

If $A = A_1 \times A_2 \times \ldots \times A_n$ and $B = B_1 \times B_2 \times \ldots \times B_n$ are intervals in $\mathbf{R}^n$, define

$$A \vee B = (A_1 \vee B_1) \times (A_2 \vee B_2) \times \ldots \times (A_n \vee B_n).$$

18.3 Inclusion Functions

It is convenient to generalize the notion of an induced function.

Definition. Let $\mathbf{X} \subseteq \mathbf{R}^m$ and $f : \mathbf{X} \to \mathbf{R}^n$. An *inclusion function* for f is a function

$$F : I(\mathbf{X}) \to I(\mathbf{R}^n)$$

with the property that for all $A \in I(\mathbf{X})$,

$$f(\mathbf{x}) \in F(A), \quad \text{for all } x \in A.$$

The inclusion function F is called *isotonic* or *inclusion monotonic* if

$$A \subseteq B \quad \text{implies that} \quad F(A) \subseteq F(B).$$

It is said to be *convergent* if for each sequence of intervals $A_1, A_2, \ldots,$ in $I(\mathbf{X})$

$$\lim_{i\to\infty} w(A_i) = 0 \quad \text{implies that} \quad \lim_{i\to\infty} w(F(A_i)) = 0.$$

Since induced functions are continuous by Theorem 18.2.8, they are certainly convergent, because this only requires continuity at 0. An arbitrary inclusion function may not be convergent however.

Inclusion functions allow us leeway in specifying accuracy. The induced functions clearly have the tightest possible bounds. For that reason they are also often called *ideal functions*. It is often interesting to know how far an inclusion function deviates from the ideal one.

Definition. Using the notation of the previous definition, define the *excess width* of the inclusion function F at $A \in I(\mathbf{R}^n)$ to be

$$w(F(A)) - w(f_I(A)).$$

The inclusion function F is said to be of *order k* if

$$w(F(A)) - w(f_I(A)) = O\left(w(A)^k\right)$$

for all $A \in I(\mathbf{R}^n)$.

Clearly, the higher the order is for an inclusion function for a function f, the tighter it matches f.

Definition. Let $\mathbf{X} \subseteq \mathbf{R}^m$ and $f: \mathbf{X} \to \mathbf{R}^n$. We shall say that f satisfies a *Lipschitz condition* if there exists an inclusion function F for f and a constant $c > 0$ so that

$$w(F(A)) \le c\, w(A), \quad \text{for all } A \in I(\mathbf{X}).$$

Let $\mathbf{X} \subseteq \mathbf{R}^m$ and $f, g: \mathbf{X} \to \mathbf{R}^n$. Let F and G be inclusion functions for f and g, respectively. Let $* \in \{+, -, \cdot, /\}$. It is easy to show that $F * G$ defined by

$$(F*G)(A) = F(A)*G(A) \tag{18.1}$$

is an inclusion function for $f * g$. Although theoretically correct, this may not actually be true on an actual computer because of round-off errors. It can be made to work on computers that support the IEEE floating point standard though by using special rounding modes called "round-to-$-\infty$" and "round-to-$+\infty$." See [Snyd92] for more details.

At any rate, equation (18.1) means that once we have inclusion functions on some primitive functions we can compute inclusion functions for a great many other functions either by applying arithmetic operators directly or by using recursion. We

shall call an inclusion function obtained in this way from a function f the *generic inclusion function* for f. In particular, we can get inclusion functions for all polynomial functions in this way. For example, since the identity function on **R** has the identity on I(**R**) as its inclusion function, the generic inclusion function F for a polynomial

$$f(x) = c_0 + c_1 x + \ldots + c_n x^n$$

is defined by

$$F([a,b]) = c_0 + c_1[a,b] + \ldots + c_n[a,b]^n.$$

As a special case, the generic inclusion function F for $f(x) = x^2$ is defined by

$$F([a,b]) = [a,b] \cdot [a,b]. \tag{18.2}$$

On the other hand, this formula gives us

$$F([-a,a] = [-a^2, a^2]).$$

when $a > 0$. Comparing this answer to what we would get in the case of the ideal function f_I (see Example 18.2.10), we see that the inclusion function F for f is not as tight as the ideal function. This shows that generic inclusion functions have their downside. It is usually better to do some things by hand using special properties of the function. In particular, it helps to find the intervals on which the function is monotone and use that information to define the inclusion functions over those intervals separately.

So far we have dealt with vector-valued functions, but there is another important class of functions, namely, Boolean functions. These can be included in interval analysis by treating these functions as having range the set {0,1}, where 0 represents false and 1 represents true. With this interpretation it is easy to associate natural inclusion functions to the standard relational operators. For example, let

$$f, g : \mathbf{R}^n \to \mathbf{R}$$

and let F and G be inclusion functions for f and g, respectively. Here is how we define the inclusion function H for a relational operator

$$h : \mathbf{R}^n \to \{0,1\}$$

between f and g. Let $A \in I(\mathbf{R}^n)$ and assume that F(A) = [a,b] and G(A) = [c,d].

The "less than" operator $h(\mathbf{x}) = (f(\mathbf{x}) < g(\mathbf{x}))$: Define

$$\begin{aligned} H(A) &= [0,0], \quad \text{if } d \le a, \\ &= [1,1], \quad \text{if } b < c, \\ &= [0,1], \quad \text{otherwise.} \end{aligned}$$

The "equality" operator $h(\mathbf{x}) = (f(\mathbf{x}) = g(\mathbf{x}))$: Define

$$\begin{aligned} H(A) &= [0,0], \quad \text{if } d < a \text{ or } b < c, \\ &= [1,1], \quad \text{if } a = b = c = d, \\ &= [0,1], \quad \text{otherwise.} \end{aligned}$$

Note that in the special case $h(\mathbf{x}) = (f(\mathbf{x}) = 0)$, where we are simply testing for zeros of f, then

$$H(A) = [0,1], \quad \text{if and only if} \quad 0 \in [a,b] \text{ but } [a,b] \neq [0,0].$$

We can also define Boolean operators between relational expressions. For example, if

$$r_1, r_2 : \mathbf{R}^n \to \{0,1\}$$

are relational expressions with inclusion functions R_1 and R_2, respectively, then an inclusion function B for the logical **and** operator

$$b : \mathbf{R}^n \to \{0,1\}, \quad b(\mathbf{x}) = r_1(\mathbf{x}) \textbf{ and } r_2(\mathbf{x}),$$

is

$$\begin{aligned} B(A) &= [0,0], \quad \text{if } (R_1(A) = [0,0]) \textbf{ or } (R_2(A) = [0,0]), \\ &= [1,1], \quad \text{if } (R_1(A) = [1,1]) \textbf{ and } (R_2(A) = [1,1]), \\ &= [0,1], \quad \text{otherwise,} \end{aligned}$$

for all $A \in I(\mathbf{R}^n)$.

If one is going to use interval analysis, then all relevant theorems and their proofs need to be reformulated in that context. As an example of this, we consider one important theorem from calculus.

18.3.1 Theorem. (The Mean Value Theorem in Interval Analysis) Let $f : \mathbf{R}^m \to \mathbf{R}^n$ be a differentiable function. Let H_{ij} be an inclusion function for $\partial f_i/\partial x_j$ and H the inclusion function for the Jacobian matrix defined by

$$H = (H_{ij}).$$

Then there exists an inclusion function F for f satisfying

$$F(A) = F([\mathbf{x}, \mathbf{x}]) + H(A)(A - \mathbf{x}),$$

for all $A \in I(\mathbf{R}^m)$ and $\mathbf{x} \in A$.

Proof. The theorem follows from basic facts about the Taylor expansion for f and its error bounds.

Definition. The function F in Theorem 18.3.1 is called a *mean value form* for the function f.

[Snyd92] proves that if a function satisfies a Lipschitz condition, then its mean value form is of order 2, which means that it matches the function very well as the width of intervals decreases. On the other hand, this is not the case for large intervals and suggests that one should use different inclusion functions depending on the size of the intervals. The point of the mean value theorem is to get a (linear) **approximation** to a function. Therefore, use it for small intervals, but on large intervals use a more direct inclusion function for f (one obtained perhaps by formulas such as (18.1)). One important way to get good inclusion functions that are close to being ideal functions is to make use of the regions over which they are monotone. This is what gave us an answer in Example 18.2.10.

18.4 Constraint Solutions

This section presents the first of three interval analysis algorithms described in [Snyd92], which have applications to a number of important problems in geometric modeling.

Constraints on a set of points in $\mathbf{R}^n$ can usually be translated into a function

$$f : \mathbf{R}^n \to \mathbf{R}$$

with the property that

$$\begin{aligned} f(\mathbf{x}) &= 1, \quad \text{if the point } \mathbf{x} \text{ satisfies the constraints,} \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

An inclusion function F for f will take on values [0,0], [1,1], or [0,1].

Definition. An element $A \in I(\mathbf{R}^n)$ will be called an *infeasible*, *feasible*, or *indeterminate region* for F if F(A) = [0,0], F(A) = [0,0], or F(A) = [0,1], respectively.

No points satisfy the constraints in an infeasible region, all points satisfy them in a feasible region, and points may or may not satisfy the constraints in an indeterminate region.

In applications it is also useful to have an additional function

$$h : I(\mathbf{R}^n) \to \{0,1\},$$

called a *set constraint* function, which tells us whether to accept an indeterminate region as a solution. Actually, we shall use an inclusion function H for h, called a *solution acceptance set constraint* function. We have the following interpretation:

$$\begin{aligned} H(A) &= [0,0]: \quad \text{subdivide A,} \\ H(A) &= [0,1]: \quad \text{subdivide A,} \\ H(A) &= [1,1]: \quad \text{accept A as a solution.} \end{aligned}$$

The case H(A) = [0,0] is usually best handled with the function F, because for isotone functions no intervals $B \subseteq A$ would ever be accepted. On the other hand, there are functions, like w(A) < d, which are not isotone.

Algorithm 18.4.1 is a generic solution to the constraint solution problem. Variants of the algorithm are useful in certain cases.

18.4.1 Theorem. If Algorithm 18.4.1 does not find a solution, then there is not one. Also, the constraint solution algorithm converges to the actual solution if inclusion functions in the equality and inequality constraint are convergent.

Proof. See [Snyd92].

Some issues addressed in [Snyd92] regarding the use of Algorithm 18.4.1 are

```
interval list function ConstraintSolution (inclusion function F, inclusion function H,
                                           interval A)
{ F is the inclusion function for a constraint function f. H is the inclusion function
  for an solution acceptance set constraint function. }
begin
    interval list S;          { the solutions }
    interval list L;
    interval      B, B1, B2;

    S := φ;
    L := ( A );
    while not (Empty (L)) do
        begin
            B := AnyElementOf (L);
            case F (B) of
                [1,1] : Insert (B,S);
                [0,0] : ; { Discard B }
                [0,1] : if H (B) = [1,1]
                            then Insert (B,S)
                            else
                                begin
                                    Subdivide (B,B1,B2); { Subdivides the interval B }
                                    Insert (B1,B2,L);
                                end
            end
        end;

    return S;
end;
```

Algorithm 18.4.1. The constraint solution algorithm.

(1) Since the algorithm needs to terminate, how do we handle indeterminate regions?
(2) How should we divide intervals?
(3) Some problem have a finite set of points as their solution. Our algorithm produces a collection of intervals. How can we group these intervals so that each union of these groups contain a unique point from the solution?

The problem of indeterminacy is that indeterminate regions may contain a solution or they may not. Sometimes one way to handle this problem is to replace equality constraints with inequality constraints. For example, rather than checking for collisions of objects, we could check if they get sufficiently close. Another approach that works if we know that there are single solutions is to use a solution acceptance set constraint based on the size of regions. We quit if they get small enough.

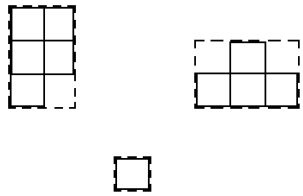
A typical approach to subdividing intervals is to divide along each individual coordinate in a cyclic fashion. This means that we divide an interval in $\mathbf{R}^n$ by bisecting the x_1-axis interval in one iteration, then the x_2-axis interval in the next iteration, and so on. After we have subdivided the x_n-axis interval, we start back with the x_1-axis interval again. Other methods can be used however.

The solution to Algorithm 18.4.1 is a collection S of intervals. It is often useful to group these together into lists that make up the connected components of the solution set. The function Components in Algorithm 18.4.2 does that and also converts the components into intervals, the list of which we then return. We are assuming that

```
interval list function Components (interval list S)
begin
    interval list C;

    Initialize C to empty;
    for each  A in S  do  C := Merge (A,C);
    return C;
end;

interval list Merge (interval A, interval list C)
begin
    for each  B in C  do
        if  (B ∩ A) ≠ ϕ  then
            begin
                A := A ∨ B;
                Delete (B,C);
            end;
    Insert (A,C);
    return C;
end;
```

Algorithm 18.4.2. Merging intervals into components.

Figure 18.1. Component intervals.



the components are sufficiently separated. The dashed rectangles in Figure 18.1 are some sample rectangles returned by the algorithm.

Finally, [Snyd92] points out that interval Newton methods can be exploited to improve the bounds on the set of solutions to the constraint problem. Furthermore, such methods provide very robust methods for solving for zeros of sets of equations.

18.5 An Application: Implicit Curve Approximations

The goal of this section is to clarify Algorithm 18.4.1 by applying it to a concrete problem. Before describing the general implicit curve algorithm we work through an example.

18.5.1 Example. Let

$$f(x,y) = y - x^2.$$

The problem is to find an approximation to the part of the implicit curve in the plane defined by

$$f(x,y) = 0$$

that lies in the unit square $A = [0,1] \times [0,1]$. See Figure 18.2(a).

Solution. The generic inclusion function F for f is

$$F([a,b] \times [c,d]) = [c,d] - [a,b]^2 = \left[c - b^2, d - a^2\right].$$

As solution acceptance set constraint function we choose

$$H(B) = (w(B) < d), \quad \text{for} \quad B \in I(\mathbf{R}^2) \quad \text{and some fixed} \quad d > 0.$$

Figures 18.2(b)–(e) show several iterations of Algorithm 18.4.1. The shaded rectangles are the rectangles that the algorithm generates and which are subdivided in the next iteration. For example, as we move from Figure 18.2(d) to Figure 18.2(e) we lose rectangle $[1/2,3/4] \times [3/4,1]$ because the rectangle

$$F([1/2,3/4] \times [3/4,1]) = [3/16,7/16]$$

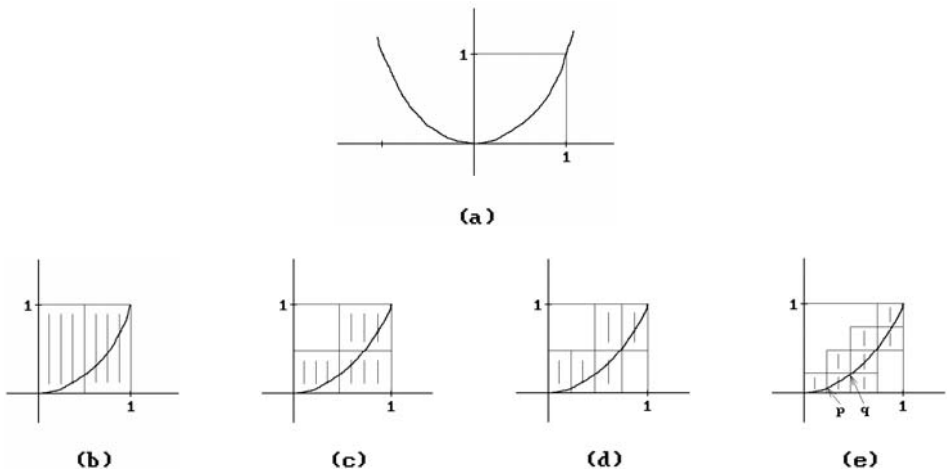


Figure 18.2. Generating the proximate intervals.

does not contain 0. Note also how the rectangles got divided. In the first iteration shown in Figure 18.2(b) we subdivided in the x-direction. In the next iteration in Figure 18.2(c) we subdivided in the y-direction, and so on.

When Algorithm 18.4.1 stops we will have the list S of rectangles of width less than d, which cover the curve we are after. (In the general algorithm, the rectangles of S are called the "proximate intervals"). Next, we will approximate the curve in each rectangle A of S and then piece these local solutions together to get complete answer. Two facts that we use here are, first, that the curve intersects A in a set that is the graph of a function, in this case a function of x and, second, that it intersects the boundary of A in a finite set of points. To find the latter intersection points we run Algorithm 18.4.1 again to find the intersection of the curve with each edge of the rectangles. This amounts to many runs of Algorithm 18.4.1 with two constraints. One is

$$f(x,y) = 0$$

and the other is of the form

$$x = \text{constant} \quad \text{or} \quad y = \text{constant}.$$

Each run will give us a set of one-dimensional solution intervals. For example, doing this for rectangle $[1/4,1/2] \times [0,1/4]$ in Figure 18.2(e) we would get two vertical intervals around the points **p** and **q**. In our case the pieces of curve that we generate in each rectangle will be straight line segments. These are then scanned to connect them together correctly. This finishes the example.

The general implicit curve approximation algorithm, Algorithm 18.5.1, comes from [Snyd92]. We assume that the implicit curve is a one-dimensional manifold without boundary. The consequence of the no boundary part that we use is that the curve has no endpoints in the interior of any interval. Step 1 has already been

The Implicit Curve Approximation Algorithm:

Assume given a curve **C** in the plane defined by f (x,y) = 0, where $f : \mathbf{R}^2 \to \mathbf{R}$.
We assume further that

(1) **C** is a one-dimensional manifold without boundary.
(2) **C** meets vertical or horizontal lines transversally, that is, the intersection consists of a finite set of points, possibly empty. Actually, this can be weakened and need only hold for the boundary segments of the proximate intervals in Step 1 below.

Inputs: An interval A that contains the part of the curve **C** we are after.
An inclusion function F for f defined on intervals in I(A).
An approximation acceptance inclusion function H defined on I(A) which specifies whether an interval is small enough so that the part of the curve it contains can be classified appropriately topologically.

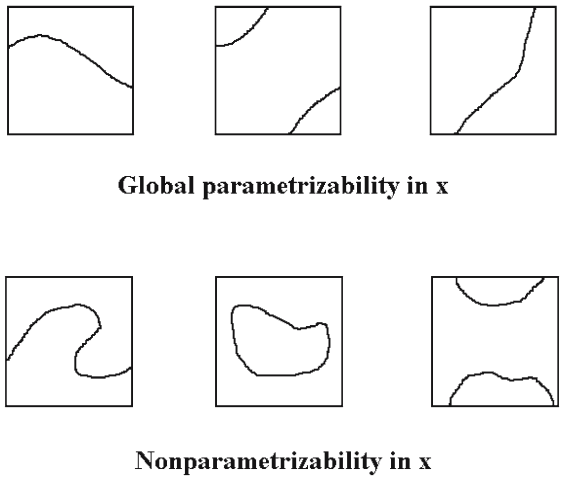
Output: A linked list of polygonal curves that approximate **C** on A.

Step 1: Use Algorithm 18.4.1 to compute the list of intervals in I(A) which satisfy H and cover the curve.
Step 2: Check each proximate interval for global parameterizability. Any interval that does not satisfy this property is subdivided further until it does.
Step 3: Use Algorithm 18.4.1 again to find the intersection of **C** with the boundaries of all the proximate intervals.
Step 4: Ensure that the intersection intervals we get from Step 3 are disjoint in the global parameterization coordinate.
Step 5: Determine the connection of the boundary intersections in each proximate interval. An interval B which that does not contain any or just one intersection points can be discarded. If B contains more than one intersection point, sort them in the global parameterization coordinate. For each successive pair of intersections in this order use Algorithm 18.4.1 to see if the curve **C** intersects the line that is the perpendicular bisector of the segment connecting the two points. If it does, then the two intersections are connected in a list. In the end, each proximate interval will have a list of point lists associated to it that correspond to the components of the intersection of the curve with that interval.
Step 6: Find the set of disjoint polygonal curve pieces that correspond to the different components of **C** on A. Make sure that the points of all these curves are ordered in a globally consistent manner.

Algorithm 18.5.1. The implicit curve approximation algorithm.

explained in Example 18.5.1. The intervals one gets are called the *proximate intervals*. For Step 2, recall that, given an equation of the form f(x,y) = 0, the implicit function theorem (Theorem 4.4.7 in [AgoM05]) asserts conditions under which one can solve for one variable in terms of the other in the neighborhood of a solution. Being able

Figure 18.3. Parameterizability in x.



to do so essentially means that the set defined by the equation can be represented as the graph of a function locally. [Snyd92] defines a curve to be *globally parameterizable in the ith coordinate* in a proximate interval if no two distinct points of the curve in the interval have the same ith coordinate. See Figure 18.3. He gives an interval version of the implicit function to be used for testing for that condition. He also describes a heuristic test for this condition that avoids expensive computations with Jacobians. In Example 18.5.1 we have global parameterizability in the y-coordinate because

$$\frac{\partial f}{\partial y} = 1 \neq 0.$$

We also have it in the x-coordinate, but because

$$\frac{\partial f}{\partial x} = -2x$$

vanishes when x is 0, we do not get this fact entirely from the implicit function theorem. The way that this condition gets used in Algorithm 18.5.1 is that when it holds, only adjacent pairs of points are connected by curve segments after the points of intersection of the curve with the boundary of a proximate interval are ordered by the ith coordinate. For Step 3 we need assumption (2) in the algorithm. By tagging edges appropriately we can arrange it so that a computation is done only once for each edge (not once for each of the two adjacent intervals, or four in the case of a corner). See Figure 18.4. The disjointness of the intervals achieved in Step 4 is needed to sort them. We need this ordering in Step 5.

Finally, [Snyd92] describes ways to relax the assumptions needed for Algorithm 18.5.1, namely, the assumption that the curve is nonsingular, that it have no endpoints in the interior of proximate intervals, and that it is transverse to boundaries of intervals.

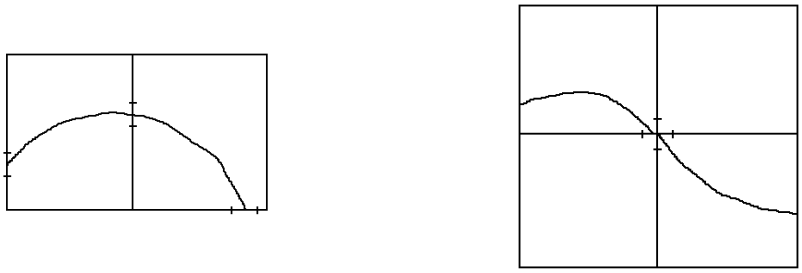


Figure 18.4. Boundary intersection sharing.

18.6 Constrained Minimizations

[Snyd92] also describes a constrained minimization algorithm. Here we are given functions

$$f, g : \mathbf{R}^n \to \mathbf{R}$$

and we seek the minimum of f on the set defined by the equation $g(\mathbf{x}) = 0$. We again choose inclusion functions F and G for f and g, respectively, and a solution acceptance set constraint H. Algorithm 18.6.1 describes the constrained minimization algorithm. The idea behind the algorithm is to progressively refine a least upper bound u of the function f. As we look at different intervals B, they affect u in the following way:

(1) If we know a feasible point $\mathbf{p}$ in $\mathbf{B}$, then we can use $f(\mathbf{p})$ as a new upper bound for a global minimum for f. In particular, if $\mathbf{B}$ is a feasible region, then any point in $\mathbf{B}$ will do.
(2) If we only know that $\mathbf{B}$ has a feasible point but do not actually know one, then $ub(F(\mathbf{B}))$ will serve as the new upper bound for a global minimum for f.
(3) If $\mathbf{B}$ is indeterminate so that we are unable to determine whether it contains a feasible point, then u cannot be updated.

Algorithm 18.6.1 has some of the same problems that Algorithm 18.4.1 has, but the following can be proved

18.6.1 Theorem. Let $B_i(k)$ be the ith interval on the priority queue Q in Algorithm 18.6.1 after the kth iteration of the while loop in the algorithm and let $l_k = lb(F(B_1(k)))$. Assume that the set of feasible points for G is nonempty and let m be the minimum of f on that set. If the inclusion functions F and G are isotone, then the numbers l_k converge to m as k goes to infinity.

Proof. See [Snyd92].

```
interval list function ConstrainedMinimization (inclusion function G, inclusion function H,
                                                inclusion function F, interval A)
{ G is the inclusion function for a constraint function g.  H is the inclusion function
  for an solution acceptance set constraint function.
  F is the inclusion function for the function f we want to minimize. }
begin
    interval list  S;                    { the solutions }
    interval priority queue  Q;
    real           u;                    { upper bound }
    interval       B, B1, B2;
    integer        i;
    point          p;

    S  := ϕ;
    Q := CreateQueue ( A );
    u  := +∞;
    while  not (Empty (Q))  do
        begin
            B := DeQueue (Q);
            if  H (B) = [1,1]
                then  Insert (B,S);
                else
                    begin
                        Subdivide (B,B1,B2);                    { Subdivides the interval B }

                        for  i:=1 to 2  do
                            if  (G (Bi) ≠ [0,0]) and (lb (F (Bi)) ≤ u)  then
                                begin
                                    EnQueue (Bi,Q);          { based on lb (F (Bi)) }
                                    if  HasIdentifiableFeasablePt (Bi,p)
                                        then  u := min (u , f (p))
                                        else  if  HasUnIdentifiableFeasablePt (Bi)
                                                  then  u := min (u , ub (F (Bi)));
                                end
                    end
        end;

    return S;
end;
```

Algorithm 18.6.1. The constrained minimization algorithm.

18.7 Conclusions

Interval analysis has many advantages. It can be implemented on a computer in hardware or software. The software only has to ensure that any rounding that takes place goes outward from the interior of an interval. It can produce robust algorithms. In fact, it has lead to new results that are not just extensions of the corresponding real number result. A good example of this is the iterative Newton-Raphson method for finding the zeros of a function f. If F is an inclusion function for f, then the interval analysis Newton-Raphson method takes the form

$$A_{n+1} = \mathrm{mid}(A_n) - \frac{F(A_n)}{F'(\mathrm{mid}(A_n))}.$$

This form is shown in [Moor66] to produce much better results than the usual Newton-Raphson method.

Some disadvantages of interval analysis are:

(1) Interval arithmetic computations are slower than floating point operations, roughly by a factor of three, although there are problems that are solved faster when implemented using interval arithmetic.
(2) There are no additive or multiplicative inverses for intervals.
(3) We do not have a strict distributive law of multiplication over addition, only subdistributivity (see Lemma 18.2.3(6)). One consequence of this is that generic inclusion functions do not give as tight a bound as would be desirable.

18.8 Exercises

Prove any or all of the unproved facts in Section 18.2.

CHAPTER 19

The Finite Element Method

19.1 Introduction

This is another very brief chapter, which is only included because the finite element method (FEM), also referred to as finite element analysis (FEA), is very important for some tasks for which geometric modelers are used and most modelers support it in one form or another. Even though it is a huge topic, it is rather specialized and tangential to the main thrust of this book. Nevertheless, it seems appropriate to say a few things about it because the term "finite element method" is often mentioned in connection with geometric modelers and so a person wanting to be knowledgeable about geometric modeling should know something about it. Two general-purpose references are [Buch95] and [John87].

Section 19.2 gives a brief overview of the topic. The mathematics behind it are described in general terms in Section 19.3 and we work through one example in Section 19.4. Finally, Section 19.5 summarizes some main points.

19.2 What Is It All About?

The explanation for most physical phenomena in science reduces to solving differential (or integral) equations. An exact solution for these equations is usually very difficult if not impossible to obtain and so what is wanted is a computable approximation to the solution. (The need for computable approximations would exist even if one could solve the equation.) The classical numerical approach to solving differential equations is using the method of finite differences. With this method, derivatives are replaced with quotients of differences and solved using the values at some finite mesh of points. See [ConD72] or [Horn75] for a general introduction to the subject. The finite element method is also a method to solve differential and integral equations. Introduced by engineers in the early 1950s and 1960s to obtain solutions to partial differential equations in structural engineering, such as elasticity and plate equations, it soon became clear that it had much wider applications. In subsequent years it came to be used in a wide variety of engineering applications. The method also subdivides

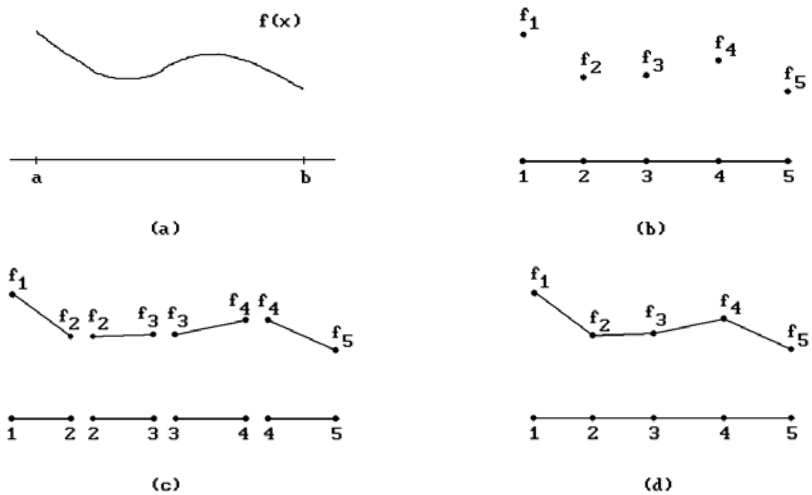


Figure 19.1. Linear one-dimensional elements and nodes.

the region of interest into a mesh of small subregions, but how these subregions are used is quite different from difference methods. Furthermore, for the finite difference method the mesh is defined by orthogonal rows and columns, whereas the finite element method allows much more general meshes.

The mathematical foundations of the FEM actually date back to variational methods introduced in the early 1900s. One can think of the FEM as modern applications of the Ritz variational and the Galerkin weighted residual methods in numerical analysis. The overall basic idea is that one subdivides problem domains into small parts called *(finite) elements* with associated simple solutions. These elements are then assembled by means of interconnections at boundary points called *nodes*. The collection of elements and nodes is called a *finite element mesh*. The simple solutions corresponding to the elements are functions of unknown values at the nodes. Let us expand on this a bit.

Suppose that we have a one-dimensional problem defined by a differential equation whose solution is a function f(x) defined over some interval [a,b]. See Figure 19.1(a). With the FEM what we do is subdivide the interval into subintervals $[x_{i-1},x_i]$, where $a = x_0 < x_1 < \ldots < x_{n-1} < x_n = b$. Over each interval we define an approximation to the solution to our problem. The positions x_i are the **nodes** and we assume that the relevant information about f is known at those points. We are free to choose our approximations to f(x) over each subinterval (the **elements**). Typically, we would use some sort of polynomial approximation. For example, if we choose linear approximations, then all we need to know are the values $f_i = f(x_i)$ to define the elements (Figure 19.1(b)). From those we can define the polygonal approximation to f(x) corresponding to the elements shown in Figure 19.1(c). Assembling the local solutions gives the global approximation shown in Figure 19.1(d). On the other hand, we might want to get a smoother approximation and try to use higher-degree polynomials. Of course, the higher the degree of the polynomial, the more computationally expensive the solution is. In higher-dimensional problems even cubic polynomial can already get very

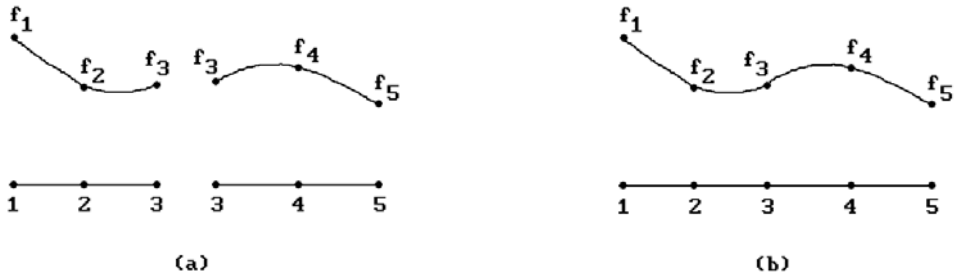


Figure 19.2. Quadratic one-dimensional elements and nodes.

expensive. Suppose that one wanted to use quadratic polynomials. The values at the endpoints of intervals would be insufficient to completely specify the polynomial since it has three degrees of freedom. What one would do is introduce an extra node on the interior of intervals. Figure 19.2 shows how a function is approximated by two quadratic elements, each of which is defined by three nodes. For a cubic one would use two extra interior nodes. As an aside, recall that a cubic is completely specified on an interval if one knows the values and derivatives at the endpoints. It might therefore occur to the reader that this would avoid the introduction of interior nodes. However, having to find the value a derivative would be expensive computationally. In any case what we have is a global solution (actually an approximation), which, being a collection of local solutions – the **elements**, depends on a finite set of unknowns, namely, some to be specified data at the **nodes**. As we shall see, the local solutions will actually be indexed by the nodes, not the elements. To summarize, the key element to getting solutions using the approach just described is having a ready collection of interpolating splines on hand to serve as basis functions for each element. These basis functions are called the *local shape functions*.

Higher-dimensional problems are handled in a similar fashion. For example, in the two-dimensional case one is trying to approximate a function defined by a differential equation over a region **A** in the plane. This region could be arbitrary and does not need to be rectangular. One has to subdivide the region by means of a mesh of nodes. Typical shapes for the elements are quadrilaterals or triangles. Over each of these regions with their associated nodes one defines approximations, which are typically low-degree polynomials (splines). One ends up with a global approximation that depends on a finite number of unknowns coming from values at the nodes. Again, the choice of basis function for each element is up to the user. Figure 19.3(a) shows a linear triangular element. Figure 19.3(b) shows a four-point interpolating surface for a quadrilateral element (see Section 12.6, equation (12.17)).

19.3 The Mathematics Behind FEM

This section gives a brief overview of the mathematics behind the FEM. As we mentioned earlier, there are basically two approaches.

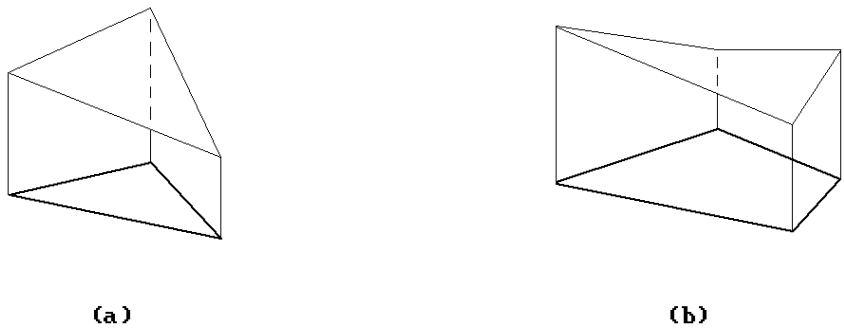


Figure 19.3. Two-dimensional elements.

The Variational Method. Here one tries to get a solution to the differential equation by translating it into a minimization problem for an energy function F that is a linear functional defined on a function space Ω. One has a problem of the form

$$\text{Find a function } u \in \Omega \text{ so that } F(u) \le F(v) \text{ for all functions } v \in \Omega. \tag{19.1}$$

In general, the functions v correspond to continuously changing quantities such as displacements of elastic bodies, temperature, etc., and F is an energy function associated to the problem. The space Ω is usually infinite dimensional and so one replaces it with a finite dimensional approximation Ω_c generated by some "simple" functions. The original problem (19.1) then becomes

$$\text{Find a function } u_c \in \Omega_c \text{ so that } F(u_c) \le F(v) \text{ for all functions } v \in \Omega_c. \tag{19.2}$$

The choice of space Ω_c is influenced by such factors as the particular formulation of the variational problem, the desired accuracy of the solution, the regularity of the exact solution, etc. It often consists of piecewise polynomial functions. The problem in (19.2) then basically becomes one of solving a large system of linear or nonlinear equations.

One issue here is whether the new solution u_c in (19.2) is an adequate approximation to the actual solution u in (19.1). (Actually, there is also the mathematical problem as to whether the variational problem (19.1) in fact has a solution in Ω, because not all do since the space may not be closed. In our case, we are only interested in approximations and can choose a closed set Ω_c.)

The basic steps in the FEM using the variational approach are:

(1) Translate the problem involving the differential equation into a variational one.
(2) Discretize using the FEM. This amount to specifying the space Ω_c.
(3) Solve the discrete problem.

Doing (1) may involve defining an artificial functional for the problem. Solving the variational problem may require less continuity than that of the actual solution. See [MitW78].

The Galerkin Method. In this method we start with an approximation g(x) to a solution f(x), where g(x) is a linear combination of functions $g_i(x)$ which are indexed by the nodes:

$$g(x) = a_1 g_1(x) + a_2 g_2(x) + \ldots + a_n g_n(x). \tag{19.3}$$

The $g_i(x)$ are typically B-spline type functions that are piecewise polynomials and vanish everywhere except on the elements adjacent to the ith node. They are called the *global shape functions*. Let

$$R(x,a) = f(x) - g(x). \tag{19.4}$$

be the error function, also called the *residual*. It depends both on x and the unknowns a_i. The *method of weighted residuals* then tries to solve for the a_i by solving

$$\int_{\mathbf{D}} w(x)R(x,a) = 0. \tag{19.5}$$

where **D** is the domain of the problem and the w(x) are one or more suitable "weighting" functions. If one applies the boundary conditions of the problem one gets a system of linear equations in the a_i that is then solved to get the solution (19.3). The Galerkin method uses the n global shape functions as weighting functions, that is, $w_i(x) = g_i(x)$. Requiring (19.5) to hold for these functions then gives n equation in the n unknown a_i.

Many of the problems to which the FEM is applied are a case of finding approximations to functions u that

(1) solve equations of the form

$$Au = f,$$

where A is a linear differential operator satisfying boundary conditions, and

(2) minimize some linear functional of the form

$$F(u) = (1/2) <Au,u> - <u,f>,$$

where <,> is an inner product of two functions.

19.4 An Example

We shall work through a fairly standard example that is an application of the FEM using the Galerkin method. Even though our one-dimensional example will be very simple, it nevertheless shows how the FEM works and more complicated examples do not involve anything different.

Our example is a one-dimensional heat conduction problem. See Figure 19.4. What we have is a rod of constant cross section and length L. We assume a given heat

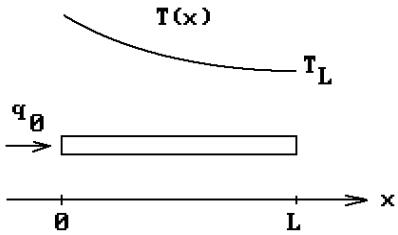


Figure 19.4. Heat conduction along a rod.

flux q_0 at one end, a constant temperature at the other, and no heat loss in between. An example of this is an insulated wire. We wish to determine the temperature T(x) along the rod. The well-known differential equation that describes this situation is

$$-K\frac{d^2T}{dx^2} = Q \quad \text{for} \quad 0 < x < L, \tag{19.6}$$

$$-K\frac{dT}{dx} = q_0 \quad \text{for} \quad x = 0, \quad \text{and} \tag{19.7}$$

$$T = T_L \quad \text{for} \quad x = L, \tag{19.8}$$

where K is the material's thermal conductivity, Q is the heat generation in the rod per unit volume, and

$$q(x) = -K\frac{dT}{dx}$$

is the heat flux. We shall assume that K and Q are constant. For reasons we shall see shortly, equation (19.6), along with the boundary conditions (19.7) and (19.8), is called the *strong form of the one-dimensional heat flow equation*. It is easy to show that the exact solution to this equation is

$$T(x) = T_L + \frac{q_0}{K}(L - x) + \frac{Q}{2K}\left(L^2 - x^2\right). \tag{19.9}$$

In general of course, one would not have an exact solution, so that having the solution (19.9) is not important. However, one is always interested in the accuracy of approximations and so it is worthwhile comparing the solution we get using the FEM to the one in (19.9).

Applying the Galerkin weighted residual method to (19.6) we get

$$\int_0^L w(x)\left(-K\frac{d^2T}{dx^2} - Q\right)dx = 0. \tag{19.10}$$

The only problem now is the high degree of differentiability that using (19.10) would require of any approximation. For example, dividing the domain [0, L] into elements and looking for a solution of the form (19.3) with linear approximations over each

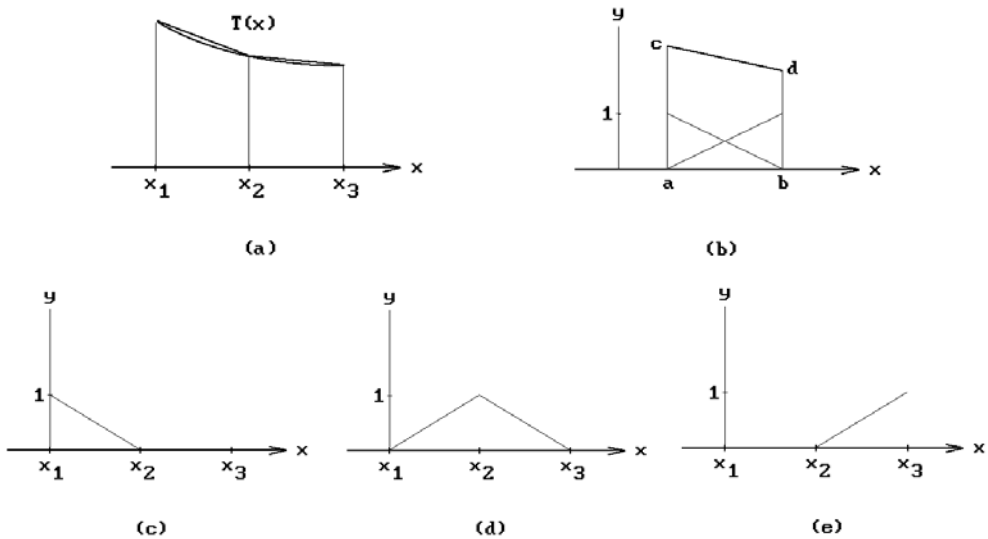


Figure 19.5. Local and global shape functions.

element would not work since the function would not be differentiable at the nodes and have zero second derivative on the interior of the elements. To get around this problem we need to reformulate (19.10). One can show, using integration by parts, that (19.10) is equivalent to

$$\int_0^L K\frac{dw}{dx}\frac{dT}{dx}dx - \int_0^L w(x)Qdx - \left[Kw(x)\frac{dT}{dx}\right]_0^L = 0. \tag{19.11}$$

The advantage with (19.11) is that only the first derivative is involved. Equation (19.11) is called the *weak form of the one-dimensional heat flow equation*. The term "weak" is used because there is less of a differentiability requirement. Using linear approximations no longer causes a problem.

The next step is to decide how many elements we want to create. Suppose that we use two. We also have to decide on basis functions. We shall use linear functions, in fact, the linear B-splines or hat functions discussed in Section 11.5. See Figure 19.5(a). For an arbitrary interval [a,b], any linear functions h(x) can be expressed as linear combinations of the two functions

$$b_a(x) = 1 - \frac{x-a}{b-a} \quad \text{and} \quad b_b(x) = \frac{x-a}{b-a} = 1 - b_a(x),$$

namely,

$$h(x) = h(a)b_a(x) + h(b)b_b(x).$$

See Figure 19.5(b). The functions $b_a(x)$ and $b_b(x)$ are the *local* (linear) shape function basis for the interval [a,b]. Using functions like this we associate a global shape func-

tion $N_i(x)$ to the node x_i. These functions are defined on **all** of **R**. Their graphs are shown in Figures 19.5(c)–(e). The functions $N_i(x)$ should have the following properties:

(1) They are defined in terms of basis functions for the elements.
(2) $N_i(x)$ is nonzero at the ith node but vanishes at all the other nodes. More precisely, we want

$$N_i(x_j) = \delta_{ij}.$$

(3) $N_i(x)$ vanishes on all elements other than the ones adjacent to the ith node.

If we had decided to use a higher degree approximation, then the only thing that would change is the local shape function basis.

Our approximation to T(x) now has the form

$$H(x) = T_1N_1(x) + T_2N_2(x) + T_3N_3(x), \tag{19.12}$$

and we solve for the T_i by substituting H(x) for T(x) and $N_i(x)$ for w(x) into equation (19.11). One will get three equations. Each integration will have to be broken up into two parts, one for each element, since there is no one formula for the $N_i(x)$ over whole interval [0, L]. We get

$$\int_0^{L/2}\left(K\frac{dN_i}{dx}\left(\sum_{j=1}^{3}\frac{dN_j}{dx}T_j\right) - N_iQ\right)dx + \left[N_i\left(-K\frac{dT}{dx}\right)\right]_0^{L/2} +$$
$$\int_{L/2}^{L}\left(K\frac{dN_i}{dx}\left(\sum_{j=1}^{3}\frac{dN_j}{dx}T_j\right) - N_iQ\right)dx + \left[N_i\left(-K\frac{dT}{dx}\right)\right]_{L/2}^{L} = 0, \quad i = 1,2,3. \tag{19.13}$$

Knowing the functions $N_i(x)$, and hence also their derivatives, we can perform the integration in (19.13). Finally, using facts about where the $N_i(x)$ vanish and initial conditions (19.7) and (19.8), the equations (19.13) simplify to

$$\frac{K}{6L}\begin{pmatrix} 14 & -16 \\ -16 & 32 \end{pmatrix}\begin{pmatrix} T_1 \\ T_2 \end{pmatrix} = \frac{QL}{6}\begin{pmatrix} 1 \\ 4 \end{pmatrix} + \begin{pmatrix} q_0 \\ 0 \end{pmatrix} + \frac{K}{6L}T_L\begin{pmatrix} -2 \\ 16 \end{pmatrix}. \tag{19.14}$$

Note that the variable T_3 does not appear since we know its value, which is T_L. The value of the heat flux at x = L is

$$-K\frac{dT}{dx}(L) = \frac{K}{6L}(16T_2 - 2T_1 - 14T_L) + \frac{QL}{6}. \tag{19.15}$$

Although we have omitted some details, which can be found in [PepH92], the steps outlined above show the general thrust of the FEM. To summarize, our approximate solution to (19.11) is the function H(x) in (19.12). The only unknowns in the formula

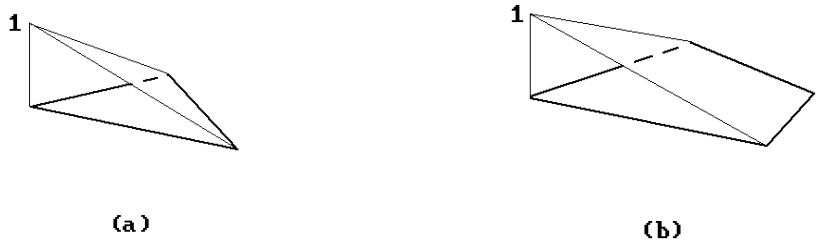


Figure 19.6. Local triangle and quadrilateral shape functions.

for H(x) were the T_i and we found those by solving a system of linear equations (19.14), which is straightforward.

Two-dimensional applications of the FEM proceed in essentially the same way. The main difference is that the differential equations get more complicated. Figure 19.6 shows some linear two-dimensional local shape functions for triangular and quadrilateral elements. Again, these functions, whether they are linear or not, have the property that they vanish at all nodes except one.

19.5 Summary

We end with some general comments. The FEM is important because most real-world problems are too complex to be solved in closed form. It is a very successful method for solving nonlinear differential and integral equations numerically. The approximations are based on local shape functions, typically polynomials, for elements whose solutions are specified by data at their nodes. This involves a choice of local shape functions and a mesh. The accuracy of the solutions is measured with respect to some suitable choice of definition for the distance between functions. Common distance functions are based on the L^p norms $\| \ \|_1$, $\| \ \|_2$, and $\| \ \|_\infty$ (see Section 21.4 for a definition). Although these are usually not easy to compute, one can find approximations to them. Rectangular or triangular meshes are common and one often uses adaptive meshes that change in size and are not uniform over the entire region of interest. Getting good meshes is very important because numeric accuracy suffers otherwise, for example, if triangles are skinny and have very large or small angles. So-called multi-grid methods have provided some of the fastest solutions.

The basic steps in the FEM approach are:

(1) Find the equations that govern the physical phenomenon.
(2) Replace these equations with their "weak" formulation.
(3) Subdivide the domain into appropriate elements and decide on the basis functions to use for each element.
(4) Combine the individual element solutions into a global solution according to the Galerkin approximation to obtain global matrix equations.
(5) Solve these matrix equations using the given boundary conditions.

Johnson ([John87]) lists the following advantages of FEM over finite difference methods:

(1) The FEM handles complicated geometry and boundary conditions better. It avoids artificial complications, is clearer, and is easier to program.
(2) The FEM is more reliable and it is easier to analyze its accuracy.

One practical difficulty that is often encountered by anyone wanting to use the FEM is finding the geometric model and physical constraints to which the analysis is applied. The original geometry usually needs to be simplified. Automating the process of producing sufficiently accurate analysis models, rather than relying on experts, would be highly desirable. We leave the reader with one reference, [Arms94], for this topic.

CHAPTER 20

Quaternions

Prerequisites: Some linear algebra and facts about motions from Chapters 1 and 2 in [AgoM05]

20.1 Introduction

In a broad sense the topic of this chapter is to determine if and how one can generalize the following two observations about the plane to $\mathbf{R}^n$:

(1) The Euclidean space $\mathbf{R}^2$ admits an algebraic structure that makes it into a two-dimensional division algebra over the reals. (This structure comes of course from the complex numbers $\mathbf{C}$ because one usually identifies $\mathbf{C}$ with $\mathbf{R}^2$.)
(2) The unit sphere (circle) $\mathbf{S}^1$ in $\mathbf{R}^2$ can be identified in a natural way with the special orthogonal group $\mathbf{SO}(2)$ and the rotations about the origin in $\mathbf{R}^2$.

Section 20.2 will begin with some general comments about the generalization of observation (1) and this will lead us to the quaternions, a less-known, but nevertheless very useful, generalization of the complex numbers. We shall define them and summarizes some of their basic properties. Section 20.3 will discuss the important connection of quaternions to transformations and this connection can be thought of as a generalization of observation (2).

20.2 Basic Facts

To be a division algebra basically means is that we have a vector space with the usual vector addition along with a multiplicative structure that includes multiplicative inverses. In the case of the complex numbers the multiplication is actually commutative, so that they are a field over the reals, but commutativity is not required for division algebras in general. What about higher-dimensional Euclidean spaces?

Question 1. For which k does $\mathbf{R}^k$ admit a multiplicative structure?

Note that the cross-product in $\mathbf{R}^3$ gives $\mathbf{R}^3$ a multiplicative structure, but it has zero divisors and there is no multiplicative identity and hence it makes no sense to talk about multiplicative inverses. Therefore let us strengthen our question.

Question 2. For which k does $\mathbf{R}^k$ admit a multiplicative structure that is a division algebra over $\mathbf{R}$?

It turns out that there is a very precise answer to this.

20.2.1 Theorem. A bilinear map

$$\mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}^n$$

without zero divisors exists if and only if n = 1, 2, 4, or 8. (We do not require associativity or a unit element here. If associativity is required, then n must be 1, 2, or 4.)

Proof. A proof of the fact that a division algebra over $\mathbf{R}$ has rank 2^k can be found in [Shaf94]. The hard part of the theorem, namely, the only if part, is proved in [BotM58]. The product for the real numbers, the complex numbers, the quaternions, and the octonions or Cayley numbers establishes the existence of the desired bilinear map for n = 1, 2, 4, and 8, respectively. The quaternions will be described in this chapter. We do not have time to describe the nonassociative algebra of Cayley numbers in this book, but the reader can find a definition in [Stee51] and a very extensive discussion of its properties and connections with other areas of mathematics in [Baez02]. There is an interesting discussion of the exceptional nature of some numbers in [Stil98].

The result of Theorem 20.2.1 is related to the question of how many linearly independent vector fields there are on S^{n-1} (see Section 8.5 in [AgoM05]), but the latter question is much harder however.

These introductory comments lead us to the subject of this chapter, quaternions, and the fact that $\mathbf{R}^4$ is a division algebra over the reals. The simplest way to show that is to write down the formula that defines the product of two 4-vectors.

Notation. The symbols $\mathbf{1}$, $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ will denote the standard basis (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1) of $\mathbf{R}^4$, respectively.

Definition. The (bilinear) product

$$\begin{aligned} \mathbf{R}^4 \times \mathbf{R}^4 &\to \mathbf{R}^4 \\ (\mathbf{a}, \mathbf{b}) &\to \mathbf{ab} \end{aligned}$$

with unit $\mathbf{1}$ defined by the equations

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -\mathbf{1}, \tag{20.1a}$$

$$\mathbf{ij} = \mathbf{k} = -\mathbf{ji},\ \mathbf{jk} = \mathbf{i} = -\mathbf{kj}, \quad \text{and} \quad \mathbf{ki} = \mathbf{j} = -\mathbf{ik} \tag{20.1b}$$

is called the *quaternion product* on $\mathbf{R}^4$.

Note. In this chapter **ab** denotes the quaternion product and **not** the segment from **a** to **b** as it does in the other parts of the book.

20.2.1 Proposition. Equations (20.1) in the definition of the quaternion product on $\mathbf{R}^4$ define a well-defined associative product.

Proof. Exercise 20.2.1.

Note that the product is not commutative since, for example, $\mathbf{ij} = -\mathbf{ji}$. Also, the identity $\mathbf{i}^2 = \mathbf{j}^2$ shows that $\mathbf{a}^2 = \mathbf{b}^2$ does not imply that $\mathbf{a} = \pm\mathbf{b}$. Condition (20.1b) could have been replaced by the single condition

$$\mathbf{ijk} = -1$$

and the assumption that we have an associative operation.

Definition. The vector space $\mathbf{R}^4$ together with the quaternion product is called the *quaternion algebra over* $\boldsymbol{R}$ and is denoted by $\mathbf{H}$.

For reasons which will become clearer as we go along, it is convenient to identify the subspace of $\mathbf{H}$ generated by $\mathbf{1}$ with $\mathbf{R}$ and the subspace of $\mathbf{H}$ generated by $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ with $\mathbf{R}^3$. In other words, we shall feel free to use expressions of the form

$$\mathrm{r} + \mathbf{v}, \quad \text{where } \mathrm{r} \in \mathbf{R} \text{ and } \mathbf{v} = (\mathrm{v}_1, \mathrm{v}_2, \mathrm{v}_3) \in \mathbf{R}^3,$$

to represent the quaternion

$$\mathrm{r}\mathbf{1} + \mathrm{v}_1\mathbf{i} + \mathrm{v}_2\mathbf{j} + \mathrm{v}_3\mathbf{k}.$$

With this identification we have expressed $\mathbf{H}$ as a direct sum of $\mathbf{R}$ and $\mathbf{R}^3$. By identifying

$$\mathrm{r} \quad \text{with} \quad \mathrm{r}\mathbf{1}$$

and

$$\mathrm{a} + \mathrm{b}\mathbf{i} \quad \text{with} \quad \mathrm{a}\mathbf{1} + \mathrm{b}\mathbf{i}$$

we shall consider both the reals and the complex numbers as being subsets of $\mathbf{H}$, that is, we have natural inclusions

$$\mathbf{R} \subset \mathbf{C} \subset \mathbf{H}.$$

Definition. Elements of $\mathbf{H}$ in the subspace generated by $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ are called *pure quaternions*. Let $\mathbf{q}$ be a quaternion and express $\mathbf{q}$ in the form $\mathrm{r} + \mathbf{v}$, where $\mathrm{r} \in \mathbf{R}$ and $\mathbf{v} \in \mathbf{R}^3$. The number r is called the *real part* of $\mathbf{q}$ and the vector $\mathbf{v}$ is called the *pure part* of $\mathbf{q}$. To extract these parts, we define functions

$$\mathrm{re}, \mathrm{pu} : \mathbf{H} \to \mathbf{H}$$

by

$$\mathrm{re}(\mathbf{q}) = \mathrm{r} \quad \text{and} \quad \mathrm{pu}(\mathbf{q}) = \mathbf{v}.$$

The direct sum decomposition of a quaternion into a real and pure part is the analog of the real and imaginary parts of a complex number. Next, we introduce a few standard definitions associated with quaternions and list some simple facts in Propositions 20.2.2–20.2.6. The proofs are left as exercises. They are trivial and simply involve expressing quaternions in terms of **i**, **j**, and **k** and then computing the appropriate expressions using the relevant definitions.

20.2.2 Proposition. A quaternion is real if and only if it commutes with every quaternion.

Proof. Exercise 20.2.2.

Definition. If $\mathbf{q} = \mathrm{r} + \mathrm{a}\mathbf{i} + \mathrm{b}\mathbf{j} + \mathrm{c}\mathbf{k}$ is a quaternion, then the *conjugate* of **q**, $\overline{\mathbf{q}}$, is defined by

$$\overline{\mathbf{q}} = \mathrm{re}(\mathbf{q}) - \mathrm{pu}(\mathbf{q}) = \mathrm{r} - \mathrm{a}\mathbf{i} - \mathrm{b}\mathbf{j} - \mathrm{c}\mathbf{k}.$$

The map $\bar{\ }: \mathbf{H} \to \mathbf{H}$, which sends **q** to $\overline{\mathbf{q}}$, is called the *conjugation map* of **H**. (Note that this map restricts to the usual conjugation map of the complex numbers.)

20.2.3 Proposition. Let $\mathbf{a}, \mathbf{b} \in \mathbf{H}$ and $\mathrm{r} \in \mathbf{R}$. The conjugation map of **H** has the following properties:

(1) $\overline{\mathbf{a} + \mathbf{b}} = \overline{\mathbf{a}} + \overline{\mathbf{b}}$
(2) $\overline{\mathrm{r}\mathbf{a}} = \mathrm{r}\overline{\mathbf{a}}$
(3) $\overline{\overline{\mathbf{a}}} = \mathbf{a}$
(4) $\overline{\mathbf{a}\mathbf{b}} = \overline{\mathbf{b}}\,\overline{\mathbf{a}}$
(5) $\mathbf{a} \in \mathbf{R}$ if and only if $\overline{\mathbf{a}} = \mathbf{a}$.
(6) $\mathbf{a} \in \mathbf{R}^3$ if and only if $\overline{\mathbf{a}} = -\mathbf{a}$.
(7) $\mathrm{re}\,(\mathbf{a}) = (\mathbf{a} + \overline{\mathbf{a}})/2$
(8) $\mathrm{pu}\,(\mathbf{a}) = (\mathbf{a} - \overline{\mathbf{a}})/2$
(9) If $\bullet$ is the dot product in $\mathbf{R}^4$, then $\mathbf{a} \bullet \mathbf{b} = \mathrm{re}\,(\overline{\mathbf{a}}\mathbf{b})$. In particular, $\mathbf{a} \bullet \mathbf{a} = \overline{\mathbf{a}}\mathbf{a}$.

Proof. This is straightforward and left as Exercise 20.2.3.

It follows from Proposition 20.2.3 (9) that $\overline{\mathbf{a}}\mathbf{a}$ is a nonnegative real. Note that the conjugate $\overline{\mathbf{a}}$ commutes with **a**, that is, $\overline{\mathbf{a}}\mathbf{a} = \mathbf{a}\overline{\mathbf{a}}$.

Definition. The *norm* or *absolute value* of a quaternion **a**, $|\mathbf{a}|$, is defined by

$$|\mathbf{a}| = \sqrt{\overline{\mathbf{a}}\mathbf{a}}.$$

It is easy to check that

$$|\mathrm{r} + \mathrm{v}_1\mathbf{i} + \mathrm{v}_2\mathbf{j} + \mathrm{v}_3\mathbf{k}|^2 = \mathrm{r}^2 + \mathrm{v}_1^2 + \mathrm{v}_2^2 + \mathrm{v}_3^2 \tag{20.2a}$$

and

$$|v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}|^2 = -(v_1^2 + v_2^2 + v_3^2). \tag{20.2b}$$

By equation (20.2a) the norm of a quaternion is just the standard length when it is thought of as a vector in $\mathbf{R}^4$, and

20.2.4 Proposition. Let $\mathbf{a},\mathbf{b} \in \mathbf{H}$.

(1) $|\mathbf{a}| = |\overline{\mathbf{a}}|$.
(2) $|\mathbf{ab}| = |\mathbf{a}|\,|\mathbf{b}|$.
(3) If $\mathbf{a} \neq \mathbf{0}$, then $\mathbf{a}^{-1}$ exists and $\mathbf{a}^{-1} = |\mathbf{a}|^{-2}\overline{\mathbf{a}}$. Furthermore, $|\mathbf{a}^{-1}| = |\mathbf{a}|^{-1}$.

Proof. Exercise 20.2.4.

Definition. A quaternion $\mathbf{a}$ is said to be a *unit quaternion* if $|\mathbf{a}| = 1$.

20.2.5 Proposition. The set of unit quaternions is just the unit sphere $\mathbf{S}^3$ in $\mathbf{R}^4$. It is a subgroup of $\mathbf{H}$.

Proof. Exercise 20.2.5.

Definition. Let $\mathbf{a}$ and $\mathbf{b}$ be pure quaternions. The *vector product* $\mathbf{a} \times \mathbf{b}$ of $\mathbf{a}$ and $\mathbf{b}$ is defined by

$$\mathbf{a} \times \mathbf{b} = \mathrm{pu}(\mathbf{ab}).$$

Since we have identified pure quaternions with $\mathbf{R}^3$, we can think of the function $\times$ as defining a product on $\mathbf{R}^3$. The next proposition justifies the notation since $\mathbf{a} \times \mathbf{b}$ agrees with the usual cross product. One way to look at this is to treat the new definition basically as an alternate algebraic definition for the cross product.

20.2.6 Proposition. Let $\mathbf{a}$ and $\mathbf{b}$ be pure quaternions.

(1) The map $\times$: $\mathbf{R}^3 \times \mathbf{R}^3 \to \mathbf{R}^3$ that sends $(\mathbf{a},\mathbf{b})$ to $\mathbf{a} \times \mathbf{b}$ is bilinear.
(2) $\mathbf{ab} = -\mathbf{a} \bullet \mathbf{b} + \mathbf{a} \times \mathbf{b}$
(3) $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$
(4) $\mathbf{a} \times \mathbf{a} = \mathbf{a} \bullet (\mathbf{a} \times \mathbf{b}) = \mathbf{b} \bullet (\mathbf{a} \times \mathbf{b}) = \mathbf{0}$
(5) In terms of our identification of pure quaternions with $\mathbf{R}^3$, $\mathbf{a} \times \mathbf{b}$ is just the ordinary cross product of $\mathbf{R}^3$.

Proof. This is again a straightforward computation that is left as Exercise 20.2.6.

Proposition 20.2.6 (2) leads to an alternate definition in closed form of the quaternionic product where one uses only basic vector operations:

20.2.7 Corollary. If $\mathbf{a} = r + \mathbf{v}$ and $\mathbf{b} = s + \mathbf{w}$ are two quaternions, where $r,s \in \mathbf{R}$ and $\mathbf{v},\mathbf{w} \in \mathbf{R}^3$, then

$$\mathbf{ab} = rs - \mathbf{v} \bullet \mathbf{w} + r\mathbf{w} + s\mathbf{v} + (\mathbf{v} \times \mathbf{w}).$$

20.3 Quaternions as Transformations

The previous section discussed the basic properties of quaternions. Now we come to the most important aspect of quaternions from the point of view of geometric modeling, namely, their close relationship to rotations in 3-space.

20.3.1 Proposition. If $\mathbf{q}$ is a nonreal unit quaternion, then $\mathbf{q}$ can be represented uniquely in the form

$$\mathbf{q} = \cos\theta + \sin\theta\,\mathbf{n}, \tag{20.3}$$

where $0 \le \theta \le \pi$ and $\mathbf{n}$ is a unit quaternion with $\mathbf{n}^2 = -1$.

Proof. Let $\mathbf{q} = r + \mathbf{v}$, $r \in \mathbf{R}$, $\mathbf{v} \in \mathbf{R}^3$. Since $\mathbf{q}$ has unit norm, $-1 \le r \le 1$ and so there is a unique θ in the stated range with $\cos\theta = r$. It follows from equation (20.2a) that

$$1 = |\mathbf{q}| = r^2 + |\mathbf{v}|^2,$$

or

$$|\mathbf{v}|^2 = 1 - r^2 = \sin^2\theta.$$

But $\sin\theta \ge 0$ for $0 \le \theta \le \pi$, so that $|\mathbf{v}| = \sin\theta$. Since $\mathbf{q}$ is not real, $\mathbf{v} \ne \mathbf{0}$ and $\sin\theta \ne 0$. Let

$$\mathbf{n} = \frac{1}{\sin\theta}\mathbf{v}.$$

Finally, equation (20.2b) implies that $\mathbf{n}^2 = -1$.

20.3.2 Corollary. Any quaternion $\mathbf{q}$ can be represented uniquely in the form

$$\mathbf{q} = r(\cos\theta + \sin\theta\,\mathbf{n}), \tag{20.4}$$

where $0 \le \theta \le \pi$ and $\mathbf{n}$ is a unit quaternion with $\mathbf{n}^2 = -1$.

Proof. This is obvious because r is just $|\mathbf{q}|$.

The representation (20.4) should remind us of the polar form representation of complex numbers.

Definition. The right hand side of equation (20.4) is called the *polar form* representation of the quaternion $\mathbf{q}$.

20.3.3 Example. To find the polar form of the quaternion $\mathbf{q} = 1 + \mathbf{i} + \mathbf{j} + \mathbf{k}$.

Solution. Since $|\mathbf{q}| = 2$ and $\cos\theta = 1/2$ implies that $\theta = \pi/3$, we get that

$$\mathbf{q} = 2\left(\frac{1}{2} + \frac{\sqrt{3}}{2}\mathbf{n}\right), \quad \text{where} \quad \mathbf{n} = \frac{2}{\sqrt{3}}(\mathbf{i} + \mathbf{j} + \mathbf{k}),$$

is the polar form of **q**.

We see that the polar form (20.4) of a quaternion is nothing startling but a useful rewrite that associates an angle with a quaternion. However, there are other similarities with the complex numbers. In fact, if we denote a **quaternion q** by the purely **formal** *exponential notation*

$$\mathbf{q} = r\, e^{\theta \mathbf{n}} \tag{20.5}$$

where r, $\theta \in \mathbf{R}$ and the pure quaternion **n** with $\mathbf{n}^2 = -1$ is defined by the polar form (20.4) for **q**, then

20.3.4 Proposition.

(1) $(re^{\theta\mathbf{n}})(se^{\eta\mathbf{n}}) = rs\, e^{(\theta+\eta)\mathbf{n}}$
(2) The inverse of $e^{\theta\mathbf{n}}$ is $e^{-\theta\mathbf{n}}$.
(3) The De Moivre Theorem holds, that is,

$$(\cos\theta + \sin\theta\, \mathbf{n})^m = \cos m\theta + (\sin m\theta)\mathbf{n}.$$

Proof. This is straightforward and left as Exercise 20.3.1.

Finally, from the polar form (20.4) we also see that the complex numbers can be imbedded in **H** in many ways, namely, we can use the imbedding

$$\begin{aligned} \mathbf{C} &\to \mathbf{H} \\ a + b\mathbf{i} &\to a + b\mathbf{n} \end{aligned}$$

20.3.5 Proposition. Let **q** be any nonzero quaternion and **z** any pure quaternion.

(1) $\mathbf{q}\,\mathbf{z}\,\mathbf{q}^{-1}$ is a pure quaternion.
(2) The map

$$\sigma_{\mathbf{q}} : \mathbf{R}^3 \to \mathbf{R}^3$$

defined by

$$\sigma_{\mathbf{q}}(\mathbf{z}) = \mathbf{q}\,\mathbf{z}\,\mathbf{q}^{-1}$$

is an isometry of $\mathbf{R}^3$. In fact, it is a rotation through an angle 2θ about the oriented line **L** through the origin with direction vector **n**, where

$$\mathbf{q} = |\mathbf{q}|(\cos\theta + \sin\theta\, \mathbf{n})$$

is the polar form of **q**.

(3) Let $\sigma_{\mathbf{q}}(\mathbf{z}) = \mathbf{z}\, M_{\mathbf{q}}$, where $M_{\mathbf{q}}$ is the 3×3 matrix that represents the linear transformation $\sigma_{\mathbf{q}}$. If

$$\mathbf{q} = r + a\mathbf{i} + b\mathbf{j} + c\mathbf{k} \quad \text{and} \quad |\mathbf{q}| = 1,$$

then

$$M_{\mathbf{q}} = \begin{pmatrix} 1-2b^2-2c^2 & 2rc+2ab & 2ac-2rb \\ 2ab-2rc & 1-2c^2-2a^2 & 2ra+2bc \\ 2rb+2ac & 2bc-2ra & 1-2a^2-2b^2 \end{pmatrix}.$$

Proof. To prove (1), note that

$$\mathbf{q}\,\mathbf{z}\,\mathbf{q}^{-1} = \frac{1}{\mathbf{q}\bullet\mathbf{q}}\,\mathbf{q}\,\mathbf{z}\,\overline{\mathbf{q}}$$

by Proposition 20.2.4(3). Repeated use of (7), (4), and (3) in Proposition 20.2.3 shows that

$$\mathbf{re}\,(\mathbf{q}\,\mathbf{z}\,\overline{\mathbf{q}}) = \frac{\mathbf{q}\mathbf{z}\overline{\mathbf{q}} + \mathbf{q}\overline{\mathbf{z}}\overline{\mathbf{q}}}{2} = \frac{\mathbf{q}(\mathbf{z}+\overline{\mathbf{z}})\overline{\mathbf{q}}}{2} = 0,$$

and (1) is proved.

The first part of (2) follows from the fact that

$$|\sigma_{\mathbf{q}}(\mathbf{z}_1) - \sigma_{\mathbf{q}}(\mathbf{z}_2)| = |\mathbf{q}(\mathbf{z}_1 - \mathbf{z}_2)\mathbf{q}^{-1}| = |\mathbf{q}||\mathbf{z}_1 - \mathbf{z}_2||\mathbf{q}^{-1}| = |\mathbf{z}_1\mathbf{z}_2|.$$

Next, observe that there is no loss in generality if we assume that $\mathbf{q}$ is a unit quaternion because the map $\sigma_{\mathbf{q}}$ stays the same if we replace $\mathbf{q}$ by any nonzero real multiple of $\mathbf{q}$. Therefore, let

$$\mathbf{q} = \cos\theta + \sin\theta\,\mathbf{n},$$

where θ and $\mathbf{n}$ are as described in Proposition 20.3.1.

Claim. $\sigma_{\mathbf{q}}$ fixes every point on the line $\mathbf{L}$.

To prove the Claim, observe that

$$\begin{aligned} \sigma_{\mathbf{q}}(\mathbf{n}) &= (\cos\theta + \sin\theta\,\mathbf{n})\,\mathbf{n}\,(\cos\theta - \sin\theta\,\mathbf{n}) \\ &= \cos^2\theta\,\mathbf{n} - \cos\theta\sin\theta\,\mathbf{n}^2 + \sin\theta\cos\theta\,\mathbf{n}^2 - \sin^2\theta\,\mathbf{n}^3 \\ &= (\cos^2\theta - \sin^2\theta\,\mathbf{n}^2)\mathbf{n} \\ &= \mathbf{n}, \end{aligned}$$

since $\mathbf{n}^2 = -1$. This clearly proves the Claim.

To prove the rest of (2), let

$$\mathbf{n} = n_1\mathbf{i} + n_2\mathbf{j} + n_3\mathbf{k},$$

and define

$$\mathbf{u} = -n_2\mathbf{i} + n_1\mathbf{j}.$$

Clearly, $\mathbf{n}$ is orthogonal to $\mathbf{u}$. We have

$$\sigma_{\mathbf{q}}(\mathbf{u}) = \cos^2\theta\,\mathbf{u} - \cos\theta\sin\theta\,\mathbf{u}\,\mathbf{n} + \sin\theta\cos\theta\,\mathbf{n}\,\mathbf{u} - \sin^2\theta\,\mathbf{n}\,\mathbf{u}\,\mathbf{n}. \tag{20.6}$$

But by Proposition 20.2.6(2)

$$\mathbf{n}\,\mathbf{u} = -\mathbf{n}\bullet\mathbf{u} + \mathbf{n}\times\mathbf{u} = \mathbf{n}\times\mathbf{u}$$

and

$$\mathbf{u}\,\mathbf{n} = -\mathbf{u}\bullet\mathbf{n} + \mathbf{u}\times\mathbf{n} = \mathbf{u}\times\mathbf{n}.$$

Therefore,

$$\mathbf{n}\,\mathbf{u} - \mathbf{u}\,\mathbf{n} = 2\mathbf{n}\times\mathbf{u}. \tag{20.7}$$

We also have

$$\mathbf{n}\,\mathbf{u}\,\mathbf{n} = -(\mathbf{n}\times\mathbf{u})\bullet\mathbf{n} + (\mathbf{n}\times\mathbf{u})\times\mathbf{n} = (\mathbf{n}\times\mathbf{u})\times\mathbf{n} = \mathbf{n} \tag{20.8}$$

(the last equality is Exercise 20.2.7). If we substitute equations (20.7) and (20.8) into equation (20.6), we get

$$\begin{aligned}\sigma_{\mathbf{q}}(\mathbf{u}) &= \left(\cos^2\theta - \sin^2\theta\right)\mathbf{u} + (2\sin\theta\cos\theta)\mathbf{n}\times\mathbf{u}\\ &= \cos 2\theta\,\mathbf{u} + \sin 2\theta\,\mathbf{n}\times\mathbf{u}.\end{aligned} \tag{20.9}$$

Since $\mathbf{u}$, $\mathbf{n}\times\mathbf{u}$, and $\mathbf{n}$ are an orthonormal basis for $\mathbf{R}^3$ and equation (20.9) is just the equation of a rotation in the plane through the origin with basis $\mathbf{u}$ and $\mathbf{n}\times\mathbf{u}$, $\sigma_{\mathbf{q}}$ must be the map we claimed it was.

Finally, to prove (3) note that the rows of $M_{\mathbf{q}}$ are just $\sigma_{\mathbf{q}}(\mathbf{i})$, $\sigma_{\mathbf{q}}(\mathbf{j})$, and $\sigma_{\mathbf{q}}(\mathbf{k})$. These values are easily computed and shown to be as indicated. This finishes the proof of the proposition.

Now the principal axis theorem (Theorem 2.5.5 in [AgoM05]) implies that every rotation in $\mathbf{R}^3$ that fixes the origin is a rotation about some line through the origin. Since that line has two unit direction vectors (one is the negative of the other), Proposition 20.3.5(2) implies the following converse:

20.3.6 Proposition. Every rotation R of $\mathbf{R}^3$ that fixes the origin is of the form $\sigma_{\mathbf{q}}$ for some non-zero quaternion $\mathbf{q}$. In fact, we may assume that $\mathbf{q}$ is a unit quaternion that is unique up to sign.

Proof. Let R be the rotation through an angle θ about the directed line through the origin with unit direction vector $\mathbf{n}$. If

$$\mathbf{q} = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\,\mathbf{n} \in \mathbf{H},$$

then $R = \sigma_{\mathbf{q}}$.

Proposition 20.3.5 defines a map

$$\begin{aligned} \rho : \mathbf{S}^3 &\to \mathbf{SO}(3), \\ \mathbf{q} &\to M_{\mathbf{q}} \end{aligned} \tag{20.10}$$

which is an important and well-known map in topology. This map is two-to-one because $\rho(\mathbf{q}) = \rho(-\mathbf{q})$. It is onto by Proposition 20.3.6. Using Proposition 20.3.4(1) we also see that the map is multiplicative because the product of two unit quaternions gets mapped to the composition of their associated rotations of $\mathbf{R}^3$. This, by the way, gives an indirect proof of the fact that the composition of two rotations about two lines is a rotation about another line. Finally, the two propositions show us how one can easily pass back and forth between the matrix representation of a rotation in $\mathbf{R}^3$ that fixes the origin and its representation as a unit quaternion.

20.3.7 Example. To find the formula for the rotation R about the z-axis through an angle of $\pi/2$ in terms of quaternions and to compute its action on $\mathbf{i}$.

Solution. We use the notation in Proposition 20.3.5. Now $\mathbf{n} = \mathbf{k}$ and $2\theta = \pi/2$. Therefore,

$$\mathbf{q} = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\mathbf{k},$$

$$\mathbf{q}^{-1} = \overline{\mathbf{q}} = \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\mathbf{k} \quad \text{(using the fact that } |\mathbf{q}| = 1 \text{ and Proposition 20.2.4(3))},$$

and

$$R(\mathbf{z}) = \mathbf{q}\,\mathbf{z}\,\mathbf{q}^{-1} = \frac{1}{2}(1+\mathbf{k})\mathbf{z}(1-\mathbf{k}).$$

It is easy to show that $R(\mathbf{i}) = \mathbf{j}$, which reestablishes the fact that R sends the x-axis to the y-axis. The matrix for R is

$$M_{\mathbf{q}} = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The fact that elements of the special orthogonal group $\mathbf{SO}(3)$ can be represented as quaternions is significant to computer graphics because quaternions make for a better representation for certain applications (see, for example, [Tayl79] and [YanF64]). One reason is that a quaternion takes less space, namely, one only needs to store four real numbers versus nine for a matrix. Another reason is that one needs

fewer arithmetic operations when making computations (see Table 2 in [Tayl79]). A third reason has to do with the fact that round-off errors cause numerical instability in matrices so that one needs to renormalize them periodically, which is not as much of a problem with the quaternions.

We finish this section with two useful conversion formulas. Although quaternions may have technical advantages, other representations such as Euler angles (defined in Section 2.5.1 in [AgoM05]) may provide a more intuitive way to describe rotations.

20.3.8 Proposition. If $[\alpha,\beta,\tau]$ is the Euler representation of a rotation R of $\mathbf{R}^3$ with center the origin, then $R(z) = \mathbf{q}\ \mathbf{z}\ \mathbf{q}^{-1}$, where $\mathbf{q}$ is the quaternion defined by

$$\mathbf{q} = r + a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$

and

$$\begin{aligned}
r &= \cos\frac{\tau}{2}\cos\frac{\beta}{2}\cos\frac{\alpha}{2} + \sin\frac{\tau}{2}\sin\frac{\beta}{2}\sin\frac{\alpha}{2}\\
a &= \cos\frac{\tau}{2}\cos\frac{\beta}{2}\sin\frac{\alpha}{2} - \sin\frac{\tau}{2}\sin\frac{\beta}{2}\cos\frac{\alpha}{2}\\
b &= \cos\frac{\tau}{2}\sin\frac{\beta}{2}\cos\frac{\alpha}{2} + \sin\frac{\tau}{2}\cos\frac{\beta}{2}\sin\frac{\alpha}{2}\\
c &= \sin\frac{\tau}{2}\cos\frac{\beta}{2}\cos\frac{\alpha}{2} - \cos\frac{\tau}{2}\sin\frac{\beta}{2}\sin\frac{\alpha}{2}.
\end{aligned}$$

Conversely, if a rotation R of $\mathbf{R}^3$ with center the origin is defined by means of a quaternion

$$\mathbf{q} = r + a\mathbf{i} + b\mathbf{j} + c\mathbf{k},$$

then the Euler representation $[\alpha, \beta, \tau]$ of R is defined by

$$\tan\alpha = \frac{m_{23}}{m_{33}}, \quad \tan\beta = -m_{13}, \quad \text{and} \quad \tan\tau = \frac{m_{12}}{m_{11}},$$

where

$$\begin{aligned}
m_{11} &= 2r^2 + 2a^2 - 1\\
m_{12} &= 2ab + 2rc\\
m_{13} &= 2ac - 2rb\\
m_{23} &= 2bc + 2ra\\
m_{33} &= 2r^2 + 2c^2 - 1.
\end{aligned}$$

Proof. See [Kuip99].

Kuiper ([Kuip99]) also shows that the quaternion representation is very useful in dealing with products of rotations and computing the rotation axis of the result.

Shoemake ([Shoe91]) discusses how quaternions can be expressed as 4×4 homogeneous matrices so that one can take advantage of fast matrix multiplication in hardware.

20.4 Exercises

Section 20.2

20.2.1 Prove Proposition 20.2.1.

20.2.2 Prove Proposition 20.2.2.

20.2.3 Prove Proposition 20.2.3.

20.2.4 Prove Proposition 20.2.4.

20.2.5 Prove Proposition 20.2.5.

20.2.6 Prove Proposition 20.2.6.

20.2.7 Prove that if $\mathbf{u}, \mathbf{v} \in \mathbf{R}^2$ are orthogonal unit vectors, them $(\mathbf{u} \times \mathbf{v}) \times \mathbf{u} = \mathbf{u}$.

Section 20.3

20.3.1 Prove Proposition 20.3.4.

20.3.2 Find a unit quaternion $\mathbf{q}$ that represents the rotation R of $\mathbf{R}^3$ about the origin with matrix

$$\mathrm{M} = \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{2}{\sqrt{6}} \end{pmatrix}.$$

CHAPTER 21

Digital Image Processing Topics

Prerequisites: Chapters 4 (integration) and Chapter 5 (metric spaces) in [AgoM05]

21.1 Introduction

This brief chapter looks at the topic of signal processing and its application to digital image processing. Complex numbers play an important role in this because of various transforms that get involved. On the other hand, except for the topic of antialiasing, digital image processing as a whole falls outside the scope of the geometric modeling topics central to this book. Our discussion of the Fourier transform and signal processing will be very limited. The only reason that we take up these topics at all is so that the reader can make some sense out of the various approaches to dealing with the antialiasing problem. The reader interested in learning more about the large field of digital image processing should consult [Glas95], [GonW87], or [RosK76], where additional references can be found. For the mathematics behind Fourier series and the Fourier transform see [Brac86], [Seel66], or [Apos58].

To motivate some of the mathematics we start, in Section 21.2, with the Laplace equation that historically has been the driving force behind a great many developments. Section 21.3 shows its connection to Fourier series. After defining the important L^p spaces in Section 21.4, we define the Fourier series for periodic functions in Section 21.5 and state a few of the important results about them. Sections 21.6 and 21.7 define the Fourier transform and convolution, respectively. For us, the main application of all these mathematics comes in Section 21.8. The reader who is not interested in the mathematics of this chapter should at least look at that section, especially Figures 21.9 and 21.10. These show the main points about the sampling problem behind aliasing.

Before getting started, however, we need to address an integration issue. Throughout this book all our integrals were implicitly assumed to be Riemann integrals, which is the integral that one learns about in calculus classes. There is another integral, the Lebesgue integral, which is more general and better for advanced mathematical topics for technical reasons. In particular it would be better for the Fourier transform and related topics because the Riemann integral would sometimes give incomplete and

unsatisfying results if we were to develop the subject in a thorough manner. Appendix D in [AgoM05] has a little bit to say about that integral and gives references where one can find more information. The hypotheses of some theorems in this chapter would be cleaner if we were to use the Lebesgue integral. We shall not do so, but needed to point this out for the mathematically minded reader because, as we have done throughout this book, we always want to state results carefully with all the correct hypotheses.

21.2 The Ubiquitous Laplace Equation

One of the most important differential equations in mathematics and science is the *Laplace equation*

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{21.1}$$

for a function u(x,y) of two variables. The solution of a great many problems lead to this equation or some variant of it. Mathematically, one is lead to this equation right from the start when studying analytic functions of a complex variable. If the function

$$f(z) = u(z) + \mathbf{i}v(z)$$

is an analytic function, then u(z) and v(z) satisfy the *Cauchy-Riemann equations*

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \quad \text{and} \quad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}.$$

(Recall that $\mathbf{C} = \mathbf{R}^2$ so that we can switch back and forth between thinking of a function as a function of a complex variable z or as a function of two real variables x and y.) Therefore, by taking partial derivatives of these two equations we get

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 v}{\partial x \partial y} \quad \text{and} \quad \frac{\partial^2 u}{\partial y^2} = -\frac{\partial^2 v}{\partial y \partial x}.$$

Since an analytic function is infinitely differentiable, one can show that the mixed partials are equal, which leads immediately to the Laplace equation. A similar argument shows that the function v also satisfies the Laplace equation.

Definition. Any function u(x,y) of two real variables that has continuous partials up to order two that satisfies the Laplace equation is called a *harmonic function*.

Harmonic functions satisfy a maximum principle like analytic functions, which we should mention in passing.

21.2.1 Theorem. (The Maximum Principle) A harmonic function defined on a closed and bounded set assumes its maximum and minimum value on the boundary of this set.

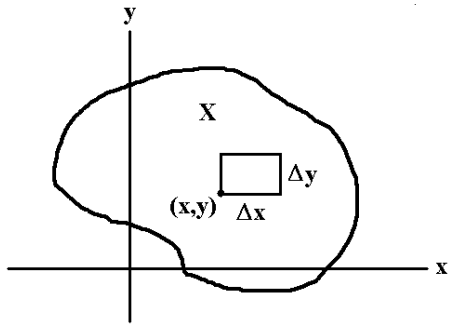


Figure 21.1. Analyzing a steady temperature system.

We have explained the mathematical basis for the importance of the Laplace equation, now we want to sketch solutions to some practical problems and show how they also lead to this equation. In a sense, classical field theory of physics is a study of the solutions to the equation. (Actually, we have only stated the two-dimensional Laplace equation and we should include the three-dimensional version for the previous sentence to hold.) The reader will have to look up in books on physics certain physical laws to which we refer in the discussion if they are unfamiliar with them.

Steady Temperature. Assume the temperature of a body **X** is defined by a function T(x,y). Basically, we are assuming that the temperature is the same on all planes parallel to the xy-plane, so that we have a two-dimensional problem. The rate at which heat crosses a curve is proportional to the integral of the normal derivative of T along the curve. Let us see what happens in a small rectangle of width Δx and height Δy. See Figure 21.1. The rate of flow of heat to the right through the left edge of the rectangle is approximately

$$-K\Delta y\frac{\partial T}{\partial x},$$

where K is a thermal conductivity constant associated to the solid. Using the derivative of this function we can estimate the loss of heat as we pass from the left edge to the right edge of the rectangle by

$$-K\Delta y\frac{\partial^2 T}{\partial x^2}\Delta x.$$

Similarly, we can estimate the loss of heat as we pass from the bottom edge to the top edge of the rectangle by

$$-K\Delta x\frac{\partial^2 T}{\partial y^2}\Delta y.$$

Since we are assuming a steady state, the losses must sum to 0, and we get the equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0. \tag{21.2}$$

The typical problem is then to solve this equation given some boundary conditions, that is, the value of T(x,y) on the boundary of the body **X**.

Electric Potential. Coulomb's law states that the force **F** between two point charges located at $\mathbf{p}_1$ and $\mathbf{p}_2$ with charge q_1 and q_2, respectively, is defined by the equation

$$\mathbf{F} = \frac{q_1 q_2}{4\pi\varepsilon_0 r^2}\mathbf{u},$$

where

$r = |\mathbf{p}_1\mathbf{p}_2|$ is the distance between the two charges,
$\mathbf{u} = (1/r)\ \mathbf{p}_1\mathbf{p}_2$ is the unit direction vector from $\mathbf{p}_1$ to $\mathbf{p}_2$, and
ε_0 is a constant.

When using the centimeter/gram/second unit system

$$4\pi\varepsilon_0 = 1.$$

Now the electric field strength **E** at a point of an electric field due to a charge distribution is the force exerted on a unit of positive charge placed at that point. It follows that **E** is related to the force **F** on a charge q at a point by the equation

$$\mathbf{F} = q\mathbf{E}.$$

The Gauss flux law implies that

$$\nabla\mathbf{E} = 0 \tag{21.3}$$

in empty space. Finally, the potential V(**p**) at a point **p** in space is defined as the work per unit charge necessary to bring a small test charge from some arbitrary reference point to **p**. One can show that

$$\mathbf{E} = -\nabla V. \tag{21.4}$$

Equations (21.3) and (21.4) imply that any potential function V(x,y) derived from some electrostatic distribution must satisfy the Laplace equation in empty space.

Two-dimensional Fluid Flow. Consider a two-dimensional fluid flow in $\mathbf{R}^3$, meaning that the velocity of the fluid is the same in every plane parallel to the xy-plane. We can represent the velocity of such a flow by a function

$$v(x,y) = (v_1(x,y), v_2(x,y)).$$

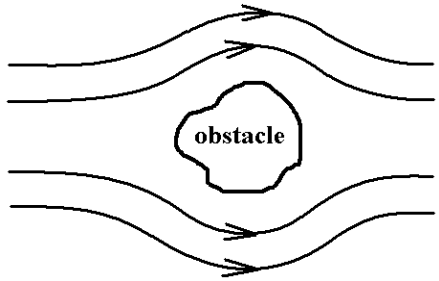


Figure 21.2. Fluid flowing around an obstacle.

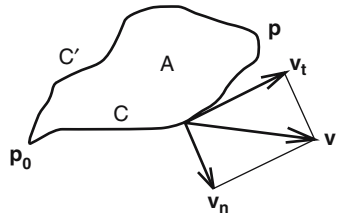


Figure 21.3. Irrotational fluids have well-defined circulations along curves.

Fluid flowing around an infinitely high cylindrical obstacle is an example. See Figure 21.2. One calls the flow *stationary* or *steady* if the velocity of the fluid depends only on the position (x,y) and not on time. Given a curve **C** in the plane, the integral

$$\int_{\mathbf{C}} v_t$$

is called the *circulation* of the fluid along the curve **C**, where v_t is the tangential component of v along **C**. Assume that the fluid is *irrotational* or *circulation free*, that is, its circulation is zero along any closed curve. In that case, one can show that for any fixed point $\mathbf{p}_0$, the line integral along a curve **C** from $\mathbf{p}_0$ to a point **p** depends only on **p** and not the curve **C**, so that there is a well-defined function

$$\phi(\mathbf{p}) = \int_{\mathbf{C}} v_t$$

where **p** is the endpoint of **C**. It follows that $v = \nabla\phi$. See Figure 21.3. Assume next that the fluid is incompressible and nonviscous. (A fluid is *incompressible* if it has constant density, which means mathematically that the integral

$$\int_{\mathbf{C}} v_n$$

along any closed curve **C** is zero, where v_n is the normal component of v along **C**, that is, the amount of fluid leaving the region bounded by **C** equals the amount entering the region. It is *nonviscous* if there is no internal friction, so that pressure forces on a surface are perpendicular to the surface.) With these assumptions, we are mathematically in a situation like in the steady temperature problem, except that our functions have other names and interpretations. It follows that the function ϕ satisfies the Laplace equation.

21.3 From Laplace to Fourier

Before defining Fourier series, the reader unfamiliar with the subject may find some motivation helpful. To motivate some of the central ideas we start with the Laplace equation.

One way to solve the Laplace equation is by expressing it in polar coordinates. If we do this, equation (21.1) becomes

$$\frac{\partial\left(r\frac{\partial u}{\partial r}\right)}{\partial r}+-\frac{1}{r}\frac{\partial^2 u}{\partial\theta^2}=0,\quad r\neq 0. \tag{21.5}$$

For simplicity assume that

$$\text{the function } u(r,\theta) \text{ is continuous for } 0\leq r\leq 1. \tag{21.6}$$

Note that $u(r,\theta + 2\pi) = u(r,\theta)$. These conditions are not enough to define u. Assume that u satisfies the boundary condition

$$u(1,\theta)=f(\theta) \tag{21.7a}$$

for some periodic function $f(\theta)$ with

$$f(\theta+2\pi)=f(\theta). \tag{21.7b}$$

The problem of finding a function $u(r,\theta)$ that satisfies (21.5)–(21.7) is called the *Dirichlet problem*. One attempt to solve this problem involves using the "method of separation of variables" and to look for a solution of the form

$$u(r,\theta)=R(r)H(\theta).$$

This reduces the problem to solving two **ordinary** differential equation that have solutions of the form

$$\begin{aligned} u_n(r,\theta) &= aA, && n=0,\\ &= \left(ar^n\right)\left(Ae^{in\theta}+Be^{-in\theta}\right), && n>0, \end{aligned}$$

for some constants a, A, and B. Since any linear combination of solutions is also a solution, one is led to suppose the more general solution

$$u(r,\theta)=\sum_{n=-\infty}^{\infty} a_n r^{|n|}e^{in\theta} \tag{21.8}$$

with boundary condition

$$u(1,\theta) = \sum_{n=-\infty}^{\infty} a_n e^{in\theta}. \tag{21.9}$$

The amazing thing is that all of this sort of works. It led Fourier to what are now called Fourier series expansions for functions. Although we may be tempted to conclude that we can express an arbitrary function $f(\theta)$ in the form (21.9), this is **not** quite correct. The problem is with the convergence of the infinite series. Theorems 21.5.3 and 21.5.7 will shortly carefully state what **is** correct.

This concludes our bird's eye view of what led up to Fourier series. See [Seel66] for more details. There is one more topic to be dealt with before we are ready to deal with the subject of Fourier series itself.

21.4 The L^p Function Spaces

In this section we introduce some function spaces and operators that play an important role in functional analysis. We will also encounter some common terminology that is worth knowing. However, the real motivation for getting into this topic is that this chapter deals with operators that map functions to other functions and it is very useful to have some notation for describing the domains of these operators and also to learn about some of the properties of these function spaces. For simplicity we restrict our discussion to functions of one variable, but similar definitions and facts hold for functions of more variables.

Definition. Let [a,b] be an interval in **R**, either bounded or unbounded, and let $1 \le p < \infty$. Define $L^p([a,b])$, the *L^p space* of functions on [a,b], as follows:

$L^1([a,b])$ is the set of integrable functions on [a,b] .
$L^p([a,b])$, $1 < p < \infty$, is the set of functions f on [a,b] whose integral $\int_a^b |f|^p$ converges.

Functions in $L^p([a,b])$ are called *L^p functions*.

Note 1. The definition of L^p space assumed Riemann integration. If we had used the Lebesgue integral, then we would not have had to make a special definition for $p = 1$. Unfortunately, integrability does not imply absolute integrability in the case of the Riemann integral, ergo the special case.

Note 2. We do not have to restrict ourselves to real-valued functions. Everything we say about $L^p([a,b])$ would hold for complex-valued functions **if** $p > 1$. We could introduce notation to distinguish between the two cases, but in what we will be doing here it does not seem worthwhile to do so.

The function spaces $L^p([a,b])$ have a great many interesting properties. Proofs of the facts we list below can, for example, be found in [Nata61].

Fact 1. $L^p([a, b])$ is an infinite dimensional vector space.

Definition. The *L^p norm* on $L^p([a,b])$, denoted by $\| \|_p$, is defined by

$$\| f \|_p = \left(\int_a^b |f|^p \right)^{1/p}.$$

Fact 2. The L^p norm satisfies

(1) $\| f \|_p \geq 0$.
(2) $\| f \|_p = 0$ if and only if $\int_a^b f = 0$.
(3) For every constant c, $\| cf \|_p = |c| \, \| f \|_p$.
(4) $\| f+g \|_p \leq \| f \|_p + \| g \|_p$.

In other words, the norm acts very much like the absolute value function on reals. We can use it to define a distance function.

Definition. If $f,g \in L^p([a,b])$, define the *L^p distance between f and g*, denoted by $d_p(f,g)$, by

$$d_p(f,g) = \| f - g \|_p$$

The properties listed under Fact 2 above show that $d_p(f,g)$ is almost a metric on $L^p([a,b])$, but not quite. It is only a pseudometric. It is symmetric and satisfies the triangle inequality, but it is possible to have the distance between two functions be 0 without the functions being equal. If the functions were continuous, then this would not happen, but we shall see later when we discuss the Fourier transform that the assumption of continuity would be too restrictive. Therefore, let us use the standard trick to get a metric from a pseudometric. Consider the equivalence relation ~ on the space $L^p([a,b])$, where

$$f \sim g \quad \text{if} \quad \int_a^b (f-g) = 0.$$

The function $d_p(f,g)$ would induce a metric on the set of equivalence classes (see Exercise 5.2.10 in [AgoM05]). With this equivalence relation we are saying that any function whose integral over [a,b] is zero is treated as if it were identical to the zero function. In fact, this identification of functions with the corresponding equivalence class is **always** made. In the future a statement such as "f = g" for two functions f and g in $L^p([a,b])$ technically means that f ~ g. Because one does not want to introduce new notation, the reader needs to remember that, although we refer to elements of $L^p([a,b])$ as functions, technically

$L^p([a,b])$ **is really considered to be a set of equivalence classes of functions!**

It follows from what has just been said that it is legitimate to call $d_p(f,g)$ a metric on $L^p([a,b])$. It is called the *L^p metric*.

Fact 3. $(L^p([a,b]),d_p)$ is a complete metric space.

We concentrate now on the space $L^2([a,b])$, also called *Hilbert space* or the space of *square integrable functions*, which is especially interesting because it admits an inner product.

Definition. Let $f,g \in L^2([a,b])$. Define the *inner product of f and g*, denoted by $\langle f,g\rangle$, by

$$< f,g > = \int_a^b f(t)\overline{g(t)}dt.$$

Since this notation depends on the interval [a,b], we shall write $\langle,\rangle_{[a,b]}$ if there is any confusion.

Fact 4. $\langle f,g\rangle$ is an inner product on the vector space $L^2([a,b])$ and the usual length and distance function associated to an inner product agree with the norm $\| \|_2$ and distance function d_2 as defined above.

There is also an $L^\infty([a,b])$ space of function but its definition is special and so we treat it separately. Because it involves some often seen terminology, it is worth defining here. The general definition would apply to "measurable" functions and everything would be defined up to a set of measure 0, but too avoid technicalities with measure that have not been satisfactorily dealt with in this book, we shall restrict out definitions to continuous functions.

Definition. Let [a,b] be an interval in **R**, either bounded or unbounded. Define the *L^∞ space*, $L^\infty([a,b])$, to be the set of bounded continuous functions on [a,b]. Its elements are called *L^∞ functions*. The *L^∞ norm* on $L^\infty([a,b])$, denoted by $\| \|_\infty$, is defined by

$$\|f\|_\infty = \sup_{x\in[a,b]} \{|f(x)|\}.$$

If $f,g \in L^\infty([a,b])$, define the *L^∞ distance between f and g*, denoted by $d_\infty(f,g)$, by

$$d_\infty(f,g) = \|f-g\|_\infty .$$

With these definitions everything we said earlier about the L^p spaces holds here. The function $\| \|_\infty$ satisfies the four metric properties listed in Fact 2 above, the function $d_\infty(f,g)$ defines a metric on the equivalence classes of functions with respect to the relation ~ , where

$$f \sim g \quad \text{if and only if} \quad \|f-g\|_\infty = 0,$$

and the resulting metric space is complete.

21.5 Fourier Series

This section presents the relevant definitions and some basic theorems. Proofs are omitted.

We start with some additional motivation. Recall that every element **v** of a vector space can be expressed as a linear combination of basis vectors $\mathbf{v}_j$, that is,

$$\mathbf{v} = \sum_{j=1}^{n} a_j \mathbf{v}_j.$$

In general, to determine the coefficients a_j one would have to solve some linear equations, but if the $\mathbf{v}_j$ are an orthonormal basis, then

$$\mathbf{v} \bullet \mathbf{v}_k = \sum_{j=1}^{n} a_j (\mathbf{v}_j \bullet \mathbf{v}_k) = a_k.$$

If the $\mathbf{v}_j$ are only an orthogonal basis, then

$$a_k = \frac{\mathbf{v} \bullet \mathbf{v}_k}{\mathbf{v}_k \bullet \mathbf{v}_k}.$$

In the case of infinite dimensional vector spaces we would get infinite sums, so that one has to be a little careful here. However, one can make sense out of such sums by introducing the notion of convergence. For example, polynomials of degree n such as

$$p(x) = \sum_{j=0}^{n} a_j x^j$$

form an n-dimensional vector space with basis the polynomials x^j. As n goes to infinity, one can ask which functions can be represented by (convergent) power series.

The trigonometric functions sin nt , cos nt , and e^{int} are periodic L^2 functions of period 2π. They are also orthogonal sets of functions, but not of unit length (with respect to the L^2 inner product).

21.5.1 Theorem. Let I be any interval of length 2π.

(1) $\langle \sin mt, \sin nt \rangle = 0$ if $m \neq n$,
$= \pi$ if $m = n$.
(2) $\langle \cos mt, \cos nt \rangle = 0$ if $m \neq n$,
$= \pi$ if $m = n$.
(3) $\langle \sin mt, \cos nt \rangle = 0$.
(4) $\langle e^{imt}, e^{int} \rangle = 0$ if $m \neq n$,
$= 2\pi$ if $m = n$.

Proof. See [Spie69].

Therefore, **if** one could represent a function f(t) by a series in the form

$$\sum_{n=-\infty}^{\infty} a_n e^{\mathbf{i}nt}, \tag{21.10}$$

then, assuming one can do the integration on a term-by-term basis, one could, using the above argument, solve for a_n, namely,

$$a_n = \frac{1}{2\pi} < f, e^{\mathbf{i}nt} >_{[-\pi,\pi]} = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)e^{-\mathbf{i}nt}dt. \tag{21.11}$$

In any case,

Definition. The a_n defined by (21.11) are called the *Fourier coefficients* of f and the series (21.10) is called the *Fourier series* for f (with respect to the functions e^{int}).

The Fourier coefficient a_n intuitively says how much of the function e^{int} appears in f.

Since

$$e^{\mathbf{i}nt} = \cos nt + \mathbf{i}\sin nt,$$

the Fourier series for real functions is easily shown to be expressible in the form

$$\frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos nt + b_n \sin nt), \tag{21.12}$$

where

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos nt\,dt \quad \text{and} \quad b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin nt\,dt. \tag{21.13}$$

21.5.2 Example. Consider the function $f(x) = x^2$ on the interval $(0,2\pi)$ and think of it as a periodic function defined on the reals as shown in Figure 21.4. Then, carrying out the relevant integrations using standard formulas for trigonometric integrals, the Fourier series for f is found to be

$$\frac{4\pi^2}{3} + \sum_{n=1}^{\infty}\left[\frac{4}{n^2}\cos nx - \frac{4\pi}{n}\sin nx\right].$$

See [Spie69] for this and other examples.

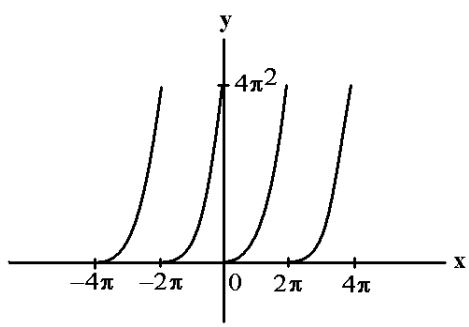


Figure 21.4. The function f(x) of Exercise 21.5.2.

The basic question here is whether the Fourier series for a function f actually converges to f. Substituting the formula for the a_n into equation (21.8) we can interchange the integration with the summation when $r < 1$, since the series converges absolutely there, and so we get

$$u(r,\theta) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \sum_{n=-\infty}^{\infty} r^{|n|} e^{in(\theta-t)} f(t)dt. \tag{21.14}$$

But the standard formula for geometric series gives us

$$\sum_{n=0}^{\infty} r^n e^{in(\theta-t)} = \frac{1}{1-re^{i(\theta-t)}}$$

and

$$\sum_{n=-1}^{-\infty} r^{-n} e^{in(\theta-t)} = \frac{re^{-i(\theta-t)}}{1-re^{-i(\theta-t)}},$$

so that (21.14) becomes

$$u(r,\theta) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \frac{1-r^2}{1-2r\cos(\theta-t)+r^2} f(t)dt \quad \text{for} \quad r<1. \tag{21.15}$$

Definition. The function

$$P(r,\phi) = \frac{1}{2\pi}\frac{1-r^2}{1-2r\cos\phi+r^2}$$

is called the *Poisson kernel*.

We are finally ready to give an answer to the Dirichlet problem that we have been studying and it is Fourier series that provide that answer.

21.5.3 Theorem.

(1) (Existence) If $f(\theta)$ is a continuous periodic function of period 2π and if $u(r,\theta)$ is the function defined by equation (21.14), then

$$\lim_{r\to 1} u(r,\theta) = f(\theta),$$

and the convergence is uniform in θ.

(2) (Uniqueness) Let $f(\theta)$ be a continuous periodic function of period 2π. If $v(r,\theta)$ is a function satisfying equations (21.5)–(21.7) and which converges uniformly in θ to $f(\theta)$ as r increases to 1, then v is the function $u(r,\theta)$ defined by equation (21.15).

Proof. See [Seel66].

Now that we have seen the usefulness of Fourier series in the context of a specific application, we look at the general question of their existence and uniqueness. Fourier series do not converge to any arbitrary function.

Definition. Let [a,b] be a finite interval. A function f defined on [a,b] is said to be of *bounded variation* on [a,b] if there is a positive constant M so that

$$\sum_{i=1}^{n} |f(x_i) - f(x_{i-1})| \le M$$

for all partitions $a = x_0 < x_1 < \ldots < x_n = b$ of [a,b]. If f is of bounded variation, then the *total variation* $V_f(a,b)$ is defined by

$$V_f(a,b) = \sup\left\{\sum_{i=1}^{n} |f(x_i) - f(x_{i-1})|\right\},$$

where the supremum is taken over the sums associated to all possible partitions of [a,b].

Here are some basic facts related to functions of bounded variation on finite intervals.

21.5.4 Theorem. Let f be a function defined on a finite interval [a,b].

(1) If f is monotonic, then the set of points where it is discontinuous is countable.
(2) If f is monotonic, then it is of bounded variation.
(3) If f is continuous and f′ exists and is bounded on (a,b), then f is of bounded variation. It follows that if f has a continuous derivative on [a,b], then f is of bounded variation.
(4) The function f is of bounded variation if and only if it can be expressed as the difference of two increasing functions. Furthermore, if f is continuous, then f is of bounded variation if and only if it can be expressed as the difference of two increasing continuous functions.

Proof. See [Apos58]

21.5.5 Example. The function

$$f(x) = x\cos(1/x), \quad 0 < x \le 1,$$
$$f(0) = 0$$

is not of bounded variation on [0,1]. See Figure 21.5. We start with the graph of the function cos(1/x), which wiggles "infinitely often" near 0. This function is not continuous at 0. Multiplying by x decreases the magnitude of the wiggles (but not their number) and turns it into a continuous function because

$$f(x) \to 0 \quad \text{as} \quad x \to 0.$$

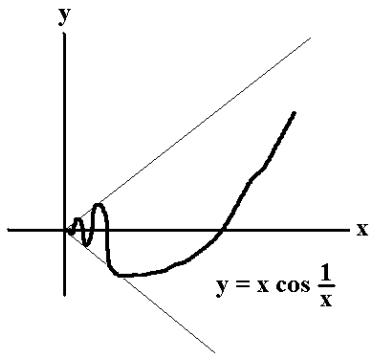


Figure 21.5. A function that is not of bounded variation.

With these preliminaries out of the way, we can get to the main theorems about Fourier series.

21.5.6 Theorem.

(1) If f(t) is a periodic function of period 2π, which is absolutely integrable and of bounded variation on $[-\pi,\pi]$, then its Fourier series (21.10) converges to

$$\frac{f(t^+)+f(t^-)}{2}$$

for each t. ($f(t^\pm)$ refers to one-sided limits of f at t.) If the function is also continuous, then the Fourier series converges to f(t) for each t.

(2) If two continuous periodic functions f and g of period 2π have the same Fourier coefficients, then f = g.

Proof. See [Apos58] and [Seel66].

Theorem 21.5.6 deals with how Fourier series converge **pointwise** to the periodic functions that defined them. However, we can also ask about convergence in the L^2 norm. The other basic question is then answered by the following theorem:

21.5.7 Theorem. Let f be an arbitrary (not necessarily periodic) function in $L^2([-\pi,\pi])$. Then the Fourier series (21.10) converge in the L^2 metric and is equal to f. (Recall that equality here means that the integral of the difference is zero.)

Proof. See [Nata61]. The fact that f is not periodic is not a problem. Since we are only interested in what happens in the interval $[-\pi,\pi]$ we can forget whatever definition it might have outside that interval and, as Example 21.5.2 showed, we can make it into a periodic function on **R** of period 2π with the formula $f(t + 2k\pi) = f(t)$, where $k = 0, \pm1, \pm2, \ldots$, and $t \in [-\pi,\pi]$.

Note that Theorem 21.5.7 does **not** say that the Fourier series of f converges pointwise to f. It may seem strange that one would be interested in non-pointwise convergence, but in fact L^2 convergence is a useful type of convergence.

Another variant of this theorem is

21.5.8 Theorem. If $f \in L^2([-\pi,\pi])$ and if $\langle f, e^{int}\rangle = 0$ for all $n \neq 0$, then $f = 0$.

Proof. See [Nata61].

Rather than using the functions e^{int} one often chooses normalized functions for a basis for $L^2([-\pi,\pi])$, such as, for example, the functions $e^{2\pi int}$. Orthonormal bases are, after all, the nicest types of bases. Additionally, one does not have to restrict oneself to these exponential type functions. Other variations of these functions are often used as an orthonormal basis. The specific choice of basis is dictated by what is most convenient for a particular application.

This is as far as we shall take the subject of Fourier series here.

21.6 The Fourier Transform

Fourier series have to do with representing functions (periodic ones to be precise) as series. The Fourier transform, on the other hand, attempts to represent functions as integrals. One advantage Fourier integrals have is that they can represent fairly arbitrary functions, not just periodic ones. Since the domain of functions may now be all of **R**, the standard trick for turning a function into a periodic one would not work.

Our discussion of the Fourier transform will concentrate on functions of one variable and only briefly mention the two-variable case.

Definition. If f, G: $\mathbf{R} \to \mathbf{R}$, then

$$F(u) = \int_{-\infty}^{\infty} f(x)e^{-2\pi \mathbf{i} ux}dx \tag{21.16}$$

is called the *Fourier transform of f(x)* and

$$g(x) = \int_{-\infty}^{\infty} G(u)e^{2\pi \mathbf{i} ux}du$$

is called the *inverse Fourier transform of G*. The functions F and g will be denoted by FT(f) and $FT^{-1}(G)$, respectively.

Note 1. There is no uniform agreement in the literature as to what is called the Fourier transform, although all the various definitions have the form

$$F(u) = a\int_{-\infty}^{\infty} f(x)e^{\mathbf{i}bux}dx$$

for suitable constants a and b.

Note 2. The formula for the Fourier transform should remind the reader of the definition of the Fourier coefficients of a function. The similarity is not accidental. Actually we could have also used the Laplace equation to motivate the definition. See [Seel66]. In this case, rather than the region being a disk, we would be dealing with a halfplane that has a function specified along its edge. If we were to work through

this type of Dirichlet problem as we did for the disk, we would end up with a solution that looked like the Fourier transform. Notice, however, one difference between the Fourier coefficient formula and the Fourier transform: the former had exponential functions e^{inx} where n was an integer, but now the "n" is allowed to be an arbitrary real number.

The obvious first question about the Fourier transform is for which functions f does it and its inverse exist?

21.6.1 Theorem. If f is absolutely integrable, then its Fourier transform F exists and is continuous.

Proof. The theorem follows from the fact that

$$|F(u) - F(v)| \le \int_{-\infty}^{\infty} |f(x)| \left| e^{2\pi i u x} - e^{2\pi i v x} \right| dx \le \left(\int_{-\infty}^{\infty} |f| \right) |u - v|.$$

Figure 21.6 shows the Fourier transforms of two functions. The function is on the left and its Fourier transform on the right. The functions in the figure have the following definitions:

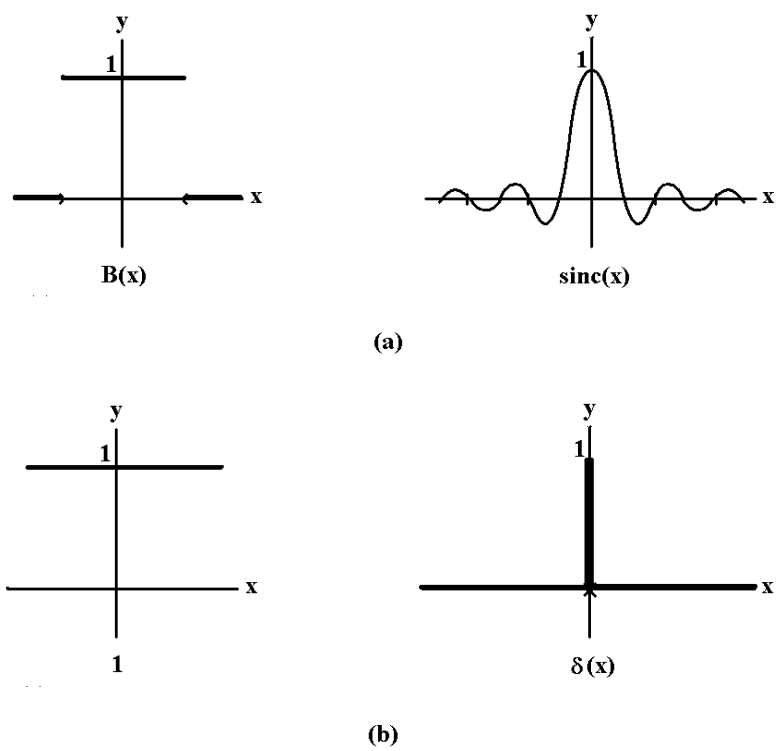


Figure 21.6. One-dimensional Fourier transforms.

The box function B(x): $B(x) = 0, \quad |x| > \frac{1}{2},$
$= 1, \quad |x| \le \frac{1}{2}.$

The function sinc(x): $\operatorname{sinc} x = 1, \quad x = 0,$
$= \frac{\sin \pi x}{\pi x}, \quad x \ne 0.$

The Gaussian function: $f(x) = \exp\left(\frac{-x^2}{\sigma^2}\right)$. (With this notation σ becomes the *standard deviation* and σ^2 is the *variance* of f(x).

The Dirac delta "function" δ(x): This is not really a function in the mathematical sense. It is usually described by the strange-looking conditions

$$\delta(x) = 0, \quad x \ne 0,$$

and

$$\int_{-\infty}^{\infty} \delta(x)dx = 1.$$

As stated this last integral expression is really nonsense. A precise mathematical definition (see [Frie63]) defines δ(x) as a *generalized function,* which is a linear functional on an appropriate space of functions. On the other hand, the actual definition is not as important as its properties, the most important of which is

$$\int_{-\infty}^{\infty} f(x)\delta(x - x_0)dx = f(x_0). \tag{21.17}$$

An intuitive discussion of generalized functions can be found in [Brac86], where they are explained in terms of limits of functions. The Dirac delta function is often called the *impulse signal* or *impulse function* in Fourier analysis.

Now, we are after more than just the Fourier transform. We want what we called the inverse Fourier transform to really be the inverse of the Fourier transform. In order to state that theorem and its hypotheses, we need to extend the notion of a function being of bounded variation to functions defined on unbounded intervals.

Definition. Let **I** be an unbounded interval of the form **R**, $[a,\infty)$, or $(-\infty,b]$. A function f defined on **I** is said to be of *bounded variation* on **I** if it is of bounded variation on every finite subinterval and there is a positive constant M so that $V_f(c,d) < M$ for every finite subinterval [c,d] of **I**.

The next theorem states the most relevant extensions of Theorem 21.5.4.

21.6.2 Theorem. Let f be a function defined on **R**.

(1) If f is bounded and monotonic, then it is of bounded variation.

(2) The function f is bounded and of bounded variation if and only if it can be expressed as the difference of two bounded increasing functions.

Proof. See [Apos58].

Theorem 21.6.2(1) implies that the following increasing function is of bounded variation:

The Heaviside unit step function H(x): $H(x) = 0, \quad x < 0,$
$= 1, \quad x \geq 0.$

Since the Heaviside function is of bounded variation, so is the box function B(x) since it differs from

$$H\left(x+\frac{1}{2}\right)-H\left(x-\frac{1}{2}\right)$$

at the single point $x = \frac{1}{2}$.

Here is the theorem we were after.

21.6.3 Theorem. Let f: $\mathbf{R} \to \mathbf{R}$ be an absolutely integrable function of bounded variation, then

$$\frac{f(x^+)+f(x^-)}{2} = \int_{-\infty}^{\infty}\left(\int_{-\infty}^{\infty} f(t)e^{2\pi i u(x-t)}dt\right)du. \tag{21.18}$$

If f is also continuous, then the left hand side of equation (21.18) equals f(x), that is, $f = FT^{-1}(FT(f))$.

Proof. See [Widd71] or [Apos58].

Theorem 21.6.3 is the fundamental theorem in the theory of Fourier transforms. It says that if one thinks of the Fourier transform as a mapping of functions to functions, then the inverse Fourier transform is basically the inverse of that mapping (if one ignores the finite number of points of discontinuity that might exist). For example, in Figure 21.4 the functions on the left are the inverse Fourier transform of the functions on the right. There are different variants of Theorem 21.6.3. The problem is that the hypotheses of the theorem are not satisfied by all the functions one might want to deal with. For example, the sinc(x) function is not absolutely integrable. We have defined what sometimes is called the "ordinary" Fourier transform. It is possible to generalize the Fourier transform by defining it in terms of a convergent sequence of integrals, but we shall not pursue this further here. At the end of the day, however, all the functions that one wants to use are covered.

Using the different names "x" and "u" for the variables of a function and for its Fourier transform, respectively, was a conscious decision. We shall do so throughout the rest of this chapter. One considers functions f(x) as defined on a "spatial" domain

and its Fourier transform on a "frequency" domain. Comparing equations (21.11) and (21.16) we see that if f(x) were expressed as a Fourier series, then F(u) picks up the coefficient of an appropriate periodic term of the series and intuitively expresses the contribution of that term to the values of f. The Fourier transform and its inverse take us back and forth between these domains.

Moving on to functions of two variables, assuming that integrals over unbounded regions have been defined, we have

Definition. If f, G: $\mathbf{R}^2 \to \mathbf{R}$, then

$$F(u,v) = \int_{\mathbf{R}^2} f(x,y)e^{-2\pi \mathbf{i}(ux+vy)}dxdy \tag{21.19}$$

is called the *Fourier transform of f(x,y)* and

$$g(x,y) = \int_{\mathbf{R}^2} G(u,v)e^{2\pi \mathbf{i}(ux+vy)}dudv \tag{21.20}$$

is called the *inverse Fourier transform of G(u,v)*. Like in the one-dimensional case, the functions F and g will be denoted by FT(f) and $FT^{-1}(G)$, respectively.

Sufficient conditions for a function f(x,y) to have a Fourier or inverse Fourier transform are similar to the one variable case. The case of "separable" functions simplifies the analysis somewhat. A function f(x,y) is called *separable* if

$$f(x,y) = f_1(x)f_2(y)$$

for some functions $f_i(x)$. In that case, the integral in equation (21.19) can be separated into two one-dimensional integrations and the proofs in that case can be used to show that the integral in equation (21.19) exists.

The box function B(x,y): $B(x,y) = 0,\ |x|,\ |y| > \frac{1}{2},$

$= 1,\ |x|,\ |y| \le \frac{1}{2}.$

The function sinc(x,y): $\text{sinc}(x,y) = \text{sinc}(x)\ \text{sinc}(y)$

Figure 21.7 shows the Fourier transform of the function B(x,y). It is just sinc(u,v).

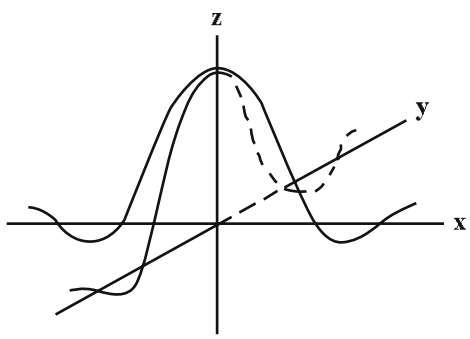


Figure 21.7. The Fourier transform of the box function B(x,y).

There are discrete versions of the Fourier transform and its inverse. We give the one-dimensional version as an example.

Definition. If f(k) and G(u) are functions defined on the integers, then

$$F(u) = \frac{1}{N}\sum_{k=0}^{N-1} f(k)\exp\left(\frac{-2\pi \mathbf{i} uk}{N}\right), \quad \text{where} \quad 0 \le u \le N-1,$$

is called the *discrete Fourier transform of f* and

$$g(k) = \sum_{k=0}^{N-1} G(u)\exp\left(\frac{2\pi \mathbf{i} uk}{N}\right), \quad \text{where} \quad 0 \le k \le N-1,$$

is called the *discrete inverse Fourier transform of G*.

The Fourier transform is computationally very expensive to implement. Therefore, it was quite a breakthrough when an efficient way to implement it, called the Fast Fourier Transform (FFT), was published in 1965 in the paper [CooT65]. Thanks to the FFT, many signal processing algorithms are now practical.

A real nice interpretation of the discrete Fourier transform can be found in [Glas99]. It can be thought of as expressing an arbitrary polygon as a sum of basic regular polygons.

21.7 Convolution

Definition. Given two functions f, g: $\mathbf{R} \to \mathbf{R}$, define their *convolution*, $f*g$, by

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt.$$

We give some conditions for when the convolution exists.

21.7.1 Theorem. If f, g: $\mathbf{R} \to \mathbf{R}$, then the convolution integral

$$\int_{-\infty}^{\infty} f(t)g(x-t)dt$$

converges absolutely for all x under either of the following two conditions:

(1) The function f is absolutely integrable and g is bounded.
(2) Both f and g are absolutely integrable and both belong to $L^2(\mathbf{R})$.

If both f and g are also continuous, then the convolution integral is also continuous under either condition (1) or (2).

Proof. See [Seel66] or [Apos58].

21.7.2 Example. Figure 21.8 shows the convolution of the two functions

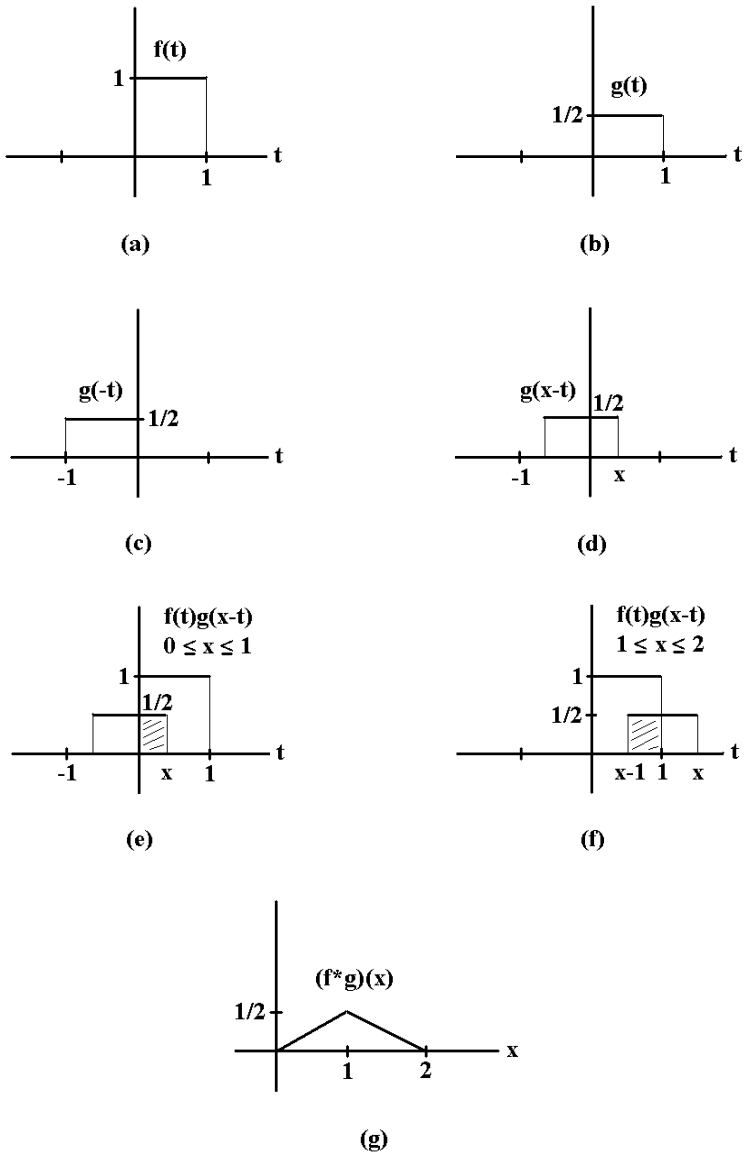


Figure 21.8. An example of convolution.

$$f(t) = 1, \quad t \in [0,1] \qquad\qquad g(t) = \frac{1}{2}, \quad t \in [0,1]$$
$$= 0, \quad t \notin [0,1] \qquad\qquad = 0, \quad t \notin [0,1]$$

The graphs of the functions f(t) and g(t) are shown in Figures 21.8(a) and (b), respectively. Figures 21.8(c) and (d) show the stages of computing g(x–t). The shaded regions in Figure 21.8(e) and (f) show the areas computed by the convolution integral that

define (f∗g)(x). Although f and g were not continuous functions, we see that f∗g is. Its formula is

$$\begin{aligned}(f*g)(x) &= \frac{1}{2}x, && x \in [0,1]\\ &= 1-\frac{1}{2}x, && x \in [1,2]\\ &= 0, && \text{elsewhere.}\end{aligned}$$

Two easily checked properties of convolution are:

(1) (commutativity) f∗g = g∗f
(2) (linearity) f∗(g + h) = f∗g + f∗h

21.7.3 Theorem. (The Convolution Theorem) Let f, g: $\mathbf{R} \to \mathbf{R}$ be two absolutely integrable functions. Then

(1) FT (f∗g) = FT (f) · FT (g).
(2) If additionally either the Fourier transform of g converges or both f and g belong to $L^2(\mathbf{R})$, then FT (f · g) = FT (f)∗FT (g).

Proof. See [Apos58] and [Seel66]. For weaker conditions on f and g, see [Widd71].

What Theorem 21.7.3 says is that multiplying functions in the spatial domain corresponds to doing a convolution in the frequency domain and vice versa.

Although we will not elaborate about this here, similar results hold in two dimensions and also for the discrete case.

21.8 Signal Processing Topics

We are ready to apply what we have learned so far. Image enhancement or reconstruction are two of the major applications of the Fourier transform. This leads us to the basic problem in sampling theory, which is to determine how many samples one must take so that no information is lost. Recall our discussion in Section 2.6.

Definition. A function f(x) whose Fourier transform F(u) vanishes outside a finite interval is called a *band-limited* function. If F(u) = 0 for |u| > w, where

$$w = \inf\{c \mid F(u) = 0 \quad \text{for} \quad |u| > c\},$$

then w is called the *cutoff frequency* of f(x).

Figure 21.9 shows a key idea behind reconstructing functions. The functions are shown in the left column and their Fourier transforms in the right column. In (a) we have the signal f(x) that we are trying to analyze. As we can see from its Fourier transform F(u) on the right, it is a band-limited function that vanishes outside the interval [−w,w]. The sampling function s(x) in (c) is applied to (a) to get (e). What you see in (e) is all that we know about the real function in (a). It is the function sampled at intervals of width d. The question is whether we can reconstruct (a) from (e). If the Fourier transform of (e) is as shown in (f), then we cannot. It is periodic with period

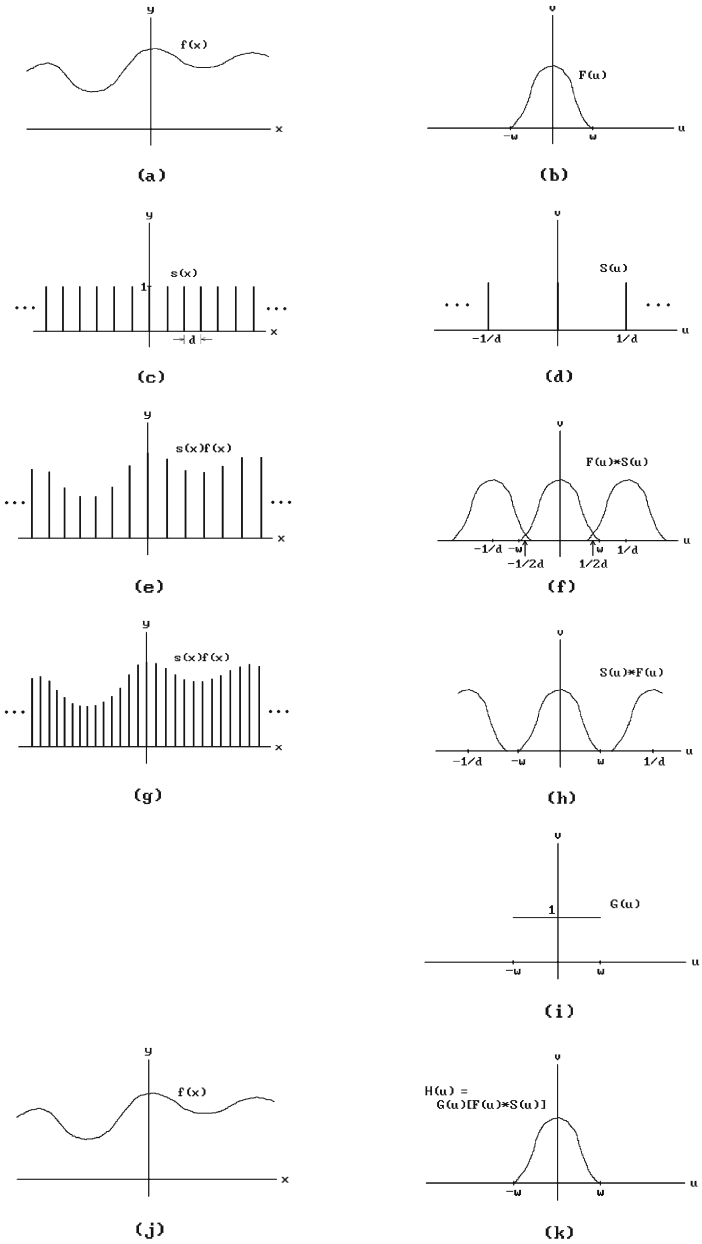


Figure 21.9. Reconstructing functions.

1/d and the repetitions of F(u) overlap. However, if we had sampled more often and used (g) where we have a smaller sampling width d, then we can get (a) back in two steps as follows:

Step 1. Multiply the Fourier transform F(u)∗S(u) of f(x)s(x) by the box function

$$G(u) = B\left(\frac{u}{2w}\right)$$

to get the function H(u) in (k) which has isolated one copy of F(u).

Step 2. Take the inverse Fourier transform of H(u) to recreate f(x) as shown in (j).

It follows that if we have a band-limited function, then we can recover a function by sampling sufficiently often. This is the Whittaker-Shannon sampling theorem.

21.8.1 Theorem. (Whittaker-Shannon Sampling Theorem) Let f(x) be a band-limited function with cutoff frequency w, so that its Fourier transform vanishes outside the interval [−w,w]. Then f(x) can be reconstructed exactly from samples at intervals d provided that the sampling frequency 1/d is at least 2w.

Proof. See [Glas95].

Definition. The frequency w in Theorem 21.8.1 is called the *Nyquist frequency* of f(x).

If the sampling frequency is not high enough, then we get the phenomena called aliasing. The two-dimensional situation is similar, but in practice one must sample a lot more because of limitations of available reconstruction algorithms. Unfortunately, in real life we usually do not have such band-limited functions, so that all this is of theoretical value only. Nevertheless, it does show us what is going on and helps us determine ways that one can mitigate the sampling problems. Actually, there is one additional complication to the reconstruction of signals. An important implicit assumption in our analysis so far has been that we took an infinite number of samples. This is of course an unrealistic assumption. We therefore need to study the problem further and ask what would be different if we only take a finite number of samples.

See Figure 21.10. In (a) we again show a function that is about to be sampled and reconstructed. The mathematical effect of sampling only a finite number of times is to multiply the function in (e) by a box-like function. Suppose that we only sample over the interval [0,a]. Let

$$h(x) = B\left(\frac{x}{a} - \frac{1}{2}\right).$$

This "windowing" function vanishes outside of [0,a]. See (g). Its transform H(u) is shown in (h). Our actual samples, represented by the function h(x)[s(x)f(x)], is shown in (i) and its Fourier transform H(u)∗[S(u)∗F(u)] in (j). Since H(u) has components that extend to infinity, this has the effect of introducing a distortion in the frequency domain representation of the function that was sampled over the finite interval. The unfortunate conclusion is that it is in general impossible to faithfully reconstruct a

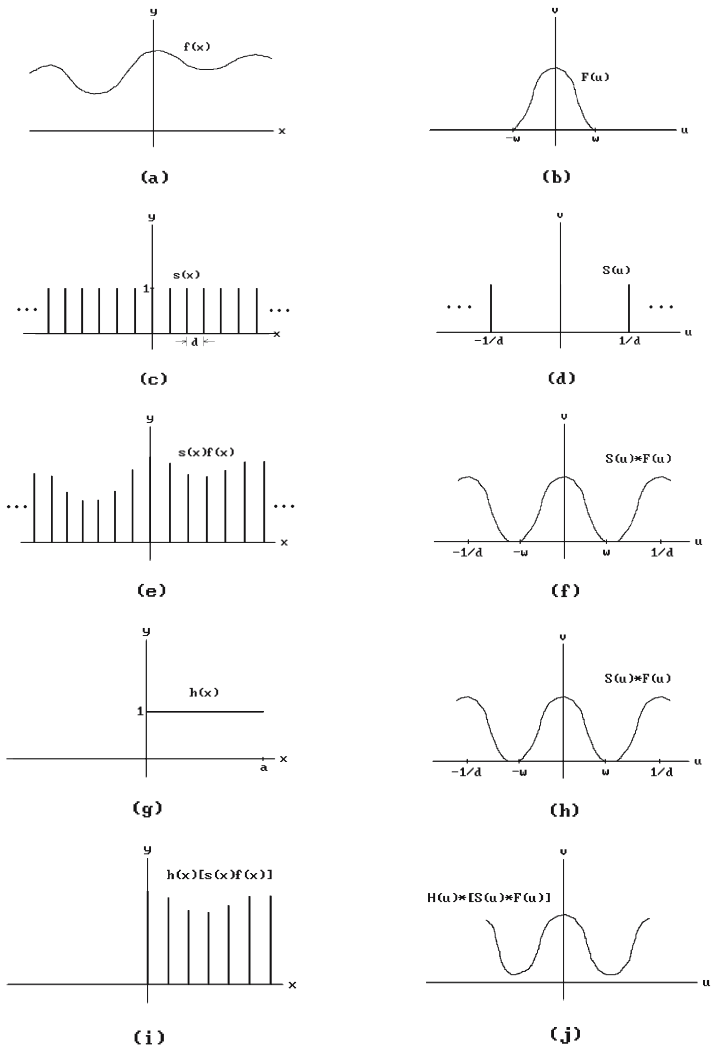


Figure 21.10. Reconstruction of finitely-sampled functions.

function that has only been sampled over a finite range. No function that is nonzero over only a finite interval can be band-limited and, conversely, any band-limited function is nonzero over an unbounded domain.

To summarize, in the context of computer graphics, what typically happens is that we start with a signal (the signal derived from a scene that we are trying to display) and then this signal is passed through a sequence of filters before it ends up on the display. (A *filter* means simply anything that modifies a signal.) Multiplying a function by another, such as a box function, is an example of filtering. We can apply a filter either in spatial or frequency domain. The latter is common. Some standard types of filters are

Low-pass filter: A filter that cuts off all frequencies above a threshold and lets those below it pass through.

High-pass filter: This is the opposite of a low-pass filter. A filter that cuts off all frequencies below a threshold and lets those above it pass through.

Band-pass filter: A filter that cuts off all frequencies outside of a range of frequencies.

We have come across these filters in our earlier discussion. We can use Figure 21.9 to help us understand a typical definition-to-display path of a signal. The original signal is shown in (a). Some equally spaced pulses and the sampled signal are shown in (c) and (e), respectively. The assumption is that we can get our hands on only the sampled signal. (If we really "had" the original signal, then none of this work we are describing would be necessary.) At this point, our goal is to reconstruct the original signal. To accomplish this one chooses a suitable "reconstruction" filter and convolves it with the sampled signal. Next, our goal is to display the resulting signal S. The display may have some finitary constraints, such as a monitor that can only show something at a finite number of pixels. For this reason, one typically uses another filter, perhaps a low-pass filter, to eliminate frequencies that one cannot display. One convolves the signal S with such a filter. Finally, this signal is sampled and reconstructed with another "reconstruction" filter to produce our displayed signal. For a good discussion of this process see [Glas95].

21.9 Wavelets

Digital image processing has two components. One deals with the representation or analysis of a function. That is what the Fourier transform was about. It represented a function in terms of their frequency content. The other part of digital image processing is concerned with putting it all back together again and reconstructing a function from its inherent frequencies. So the Fourier transform computed the Fourier coefficients, which gave us the representation or analysis of the function, and then the Fourier series reconstructed the function using these Fourier coefficients. One problem with the Fourier transform though is that it is based on the sine and cosine functions that do **not** have compact support. That is why, if we want to represent a function that **has** compact support, its representation in the frequency domain will not also be a function with compact support but involve an infinite number of frequencies. We saw that in the case of the box function and the impulse function. This is where wavelets come in.

This section will only give a brief glimpse of wavelets. We shall make no attempt to give a formal definition of what they are but simply discuss them in the context of examples. Wavelets are functions that can have compact support. By analyzing functions using wavelets, one can get a better handle on those functions with compact support. Additionally, they give one the means of specifying the amount of detail one wants from the analysis based on the number of samples taken. There is no unique set of wavelet basis functions. We briefly describe the Haar basis and some of its properties. For simplicity we restrict ourselves to analyzing functions defined on [0,1].

Define functions ϕ, $\phi_{j,i}$, Ψ , $\Psi_{j,i}$: $\mathbf{R} \to \mathbf{R}$ by

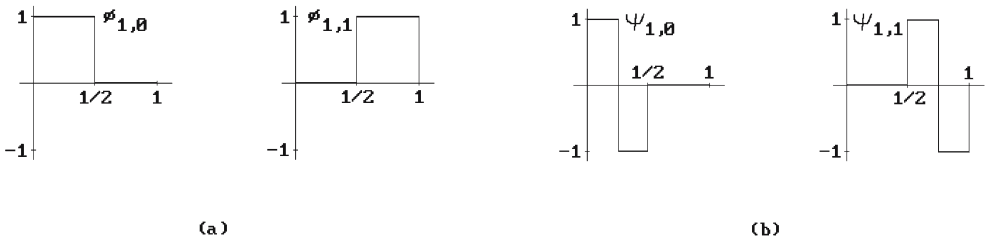


Figure 21.11. Box and wavelet basis functions.

$$\begin{aligned} \phi(x) &= 1 && \text{for} \quad 0 \le x < 1, \\ &= 0 && \text{elsewhere}, \\ \phi_{j,i}(x) &= \phi(2^j x - i) && \text{for} \quad i = 0,1,\ldots,2^j - 1, \end{aligned}$$

and

$$\begin{aligned} \psi(x) &= 1 && \text{for} \quad 0 \le x < \frac{1}{2}, \\ &= -1 && \text{for} \quad \frac{1}{2} \le x < 1, \\ &= 0 && \text{elsewhere}, \\ \psi_{j,i}(x) &= \psi(2^j x - i) && \text{for} \quad i = 0,1,\ldots,2^j - 1. \end{aligned}$$

See Figure 21.11. Let $V^j \subset L^2(\mathbf{R})$ be the vector subspace with basis $(\phi_{j,i}\}_i$. Clearly,

$$V^0 \subset V^1 \subset V^2 \subset \ldots.$$

If W^j is the orthogonal complement of V^j in V^{j+1}, then

$$V^{j+1} = V^j \oplus W^j$$

and the $\{\Psi_{j,i}\}_i$ are an orthogonal basis of W^j. It follows that

$$V^{j+1} = V^0 \oplus W^0 \oplus W^1 \oplus \ldots \oplus W^j.$$

Definition. The functions $\phi_{j,i}(x)$ are called *scaling functions*. Each function $\Psi_{j,i}(x)$ is called a *Haar wavelet* and for each fixed j, the collection of these wavelets is called the *one-dimensional Haar wavelet basis functions*.

Now the functions $\phi_{j,i}$ and $\Psi_{j,i}$ are not unit vectors in $L^2(\mathbf{R})$. To make them unit vectors one must multiply them by $2^{j/2}$. We did not do this here is because it would cause some ugly coefficients in Example 21.9.1 below and obscure what is going on. Nevertheless, this is normally done because orthonormal bases are desirable, so that the terms "scaling function" and "wavelet" usually refer to the normalized versions of the functions we are using here.

The next example shows how wavelets are used to represent functions up to a desired resolution.

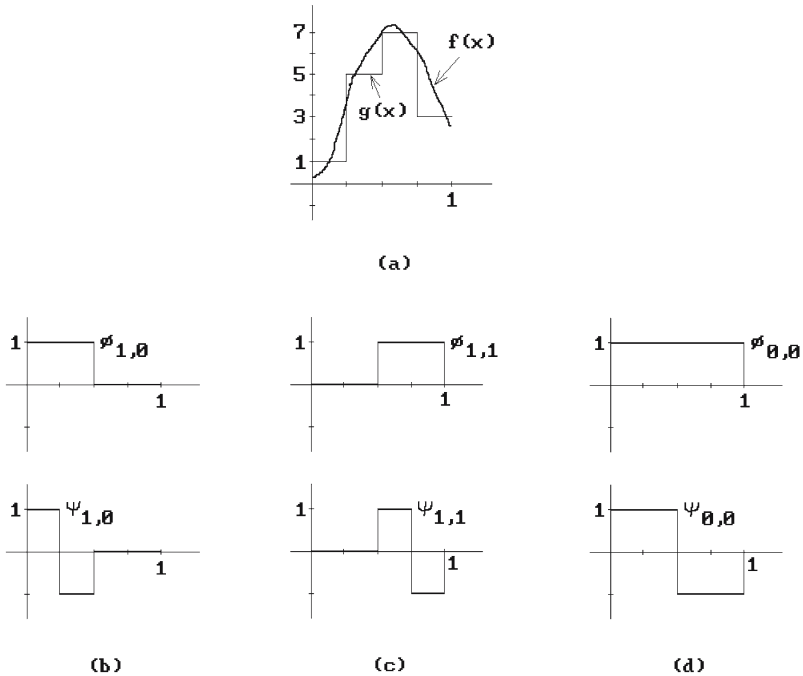


Figure 21.12. Approximating with wavelets.

21.9.1 Example. Consider the function f(x) in Figure 21.12(a). If the function was sampled at four evenly spaced points with values 1, 5, 7, and 3, then its approximation would be the step function

$$g(x) = 1\phi_{2,0}(x) + 5\phi_{2,1}(x) + 7\phi_{2,2}(x) + 3\phi_{2,3}(x). \tag{21.21}$$

We show how this representation can be obtained with wavelets.

Step 1. The first two sample values average to 3. To get the actual values back from this difference we have to add –2 and +2, respectively. To put this another way,

$$1\phi_{2,0}(x) + 5\phi_{2,1}(x) = 3\phi_{1,0}(x) - 2\psi_{1,0}(x), \tag{21.22}$$

as is easily checked. See Figure 21.12(b). Similarly, the second two sample values average to 5. To get the actual values back from this difference we have to add +2 and –2, respectively. Again, this means that

$$7\phi_{2,2}(x) + 3\phi_{2,3}(x) = 5\phi_{1,1}(x) + 2\psi_{1,1}(x). \tag{21.23}$$

See Figure 21.12(c). Substituting (21.22) and (21.23) into (21.21) shows that

$$g(x) = 3\phi_{1,0}(x) + 5\phi_{1,1}(x) - 2\psi_{1,0}(x) + 2\psi_{1,1}(x). \tag{21.24}$$

Step 2. We repeat this process with the two averages 3 and 5. Their average is 4 and to get the actual values back we have to add +1 and −1, respectively. In other words,

$$3u_{1,0}(x) + 5u_{1,1}(x) = 4u_{0,0}(x) - 1w_{0,0}(x). \tag{21.25}$$

See Figure 21.12(d). Substituting (21.25) into (21.24) finally gives

$$g(x) = 4\phi_{0,0}(x) - 1\psi_{0,0}(x) - 2\psi_{1,0}(x) + 2\psi_{1,1}(x). \tag{21.26}$$

This is the wavelet representation of g(x) that we were looking for.

Example 21.9.1 really describes how one can find the wavelet approximation to sampled functions in general by successively taking averages of adjacent values and specifying the differences between the actual and average values. This is the basis for a compression algorithm since the differences, called *detail coefficients*, are smaller numbers than the actual ones and so it takes fewer bits to represent the sampled values of the function. For example, the sequence 4, −1, −2, +2 completely represents the samples 1, 5, 7, 3 if one understands it: From 4 and −1 one forms 3 and 5, then from the sequence 3, 5, −2, +2, one gets 1, 5, 7, 3.

Now in the definitions of $\phi_{j,i}$ and $\Psi_{j,i}$ above we had some constraints on the range of i and j, but that was only because we were temporarily interested in functions defined on [0,1]. The definitions actually make sense for all integers i and j and we need to allow this if we want to analyze functions defined on **R**. The following holds:

21.9.2 Theorem. The functions $2^{j/2}\,\Psi_{j,i}$, $i, j \in \mathbf{Z}$, form an orthonormal basis for $L^2(\mathbf{R})$.

Proof. See [GomV98].

One extremely nice feature of wavelets is that they allow a multi-resolution analysis of functions, which means that one can specify the degree of detail that one wants to capture. This should be clear from the above. To get higher resolutions, we simply increase the j above so that we are approximating with smaller-width step functions. See [GomV98] or [Glas95] for a definition of what is meant by a multi-resolution representation. For multi-resolution analysis on arbitrary surfaces, not just $\mathbf{R}^2$, see [LoDW97].

The Haar wavelet basis and the associated spaces V^j are only one possible choice. We can use smoother functions. For a continuous basis one can use hat functions or, more generally, one can use higher-order B-splines for more smoothness. See [Four95]. They would play the role of the $\phi_{j,i}$ above. Aside from the particular wavelet basis, the basic constructions would stay the same. One does lose orthogonality, however. Once one has a basis, then the analysis of a function in $L^2(\mathbf{R})$ is simply to take the orthogonal projection of the function onto the space spanned by the wavelet basis.

Definition. Given an orthonormal wavelet basis $\Psi_{j,i}$ of functions in $L^2(\mathbf{R})$, the *discrete wavelet transform* WT is defined on a function f in $L^2(\mathbf{R})$ by

$$WT(f)(i,j) = \langle w_{j,i}, f\rangle = \int_{-\infty}^{\infty} f(x)\psi_{j,i}(x)dx.$$

Theorem 21.9.2 implies that if the discrete Haar wavelet transform (computing the coefficients of a function with respect to the Haar basis functions) is known, then one can reconstruct the function.

Finally, with regard to higher dimensions, in dimension two we can take the "tensor product" of one-dimensional wavelets to get the obvious extension. Wavelets are a very active field of research currently with many applications in a great variety of fields, in particular, computer graphics.

21.10 Exercises

Section 21.5

21.5.1 Verify that the following Fourier series are correct on $(-\pi,\pi)$.

(a) $x = 2\sum_{n=1}^{\infty} \frac{(-1)^{n-1}\sin nx}{n}$

(b) $x^2 = \frac{\pi^2}{3} + 4\sum_{n=1}^{\infty} \frac{(-1)^n \cos nx}{n^2}$

(c) $x\cos x = -\frac{1}{2}\sin x + 2\sum_{n=2}^{\infty} \frac{(-1)^n n \sin nx}{n^2 - 1}$

CHAPTER 22

Chaos and Fractals

Prerequisites: Chapter 5 and Chapter 7 (for Theorem 22.3.1) in [AgoM05]

22.1 Introduction

Physics, and science in general, is filled with the study of *dynamical systems*, that is, processes that evolve with time and where future states are determined by past states. The evolution of the universe and the motion of the planets around the sun are two examples of dynamical systems. Describing what happens in such systems, even ones that are defined in rather simple ways, is often extremely difficult. What is the final state of the universe? What orbits will the planets eventually assume? Answering such questions involves a lot of topology but is also a natural context in which to discuss chaos and fractals. Fractals are the de facto way to model natural phenomena in computer graphics (see Section 5.4). The interconnection between all these topics is what this chapter is all about. We can only provide a brief outline of some highlights and the reader who wants to learn more will need to consult the references.

Section 22.2 defines dynamical systems and some basic terminology associated to them. We also explain when such systems are said to be chaotic. In Section 22.3 we define what is meant by the dimension of a topological space and discuss the connection of this concept with that of a fractal. Finally, Section 22.4 is on iterated function systems on which many of the common algorithms for generating fractals are based.

22.2 Dynamical Systems and Chaos

Mathematically, the study of dynamical systems reduces to

a space $\mathbf{X}$ (the states),
a map f: $\mathbf{X} \to \mathbf{X}$ (f corresponds to the laws which control a process),

an initial point $\mathbf{x}_0$ in $\mathbf{X}$ (the start state), and
an analysis of the sequence of points $\mathbf{x}_n$, n = 1,2, . . . , defined by

$$\mathbf{x}_{n+1} = f(\mathbf{x}_n). \tag{22.1}$$

This section will look at some aspects of such dynamical systesm.

Consider a map f: $\mathbf{X} \to \mathbf{X}$.

Definition. The *k-fold composite* or *k-fold iterate* of f, f^k: $\mathbf{X} \to \mathbf{X}$, is defined recursively by

$$f^0(\mathbf{x}) = \mathbf{x} \quad \text{and} \quad f^k(\mathbf{x}) = f\left(f^{k-1}(\mathbf{x})\right) \quad \text{for} \quad k > 0.$$

Note that with this notation the sequence defined by equation (22.1) can also be defined by

$$\mathbf{x}_n = f^n(\mathbf{x}_0).$$

Definition. The set $\{\mathbf{x}, f^1(\mathbf{x}), \ldots, f^n(\mathbf{x}), \ldots\}$ of iterates of the point $\mathbf{x}$ is called the *orbit* of $\mathbf{x}$ with respect to f. A point $\mathbf{x}$ is said to be *periodic* of *period k* if $\mathbf{x} = f^k(\mathbf{x})$.

The kinds of questions to which we want answers are

Question 1. Do the points $\mathbf{x}_n$ above converge to some $\mathbf{x}$?
Question 2. Does the sequence of $\mathbf{x}_n$s become cyclic?
Question 3. If one perturbs the initial point $\mathbf{x}_0$ a little to a point $\mathbf{x}_0'$, then how different are the orbits of these two points?

The answer to Question 3 is particularly important to computer science because of the inherent problem with round-off errors. If small changes to initial conditions for a problem lead to large changes in the solution to the problem, then one would have to be very skeptical of any solutions to the problem obtained with the help of a computer. Unstable problems would require much extra care. An example of an unstable system is the case of a pendulum that is pointed straight up. The pendulum will stay in that position indefinitely (assuming that no force acts on it), but if it is moved ever so slightly to either side, then it will immediately fall to that side and start to swing wildly back and forth until it eventually settles on the bottom. Users who are asked to input the initial top position to a program would potentially get radically different results.

Unfortunately, many real-life problems involve "unstable" systems. It is important that one can recognize them.

Definition. Let $(\mathbf{X},d)$ be a metric space. A map f: $\mathbf{X} \to \mathbf{X}$ is said to have *sensitive dependence* on initial conditions if there is a $\delta > 0$ with the property that for all $\mathbf{x} \in \mathbf{X}$ and all neighborhoods $\mathbf{U}$ of $\mathbf{x}$, there is a $\mathbf{y} \in \mathbf{U}$ and an $n \geq 0$ such that $d(f^n(\mathbf{x}), f^n(\mathbf{y})) > \delta$. The map f is *topologically transitive* if for every pair of open sets $\mathbf{U}, \mathbf{V} \subseteq \mathbf{X}$ there is an $n > 0$ so that $f^n(\mathbf{U}) \cap \mathbf{V} \neq \phi$. The map f is said to be *chaotic* on $\mathbf{X}$ if

(1) it has sensitive dependence on initial conditions,
(2) it is topologically transitive, and
(3) its periodic points are dense in $\mathbf{X}$.

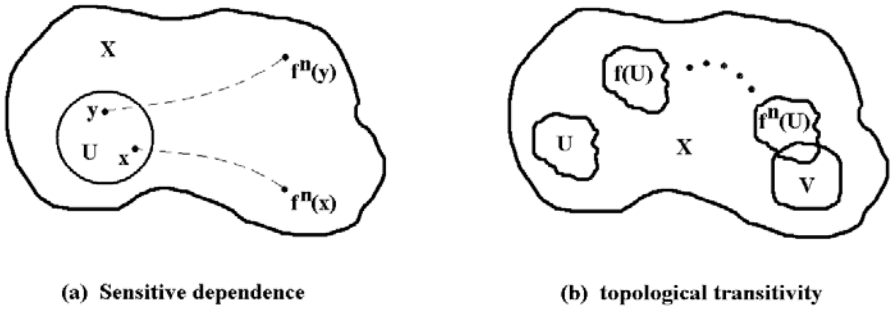


Figure 22.1. Properties of maps with respect to iteration.

Saying that a map f has sensitive dependence on initial conditions means that there are points arbitrarily close to any given point **x** which separate from **x** by some fixed δ under iterations of f. See Figure 22.1(a). Saying that a map f is topologically transitive means that every neighborhood has points that are eventually moved into any other neighborhood under iterates of the map. See Figure 22.1(b). If all orbits are dense, then the map is topologically transitive. The converse is true for compact subsets of **R** or $\mathbf{S}^1$.

The definition of a chaotic map is the well-known definition from [Deva86]. It turns out however that conditions (1)–(3) in the definition are not independent. It is shown by Banks et al. ([BBGDS92]) that conditions (2) and (3) imply (1), so that being chaotic is a purely topological property and does not depend on a metric. Further, Vellekoop and Berglund ([VelB94]) proved that in the case of connected subsets of **R** condition (2) implies (3). They also showed that (1) and (3) do not imply (2). Nor do (1) and (2) imply (3). Therefore, for connected sets of **R**, a map is chaotic if and only if it satisfies condition (3) alone. It follows that the essential condition is topological transitivity. Crannell ([Cran95]) looks at alternatives to that condition that are more intuitive.

Chaotic maps are maps that we would prefer not to have to deal with, but some very simple maps are chaotic.

22.2.1 Example. If we parameterize the points of the unit circle by the angle they make with the x-axis, then the map

$$f : S^1 \to S^1$$
$$f(\theta) = 2\theta$$

is chaotic. Any small arc eventually covers all of the circle, some points move apart, and the periodic points are dense.

22.2.2 Example. Another simple chaotic map is the map

$$f : [0,1] \to [0,1]$$
$$f(x) = 4x(1 - x).$$

See [Deva86] for a much more thorough discussion of these ideas. It turns out that almost any nonlinear feedback process of this type leads to an interesting dynamical system. In particular, let us look at rational maps of the complex plane **C**. These provide a rich source of chaotic maps

Consider the map

$$f_{\mathbf{c}} : \mathbf{C} \to \mathbf{C}$$

defined by

$$f_{\mathbf{c}}(\mathbf{z}) = \mathbf{z}^2 + \mathbf{c}. \tag{22.2}$$

This map was studied by Mandelbrot [Mand83] and looks innocent enough, but looks are deceiving. We have the sequence

$$\mathbf{z}_{n+1} = f_{\mathbf{c}}(\mathbf{z}_n) = \mathbf{z}_n{}^2 + \mathbf{c}.$$

The trivial case where $\mathbf{c} = 0$ is easy to analyze. If $|\mathbf{z}| < 1$, then the $\mathbf{z}_n$ will converge to 0. If $|\mathbf{z}| > 1$, then the $\mathbf{z}_n$ will converge to ∞. On the other hand, things are not so simple for may other values of **c**. For example, Figure 22.2 shows the case where $\mathbf{c} = 0.31 + 0.04\mathbf{i}$. Iterates of points in the interior of the black region converge to the point labeled "attractive fixed point" (the terminology will be explained shortly) and those of points outside the region converge to ∞. The boundary curve of the black region is a very wild curve.

We need some more definitions. Assume f: $\mathbf{R}^n \to \mathbf{R}^n$. (This includes the important special case of complex maps since the complex plane can be identified with $\mathbf{R}^2$.)

Definition. A fixed point **p** of f is a *repelling fixed point* of f if $f'(\mathbf{p})$ has eigenvalues larger than 1. The point **p** is an *attractive fixed point* of f if $f'(\mathbf{p})$ has eigenvalues smaller than 1. If **p** is a periodic point of period k, then **p** is called an *attractive periodic point* if it is an attractive fixed point of f^k.

Definition. If **p** is an attractive fixed point of f, then its *basin of attraction*, A(f,**p**), is defined by

$$A(f, p) = \{\mathbf{x} \mid f^k(\mathbf{x}) \to \mathbf{p} \text{ as } k \to \infty\}.$$

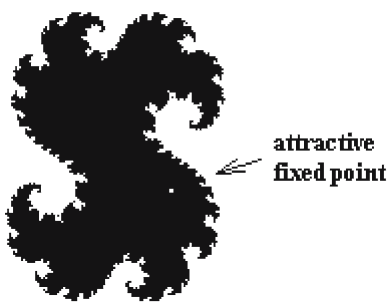


Figure 22.2. The map $\mathbf{z}^2 + (0.31 + 0.04\mathbf{i})$.

For example, the "point" ∞ is always an attractive fixed point of the map $f_{\mathbf{c}}$ defined by equation (22.2), so that we always have a basin of attraction in this case which we shall denote by $A(f_{\mathbf{c}},\infty)$. It turns out that for some values of **c** (for example, for **c** with $|\mathbf{c}|$ much smaller than 1) there is a second attractive fixed point and that for other values of **c** (for example, for **c** with $|\mathbf{c}|$ much larger than 1) ∞ is the **only** attractive fixed point. With regard to the case $\mathbf{c} = 0.31 + 0.04\mathbf{i}$ and Figure 22.2, there is one finite attractive fixed point and its basin of attraction is the interior of the black region in the figure.

One can show that $A(f,\mathbf{p})$ is invariant under f. Some basic questions in the theory of dynamical systems are:

(1) What does the boundary of $A(f,\mathbf{p})$ look like?
(2) How "nice" a set is this boundary?
(3) What is its dimension?

Except for certain special cases such as the *Julia set* $J_{\mathbf{c}} = \partial A(f_{\mathbf{c}},\infty)$, few answers are known in general. Usually the only way to get some information is via approximation.

Let $\mathbf{A} = A(f,\mathbf{p})$. No point of $\partial\mathbf{A}$ can converge to **p**, but points arbitrarily close to $\partial\mathbf{A}$ will. Furthermore, roughly speaking, the closer a point is to $\partial\mathbf{A}$, the longer it will take to converge to **p**. Thus what one can do is to take small disks **D** around **p** and ask which points will end up in **D** after k iterations. Let us call this set $\mathbf{A}_k$. It is an approximation for **A**. The larger k, the better the approximation. The "level sets"

$$L_k = \mathbf{A}_k - \mathbf{A}_{k-1}$$

are approximations to $\partial\mathbf{A}$.

Finally,

Definition. The set

$$\mathbf{M} = \left\{\mathbf{c} \mid f_c^{\,k}(0) \to \infty \text{ as } k \to \infty\right\}$$

is called the *Mandelbrot set*.

See Figure 22.3. Today there are probably very few people, if any, who work with computers who have not seen some of the beautiful computer-generated images of

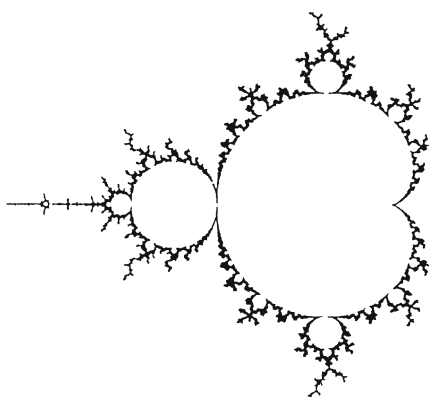


Figure 22.3. The Mandelbrot set.

this set. It was popularized by B. Mandelbrot's work on fractals [Mand83]. So what is a fractal? This leads us to the next topic.

22.3 Dimension Theory and Fractals

The concept of dimension, as obvious as it might seem, is something that only began to be studied seriously in recent times. Of course, everyone is aware of the fact that we seem to live in a three-dimensional world, but this idea remained vague, intuitive, and was relatively unexplored until the late nineteenth century, although the concept of n dimensions was introduced earlier in that century by A. Cayley, H. Grassmann, and B. Riemann. At that time the concept meant quantities that needed, in some unspecified way, a minimum of n real parameters to describe their points. Cantor's discovery in 1878 that there was a bijective function between $\mathbf{R}$ and $\mathbf{R}^2$ showed that "dimension" had nothing to do with the "number" of points involved. A few years later in 1890, Peano proved the existence of a continuous map from [0,1] onto the unit square $[0,1] \times [0,1]$, so that "dimension" could in fact **not** be defined in terms of the least number of continuous parameters required to describe a space. It was only at the beginning of the twentieth century that satisfactory definitions of dimensions were developed as a result of work of H. Poincaré, L.E.J. Brouwer, K. Menger, and P. Urysohn. These definitions were essentially inductive definitions that involved looking at the lower-dimensional subsets of a space that disconnect it. Many other definitions of dimensions have been given since then. They are all equivalent for "nice" spaces but can differ in other cases.

One of the first most basic results was the proof by Brouwer of the topological invariance of what one calls the dimension of Euclidean space.

22.3.1 Theorem. (Brouwer) If $n \neq m$, then $\mathbf{R}^n$ is not homeomorphic to $\mathbf{R}^m$.

Proof. This is Theorem 7.2.3.5(2) in [AgoM05].

Here is how dimension is defined in [HurW48]. One starts off defining the empty space to have dimension –1 and then recursively defines the dimension of the space to be the least integer n for which every point has arbitrarily small neighborhoods whose boundaries have dimension less than n. More precisely,

Definition. The *topological dimension* of a space $\mathbf{X}$, denoted by $\dim \mathbf{X}$, is an integer n satisfying the following conditions:

(1) The empty set and only it has dimension –1.

(2) $\mathbf{X}$ has dimension $\leq n$ ($n \geq 0$) at a point $\mathbf{p}$ if $\mathbf{p}$ has arbitrarily small neighborhoods whose boundaries have dimension $\leq n - 1$.

(3) $\mathbf{X}$ has dimension $\leq n$ if $\mathbf{X}$ has dimension $\leq n$ at each of its points.

(4) $\mathbf{X}$ has dimension n at a point $\mathbf{p}$ if it is true that $\mathbf{X}$ has dimension $\leq n$ at $\mathbf{p}$ and it is false that $\mathbf{X}$ has dimension $\leq n - 1$ at $\mathbf{p}$.

(5) $\mathbf{X}$ has dimension n if $\dim \mathbf{X} \leq n$ is true and $\dim \mathbf{X} \leq n - 1$ is false.

(6) $\mathbf{X}$ has dimension ∞ if $\dim \mathbf{X} \leq n$ is false for all n.

See [HurW48] for how this rather complicated definition can be used to prove some standard facts about the dimension function. In particular, $\mathbf{R}^n$ has dimension n in this new sense and a subset of $\mathbf{R}^n$ is n-dimensional if and only if it contains an open subset of $\mathbf{R}^n$. Boundaries of nice n-dimensional sets like manifolds are (n – 1)-dimensional. Subsets of dimension less than n – 1 cannot disconnect $\mathbf{R}^n$.

Another approach to the concept of dimension is via measure theory. This fact should not be surprising when one considers how different words such as "length" and "area" are used to measure the "size" of one- and two-dimensional sets, respectively. Note also how length is obtained by approximations using finite segments. The sum of the individual segment lengths (raised to the power 1) is the approximating value. In the two-dimensional case, one could use a union of squares (or rectangles) to approximate the region, so that the area would be approximated by a sum of terms each of which is a length squared. A two-dimensional region could be thought of as an infinite union of segments, but defining its area by adding up the lengths of these segments (raised to the power 1) would give the value infinity. One could say that a two-dimensional region has infinite "length" and zero "volume." Only areas, the measure of rectangles, lead to a nontrivial finite result. In general, the "correct" measure is the power n that one needs so that adding up the individual measures leads to a non-trivial result. In 3-space, we need cubes, but covering an area with cubes would take an arbitrarily small volume.

In order to motivate the next definition of dimension, we change our viewpoint slightly and also try to make things more precise. It takes $1/\varepsilon$ intervals of length ε to cover the interval [0,1]. It takes $1/\varepsilon^2$ squares of width ε to cover $[0,1] \times [0,1]$. More generally, one would expect that the number, $n(\varepsilon)$, of d-dimensional cubes of width ε needed to cover a nonempty compact d-dimensional set $\mathbf{X}$ in $\mathbf{R}^n$ should be defined by a formula of the form

$$n(\varepsilon) = c\left(\frac{1}{\varepsilon}\right)^d,$$

where c is a constant that depends on the set. Solving for d gives

$$d = \frac{\ln n(\varepsilon) - \ln c}{\ln(1/\varepsilon)}.$$

We are interested in what happens when ε gets small. In that case the term

$$\frac{\ln c}{\ln(1/\varepsilon)}$$

goes to zero. These intuitive comments lead to the following.

Definition. Let $\varepsilon > 0$ and let $n(\mathbf{X},\varepsilon)$ be the smallest number of closed balls of radius ε whose union covers $\mathbf{X}$. Define the *fractal dimension* of $\mathbf{X}$, $\dim_F \mathbf{X}$, by

$$\dim_F \mathbf{X} = \lim_{\varepsilon \to 0} \frac{\ln n(\mathbf{X},\varepsilon)}{\ln 1/\varepsilon},$$

provided that this limit exists.

We should point out that we are using Barnsley's terminology ([Barn88]) here, but there does not seem to be any universal agreement for the name "fractal dimension." For example, in Alligood et al. ([AlSY97]) it is called the "box-counting dimension." Alligood et al. use d-dimensional boxes instead of closed balls, but that is an unimportant difference.

We shall describe one more notion of dimension. Although we restrict ourselves to subsets of $\mathbf{R}^n$, what we say next applies to subsets of an arbitrary metric space.

Let $\mathbf{X}$ be a nonempty bounded subset of $\mathbf{R}^n$. Let p be an arbitrary real number, $0 \le p < \infty$. If $0 < \varepsilon$, define

$$m_p(\mathbf{X},\varepsilon) = \inf\left\{\sum_{i=1}^{\infty}(\operatorname{diam}(\mathbf{U}_i))^p \;\middle|\; \mathbf{U}_i \subset \mathbf{X},\ \bigcap_{i=1}^{\infty}\mathbf{U}_i = \mathbf{X},\ \text{and}\ \operatorname{diam}(\mathbf{U}_i) < \varepsilon\right\}$$

and

$$m_p(\mathbf{X}) = \sup\{m_p(\mathbf{X},\varepsilon) \mid \varepsilon > 0\}.$$

Definition. The quantity $m_p(\mathbf{X})$ is called the *Hausdorff p-dimensional measure* of $\mathbf{X}$.

The interesting property of the function $m_p(\mathbf{X})$ is that, as a function of p, it assumes only three possible values, namely, 0, ∞, or a **single** finite value.

22.3.2 Theorem. There is a unique real number d_H, $0 \le d_H \le n$, so that

$$\begin{aligned} m_p(\mathbf{X}) &= \infty \quad \text{if } p < d_H \\ &= 0 \quad \text{if } p > d_H. \end{aligned}$$

Proof. See [Fede69] or [Falc85].

Definition. The number d_H appearing in Theorem 22.3.2 is called the *Hausdorff-Besicovitch dimension* of $\mathbf{X}$ and will be denoted by $\dim_H \mathbf{X}$.

The various dimensions can be related.

22.3.3 Theorem. If $\mathbf{X}$ is a bounded subset of $\mathbf{R}^n$, then $\dim \mathbf{X} \le \dim_H \mathbf{X} \le \dim_F \mathbf{X} \le n$.

Proof. See [HurW48] and [Barn88].

On "nice" spaces these definitions of dimension lead to the same **integer** dimension. One final definition of dimension will be given in the next section.

What we have here is a good example of where some topic that people used to think was of interest only to hardcore mathematicians, all of a sudden got some very practical importance. After all, who would have thought that weird spaces with non-integer dimensions would ever be relevant to the "real" world. Of course, most people might not even have been aware of weird-dimensional spaces because dimension theory is typically only encountered by graduate students in mathematics. Perhaps a

more meaningful example, which shows the kind of reactions that mathematicians encounter from nonmathematicians, has to do with the existence of continuous but nowhere differentiable functions. A typical nonmathematics major would quickly put this out of his/her mind as being totally irrelevant.

Here is how Mandelbrot defined a fractal.

Definition. A *fractal* is a set whose Hausdorff-Besicovitch dimension is larger than its topological dimension.

This definition is not totally satisfactory as even Mandelbrot pointed out. There are spaces that have "fractal"-type properties that are not fractals. The definition has therefore evolved into a more visual one, namely, that a fractal set should exhibit one or more of the following: a complicated structure over a wide range of scales, self-similarity (see Section 22.4), and/or have a noninteger type of dimension.

A well-known simple example of a fractal is the *triadic Koch curve*. See Figure 22.4. One starts with an equilateral triangle and successively replaces each straight line segment of length e by four smaller segments, each of length e/3, as indicated in the figure. If L(e) is the length of the curve when the length of each edge is e, then one can show that this function satisfies the recurrence relation

$$L\left(\frac{e}{3}\right) = \frac{4}{3}L(e).$$

This leads to a solution $L(e) = e^{1-d}$, where

$$d = \frac{\log 4}{\log 3} = 1.2618$$

and the logarithm is to any base. Rewriting L(e) as

$$L(e) = e^{-d}e,$$

we see that the term e^{-d} can be thought of as the number of sides in the Koch curve if one uses edges of length e. Since the only product

$$e^{-d}e^{p}$$

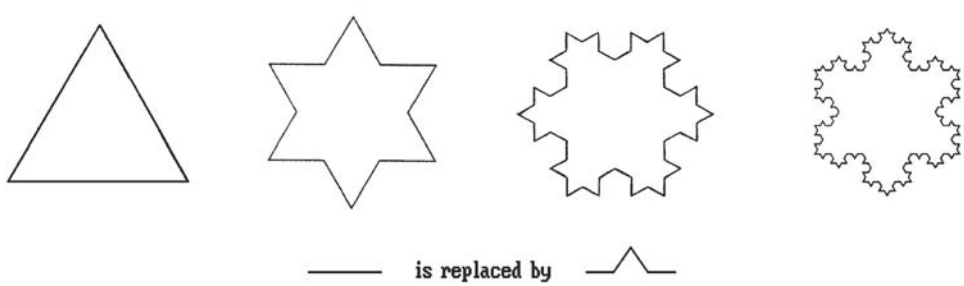


Figure 22.4. Four stages of the triadic Koch curve.

which stays finite as e approaches 0 is where p = d, the Koch curve is a fractal of dimension d. See [Mand83] for an interesting connection between the Koch curve and the coastline of Britain (they both have the same dimension). The Koch curve is only one example of a space that can be defined by recursive substitution. See Glassner's overview ([Glas92]) of constructions of this type and how they can create interesting objects that can be viewed at different scales.

Looking at the concept of dimension from a slightly different point of view, one notices that, in real life, dimension is relative to a particular context. Mandelbrot refers to this everyday notion of dimension as the *effective dimension*. For example, consider a 10-centimeter-wide ball of 1-millimeter thread. The dimension associated to it would depend on the distance of the viewer to the object:

Distance to object	Effective dimension
∞	0 (one sees a point)
10 cm	3 (one sees a solid ball)
10 mm	1 (one sees the threads)
0.1 mm	3 (the threads look like columns)
0.01 mm	1 (one sees the fibers in the threads)

Finally, what do fractals have to do with dynamical systems and chaos? In a sense, not much. The Mandelbrot set above is a fractal, however, and so the connection between the two is that the sets important to the study of a dynamical system, in particular chaotic ones, often are fractals.

22.4 Iterated Function Systems

This section gives a very brief overview of iterated function systems. A good reference for this topic is [Barn88] and our discussion here basically touches on some highlights from that book. We start with two examples.

Consider transformations $w_1, w_2, \ldots, w_k$ of the plane of the form

$$w_i(\mathbf{p}) = \mathbf{p}A_i + \mathbf{p}_i,$$

where the A_i are 2×2 matrices and the $\mathbf{p}_i$ are fixed points. We are interested in what happens to points as we repeatedly apply the transformations w_i to them. For example, consider the transformations w_1, w_2, and w_3, where

$$A_1 = A_2 = A_3 = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix} \tag{22.3}$$

and

$$\mathbf{p}_1 = (1,1), \quad \mathbf{p}_2 = (1,50), \quad \text{and} \quad \mathbf{p}_3 = (50,50). \tag{22.4}$$

Consider Algorithm 22.4.1. What the algorithm does is start out with a white rectangle $[1,50] \times [1,50]$ that has a black boundary and then successively applies the trans-

formations w_1, w_2, and w_3 to all the points of the rectangle. Figure 22.5 shows the pictures after the first seven iterations in Algorithm 22.4.1.

Algorithm 22.4.1 is what Barnsley calls a deterministic algorithm. We can introduce some chance into the algorithm by picking our transformations w_i randomly. This and a slight generalization of Algorithm 22.4.1 leads to the nondeterministic Algorithm 22.4.2. We start off with $\mathbf{q}_0 = (0,0)$ and then generate the sequence of points

```
integer array x[1..100,1..100], y[1..100,1..100], p1[1..2], p2[1..2], p3[1..2] ;
real array A1[1..2,1..2], A2[1..2,1..2], A3[1..2,1..2];
integer i, j;

Assuming that 1 corresponds to the color white and 0 to black, initialize the array x
so that

    x[i,j] := 0  if  i = 1,100  or  j = 1,100 ,
           := 1  ,  otherwise

Draw x;
Initialize the arrays Ai and pi to the values shown in (22.3) and (22.4), respectively.

loop
    Initialize the array y to all 1s.
    for  i:=1 to 100  do
        for  j:=1 to 100  do
            if  x[i,j] = 0  then
                begin
                    y[ [i,j]A1+p1 ] := 0;        { We truncate coordinates to integers }
                    y[ [i,j]A2+p2 ] := 0;
                    y[ [i,j]A3+p3 ] := 0;
                end;
    x := y;
    Draw x;
endloop;
```

Algorithm 22.4.1. A deterministic iterated function system.

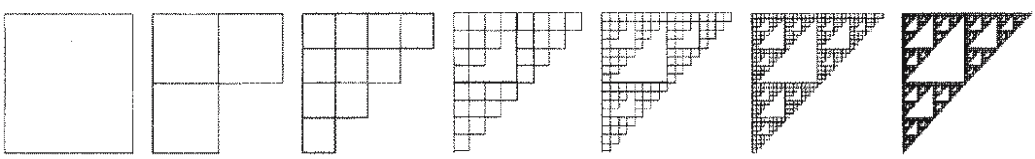


Figure 22.5. Results of a deterministic iterated function system ([Barn88]).

```
real array  A1[1..2,1..2], A2[1..2,1..2], … , Am[1..2,1..2], p1[1..2], p2[1..2], … , pm[1..2];
real array  σ[1..m];
real        x, y, newx, newy;
integer     numIterations, n, ix, iy;

Initialize the arrays Ai, pi, and σ;

Set the screen to white;
(x,y) := (0,0);  numIterations := Desired number of iterations;
for  n:=1 to numIterations  do
    begin
        k := one of the indices {1,2, … ,m}, where probability of picking index i is σ[i];
        (newx,newy) := [x,y]Ak+pk;
        (ix,iy) := round   (scale factor)*(newx,newy)+(screen center)  to integers;
        if  n > 10  then  SetPixel (ix,iy,black);                    { Skip first 10 iterations }
        (x,y) := (newx,newy);
    end;
```

Algorithm 22.4.2. A nondeterministic iterated function system.

Figure 22.6. Results of a nondeterministic iterated function system.

$\mathbf{q}_i$, $i > 0$, using the formula $\mathbf{q}_i = T(\mathbf{q}_{i-1})$, where T is one of the transformations w_1, w_2, . . . , w_m chosen at random (with possibly different probabilities) at each stage. The scaled and offset points are plotted on the screen as we go along, except that the first few are discarded and not plotted because they would introduce some initial "noise" into the picture. Figure 22.6 shows the result after a sufficient number of iterations using the following data with m = 4 from [Barn88]:

$$A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 0.85 & -0.04 \\ 0.04 & 0.85 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 0.2 & 0.23 \\ -0.26 & 0.22 \end{pmatrix}, \quad A_4 = \begin{pmatrix} -0.15 & 0.26 \\ 0.28 & 0.24 \end{pmatrix},$$

$$\mathbf{p}_1 = (0,0), \quad \mathbf{p}_2 = (0,1.6), \quad \mathbf{p}_3 = (0,1.6), \quad \mathbf{p}_4 = (0,0.44),$$

$$\sigma[1] = 0.01, \quad \sigma[2] = 0.85, \quad \sigma[3] = 0.07, \quad \sigma[4] = 0.07.$$

As we can see, the picture looks like a fern. The question arises as to how one could generate some other natural phenomena. To answer this question we first need to develop the mathematical setting in which the iterated function systems referred to in the figures will be defined. Some of the definitions and theorems will sound pretty abstract, but hopefully the reader will look beyond that and see the intuitive ideas they are trying to capture.

Let $(\mathbf{X},d)$ be a complete metric space.

Notation. Let $H(\mathbf{X})$ denote the set of nonempty compact subsets of $\mathbf{X}$.

Define a map

$$d_H : H(\mathbf{X}) \times H(\mathbf{X}) \to \mathbf{R}$$

by

$$d_H(\mathbf{A},\mathbf{B}) = \max(\mathrm{dist}(\mathbf{A},\mathbf{B}), \mathrm{dist}(\mathbf{B},\mathbf{A})).$$

22.4.1 Lemma. The function d_H is a metric on $H(\mathbf{X})$.

Proof. See [Barn88].

Definition. The metric d_H is called the *Hausdorff metric* on $H(\mathbf{X})$.

Barnsley calls the metric space $(H(\mathbf{X}),d_H)$ the *space of fractals*.

22.4.2 Theorem. $(H(\mathbf{X}),d_H)$ is a complete metric space.

Proof. See [Barn88].

Definition. Let $(\mathbf{X},d)$ be a metric space. A map $f: \mathbf{X} \to \mathbf{X}$ is called a *contraction mapping*, or simply a *contraction*, of $\mathbf{X}$ if there exists a constant c, $0 \le c < 1$, so that

$$d(f(\mathbf{x}),f(\mathbf{y})) \le c\,d(\mathbf{x},\mathbf{y})$$

for all $\mathbf{x}$, $\mathbf{y} \in \mathbf{X}$. The constant c is called a *contractivity factor* of f.

The simplest examples of contraction mappings are the *radial transformation* $f: \mathbf{R}^n \to \mathbf{R}^n$, $f(\mathbf{p}) = c\mathbf{p}$, with $0 \le c < 1$.

22.4.3 Theorem. (The Contraction Mapping Theorem) Any contraction mapping f of a complete metric space $\mathbf{X}$ has a unique fixed point. In fact, if $\mathbf{x}$ is any point of $\mathbf{X}$, then the sequence of points $\mathbf{x}$, $f(\mathbf{x})$, $f^2(\mathbf{x})$, . . . converges to that fixed point.

Proof. The proof follows from the easily proved observation that the sequence of points $\mathbf{x}$, $f(\mathbf{x})$, $f^2(\mathbf{x})$, . . . is a Cauchy sequence.

We can now describe a way to generalize Algorithms 22.4.1 and 22.4.2 (and also the chaos game construction in Programming Project 1.5.5). The key elements are captured by the following definition:

Definition. An *iterated function system*, or *IFS* for short, is a pair $((\mathbf{X},d),W)$, where $(\mathbf{X},d)$ is a complete metric and $W = \{w_1,w_2,\ldots,w_k\}$ is a finite set of contraction maps w_i: $\mathbf{X} \to \mathbf{X}$. The *contractivity factor* c of this system is defined by

$$c = \max\{c_1,c_2,\ldots,c_k\},$$

where c_i is the contractivity factor of w_i. An *iterated function system with probabilities* is a tuple $((\mathbf{X},d),W,P)$, where $((\mathbf{X},d),W)$ is an IFS and $P = \{p_1,p_2,\ldots,p_{|W|}\}$ is a set of probabilities, that is,

$$p_i > 0 \quad \text{and} \quad p_1 + p_2 + \ldots + p_{|w|} = 1.$$

22.4.4 Theorem. Let $((\mathbf{X},d),W)$ be an IFS with contraction factor c. Define

$$w_W : H(\mathbf{X}) \to H(\mathbf{X})$$

by

$$w_W(\mathbf{B}) = \bigcup_{w \in W} w(\mathbf{B}).$$

Then the map w_W is a contraction map on $(H(\mathbf{X}),d_H)$ with contractivity factor c. Furthermore, the unique fixed point $\mathbf{A}$ of w_W, called the *attractor* of the IFS $((\mathbf{X},d),W)$, satisfies

(1) $\mathbf{A} = \bigcup_{w \in W} w(\mathbf{A})$, and

(2) $\mathbf{A} = \lim_{n\to\infty} w_W{}^n(\mathbf{B})$ for any $\mathbf{B} \in H(\mathbf{X})$.

Proof. See [Barn88].

It is the theory behind Theorem 22.4.4 that explains the "deterministic" Algorithm 22.4.1. We refer the reader to Barnsley's book for the theorem that corresponds to Theorem 22.4.4 in the case of iterated function systems with probabilities and the "nondeterministic" Algorithm 22.4.2. It involves technicalities of measure theory that would take us past the level of our presentation here. Nevertheless, it is the study of iterated function systems with probabilities that really leads to an understanding of fractals.

We move on in our discussion of "deterministic" IFSs in order to state the fundamental theorem in this topic.

Definition. Let $(\mathbf{X},d)$ be a complete metric space and let $\mathbf{A} \in H(\mathbf{X})$. The map w: $H(\mathbf{X}) \to H(\mathbf{X})$ defined by $w(\mathbf{B}) = \mathbf{A}$, $\mathbf{B} \in H(\mathbf{X})$, is called a *condensation transformation* of $\mathbf{X}$ and $\mathbf{A}$ is called its *condensation set*.

It is easy to check that a condensation transformation on $\mathbf{X}$ is a contraction mapping on $H(\mathbf{X})$ that has contractivity factor 0 and that its unique fixed point is its condensation set.

Definition. If $((\mathbf{X},d),W)$ is an IFS and w is a condensation transformation of $\mathbf{X}$, then $((\mathbf{X},d),W \cup \{w\})$ is called an *IFS with condensation*.

22.4.5 Theorem. (The Collage Theorem) Let $(\mathbf{X},d)$ be a complete metric space and assume that we are given an $\mathbf{L} \in H(\mathbf{X})$ and an $\varepsilon > 0$. Let $((\mathbf{X},d),W)$ be any IFS, or IFS with condensation, so that

$$d_H\left(\mathbf{L}, \bigcup_{w \in W} w(\mathbf{L})\right) \le \varepsilon,$$

where d_H is the Hausdorff metric on $H(\mathbf{X})$. If c is the contractivity factor of the IFS and $\mathbf{A}$ is its attractor, then

$$d_H(\mathbf{L},\mathbf{A}) \le \frac{\varepsilon}{1-c},$$

or equivalently,

$$d_H(\mathbf{L},\mathbf{A}) \le \frac{1}{1-c} d_H\left(\mathbf{L}, \bigcup_{w \in W} w(\mathbf{L})\right) \text{for all } \mathbf{L} \in H(\mathbf{X}).$$

Proof. See [Barn88].

The importance of the Collage Theorem is that it tells us how we can find an IFS whose attractor is close to a given set. Specifically, it means that if we want to generate a particular type of shape, say a tree, we do not have to try lots of contraction mappings at random. On the other hand, one needs to realize that IFSs are not used to reproduce any specific object. Furthermore, they define an image and not individual objects in an image. A useful fact is that small changes to the contraction maps will lead to small changes in the attractor and that the attractor depends in a "continuous" way on the contraction maps. Changing one contractive map may make parts of objects appear or disappear in the image.

We return now to the topic of dimension. What is the dimension of the attractors of IFSs? To answer this question, we first define the "address" of a point in its attractor. Given an IFS $((\mathbf{X},d),W)$, one of the things one does is take points $\mathbf{x}$ of $\mathbf{X}$ and look at what happens to them under successive transformations via elements of W, that is, one deals with points

$$\mathbf{y} = (\omega_1 \circ \omega_2 \circ \ldots \circ \omega_m)(\mathbf{x}),$$

where $\omega_i \in W$. If W consists of the maps $w_1, w_2, \ldots, w_k$, then

$$\omega_i = w_{j_i}$$

for some index j_i and so $\mathbf{y}$ is completely specified by the sequence of indices $j_1, j_2, \ldots, j_m$. It is useful to introduce some notation.

Definition. The *code space* of an IFS $((\mathbf{X},d),W)$ is defined to be the metric space $(\mathbf{\Sigma},d_{\mathbf{\Sigma}})$, where $\mathbf{\Sigma}$ is the space of all infinite sequences $(s_1,s_2,\ldots)$, $s_i \in \{1,2,\ldots,|W|)$, and $d_{\mathbf{\Sigma}}$ is defined by

$$d_{\Sigma}((s_1,s_2,\ldots),(t_1,t_2,\ldots)) = \sum_{i=1}^{\infty} \frac{|s_i - t_i|}{(|W|+1)^i}.$$

One can show that the map $d_{\mathbf{\Sigma}}$ in the definition of a code space is in fact a metric.

Definition. Let $(\mathbf{\Sigma},d_{\mathbf{\Sigma}})$ be the code space of the IFS $((\mathbf{X},d),W)$. Let $W = \{w_1,w_2,\ldots w_k\}$. For $m = 1,2,\ldots$, define

$$\varphi_m : \mathbf{\Sigma} \times \mathbf{X} \to \mathbf{X}$$

by

$$\varphi_m((s_1,s_2,\ldots),\mathbf{x}) = (w_{s_m} \circ w_{s_{m-1}} \circ \ldots \circ w_{s_1})(\mathbf{x}).$$

22.4.6 Theorem. Let $((\mathbf{X},d),W)$ be an IFS with code space $(\mathbf{\Sigma},d_{\mathbf{\Sigma}})$ and attractor $\mathbf{A}$. Then

$$\lim_{m\to\infty} \varphi_m(\sigma,\mathbf{x})$$

exists for all $\sigma \in \mathbf{\Sigma}$ and $\mathbf{x} \in \mathbf{X}$ and its value is independent of $\mathbf{x}$ and lies in $\mathbf{A}$. In other words, there is a well-defined map

$$\varphi : \Sigma \to \mathbf{A}$$

defined by

$$\varphi(\sigma) = \lim_{m\to\infty} \varphi_m(\sigma,\mathbf{x}). \tag{22.5}$$

The map φ is continuous and onto $\mathbf{A}$.

Proof. See [Barn88].

Given a point in the attractor $\mathbf{A}$, there may be many ways that one might converge to it under compositions of maps from W.

Definition. Let $((\mathbf{X},d),W)$ be an IFS with code space $(\mathbf{\Sigma},d_{\mathbf{\Sigma}})$ and attractor $\mathbf{A}$. Let φ: $\mathbf{\Sigma} \to \mathbf{A}$ be the map defined by equation (22.5). Let $\mathbf{a} \in \mathbf{A}$. Any element of $\varphi^{-1}(\mathbf{a})$ is called an *address* of $\mathbf{a}$. The IFS is said to be *totally disconnected* if each point of $\mathbf{A}$ has a unique address. It is *just touching* if it is not totally disconnected and $\mathbf{A}$ has an open set $\mathbf{O}$ so that

(1) $w(\mathbf{O}) \cap w'(\mathbf{O}) = \phi$ for all $w, w' \in W$ with $w \neq w'$, and
(2) $\bigcup_{w\in W} w(\mathbf{O}) \subseteq \mathbf{O}$.

The IFS is said to be *overlapping* if it is neither just touching nor totally disconnected.

22.4.7 Theorem. Let $((\mathbf{R}^n,d),W)$ be an IFS where the elements of W are similarity transformations. Let $W = \{w_1,w_2, \ldots ,w_k\}$, let c_i be the contraction factor of w_i, and let D, $D \in [0,\infty)$, be the unique solution to the equation

$$\sum_{i=1}^{k} |c_i|^D = 1.$$

Let **A** be the attractor of the IFS and let $\dim_F \mathbf{A}$ and $\dim_H \mathbf{A}$ be its fractal and Hausdorff-Besicovitch dimension.

(1) If the IFS is totally disconnected or just-touching, then $D \le n$ and $D = \dim_F \mathbf{A} = \dim_H \mathbf{A}$.

(2) If the IFS is overlapping, then $\dim_F \mathbf{A} \le D$.

Proof. See [Barn88].

Barnsley explains how one can use Theorem 22.4.7 to estimate the fractal dimension of an attractor of an IFS. The integer D in the theorem actually has a name.

Definition. A set $\mathbf{X} \subseteq \mathbf{R}^n$ is said to be *invariant* for a set $\{w_1,w_2, \ldots ,w_k\}$ of contractions of $\mathbf{R}^n$ if

$$\mathbf{X} = \bigcup_{i=1}^{k} w_i(\mathbf{X}).$$

If the contractions are similarity transformations of $\mathbf{R}^n$ and if for some integer s, the Hausdorff s-dimensional measure $m_s(\mathbf{X}) > 0$ but

$$m_s(w_i(\mathbf{X}), w_j(\mathbf{X})) = 0, \quad \text{for} \quad i \ne j,$$

then the set **X** is said to be *self-similar* and the *similarity dimension* of **X**, $\dim_S \mathbf{X}$, is the integer D defined by the equation

$$\sum_{i=1}^{k} |c_i|^D = 1, \tag{22.5}$$

where c_i is the contraction factor of w_i.

For a more thorough discussion of self-similarity and a proof of the fact that the similarity dimension is well-defined see [Falc85]. Intuitively, a self-similar set is a set that is built up out of parts that are similar to the entire set. For example, a cube is self-similar because it can be divided into four smaller cubes, each of which can be divides into four smaller cubes, and so on. The triadic Koch curve is also a self-similar set. The next corollary, which states that the fractal, Hausdorff-Besicovitch, and sim-

ilarity dimension are the same in certain situations, is an immediate consequence of Theorem 22.4.7.

22.4.8 Corollary. Let $((\mathbf{R}^n,d),W)$ be a totally disconnected IFS where the elements of W are similarity transformations. If $\mathbf{A}$ is the attractor of this IFS, then

$$\dim_F \mathbf{A} = \dim_H \mathbf{A} = \dim_S \mathbf{A}.$$

This finishes our brief overview of iterated function systems. It should have at least given the reader an idea of how one can compute fractals very easily and the kind of mathematics that is involved in **proving** the properties that one hopes to be true.

APPENDIX A

Notation

$\mathbf{N}$	= the natural numbers $\{0,1,2,\ldots\}$
$\mathbf{Z}$	= the ring of integers
$\mathbf{R}$	= the field of real numbers
$\mathbf{R}^*$	= the extended real numbers, that is, $\mathbf{R} \cup \{\infty\}$
$\mathbf{I}$	= the unit interval $[0,1]$
$\mathbf{C}$	= the field of complex numbers
$\mathbf{H}$	= the non-commutative division ring of quaternions

In the context of an n-tuple $\mathbf{p}$, p_i will always refer to the ith component of $\mathbf{p}$. The same holds for functions. If f: $\mathbf{R}^n \to \mathbf{R}^m$, then f_i is the ith component function of f, that is,

$$f(\mathbf{p}) = (f_1(\mathbf{p}), f_2(\mathbf{p}), \ldots, f_m(\mathbf{p})).$$

$\mathbf{N}^n$	$= \{\mathbf{z} = (z_1, z_2, \ldots, z_n) \mid z_i \in \mathbf{N}\}$
$\mathbf{Z}^n$	$= \{\mathbf{z} = (z_1, z_2, \ldots, z_n) \mid z_i \in \mathbf{Z}\}$
$\mathbf{R}^n$	$= \{\mathbf{p} = (p_1, p_2, \ldots, p_n) \mid p_i \in \mathbf{R}\}$
	= n-dimensional Euclidean space
$\mathbf{R}^n_+$	$= \{\mathbf{p} \in \mathbf{R}^n \mid \mathbf{p}_n \geq 0\}$
	= the upper halfplane of $\mathbf{R}^n$
$\mathbf{R}^n_-$	$= \{\mathbf{p} \in \mathbf{R}^n \mid \mathbf{p}_n \leq 0\}$
	= the lower halfplane of $\mathbf{R}^n$
$\mathbf{I}^n$	$= \{\mathbf{p} = (p_1, p_2, \ldots, p_n) \mid p_i \in \mathbf{I}\}$
	= the unit "cube" in $\mathbf{R}^n$
δ_{ij}	= Kronecker delta (1, if i = j, and 0, otherwise)

$\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ = standard (orthonormal) basis of $\mathbf{R}^n$, that is, $\mathbf{e}_i = (\delta_{i1}, \delta_{i2}, \ldots, \delta_{in})$

$\lvert\mathbf{v}\rvert$	= length of vector $\mathbf{v}$
$\mathbf{pq}$	= the segment from point $\mathbf{p}$ to point $\mathbf{q}$ in $\mathbf{R}^n$, **unless** $\mathbf{p}$ and $\mathbf{q}$ are quaternions in which case this denotes their product
$\lVert\mathbf{pq}\rVert$	= signed distance from $\mathbf{p}$ to $\mathbf{q}$
$\angle(\mathbf{u},\mathbf{v})$	= angle between vectors $\mathbf{u}$ and $\mathbf{v}$
$\angle_s(\mathbf{u},\mathbf{v})$	= signed angle between vectors $\mathbf{u}$ and $\mathbf{v}$

$\mathbf{B}^n(\mathbf{p},r)$	$= \{\mathbf{q} \in \mathbf{R}^n \mid \lvert\mathbf{pq}\rvert < r\}$
$\mathbf{B}^n(r)$	$= \mathbf{B}^n(\mathbf{0},r)$
$\mathbf{B}^n$	$= \mathbf{B}^n(\mathbf{0},1)$
	= the open (n-dimensional) unit disk in $\mathbf{R}^n$
$\mathbf{D}^n(\mathbf{p},r)$	$= \{\mathbf{q} \in \mathbf{R}^n \mid \lvert\mathbf{pq}\rvert \le r\}$
	= an n-dimensional closed disk
$\mathbf{D}^n$	$= \mathbf{D}^n(\mathbf{0},1)$
	= the closed (n-dimensional) unit disk in $\mathbf{R}^n$
$\mathbf{S}^{n-1}$	$= \{\mathbf{q} \in \mathbf{R}^n \mid \lvert\mathbf{q}\rvert = 1\}$
	= the (n − 1)-dimensional unit sphere in $\mathbf{R}^n$
$\mathbf{S}^{n-1}_+$	$= \mathbf{S}^{n-1} \cap \mathbf{R}^n_+$
	= the upper hemisphere
$\mathbf{S}^{n-1}_-$	$= \mathbf{S}^{n-1} \cap \mathbf{R}^n_-$
	= the lower hemisphere
$\mathbf{P}^n$	= n-dimensional projective space

There are natural inclusions: $0 = \mathbf{R}^0 \subset \mathbf{R}^1 \subset \mathbf{R}^2 \subset \ldots$
Similarly for the other spaces above.

$\mathbf{X}^k$	= the k-fold Cartesian product $\underbrace{\mathbf{X} \times \mathbf{X} \times \ldots \times \mathbf{X}}_{k}$ of the set $\mathbf{X}$
$\mathbf{X} \Delta \mathbf{Y}$	$= (\mathbf{X} - \mathbf{Y}) \cup (\mathbf{Y} - \mathbf{X})$ (symmetric difference)
$\mathbf{H}_\pm(f)$	= halfspace associated to function f
$\mathbf{H}(\mathbf{p},\mathbf{n})$	= halfplane defined by plane containing point **p** and with normal vector **n**
$i\mathbf{H}(\mathbf{p},\mathbf{n})$	= interior of halfplane $\mathbf{H}(\mathbf{p},\mathbf{n})$
$\mathbf{P}(\mathbf{p},\mathbf{n})$	= hyperplane $\partial\mathbf{H}(\mathbf{p},\mathbf{n})$
inf $\mathbf{X}$	= infimum or greatest lower bound of the set **X** of real numbers
sup $\mathbf{X}$	= supremum or least upper bound of the set **X** of real numbers
cl $(\mathbf{X})$	= closure of **X**
int $(\mathbf{X})$	= interior of **X**
r$\mathbf{X}$	= regularization of **X**
op*	= regularized set operator, where op = $\cup$, $\cap$, −, c (complement), and Δ (symmetric difference)
aff $(\mathbf{X})$	= affine hull of **X**
conv $(\mathbf{X})$	= convex hull of **X**
$f(a^+)$	= right-handed limit of f at a
$f(a^-)$	= left-handed limit of f at a
$f^{(d)}(x)$	= the dth derivative of f

$I = I^n$	= $n \times n$ identity matrix which consists of 1s along the diagonal and 0s elsewhere
A^T	= transpose of the matrix A
GL (n,k)	= the linear group of nonsingular $n \times n$ matrices over k = **R** or **C**
O (n)	= the group of real orthogonal $n \times n$ matrices
SO (n)	= the group of real special orthogonal $n \times n$ matrices
R (f,g) = R_X (f,g)	= the resultant of polynomials f(X) and g(X)
$\bar{\mathbf{z}}$	= the complex conjugate of the complex number or quaternion **z**

T (**A**,**B**, . . .) = (**A**′,**B**′, . . .): This means that T (**A**) = **A**′, T (**B**) = **B**′, . . .

$1_{\mathbf{X}}$	= the identity map on the set **X**
$\chi_{\mathbf{A}}$	= the characteristic function of a set **A** as a subset of a given larger set **X** ($X_A(x) = 1$ if $x \in A$ and 0 otherwise.)
f^{-1} (y)	= {x | f(x) = y}
$\|\ \|_p$	= L^p norm
<f,g>	= the inner product of f and g in $L^2([a,b])$
a | b	= a divides b
Sign (x)	= +1 if $x \geq 0$ and −1 otherwise (returns an **integer**)
Sign (σ)	= sign of permutation σ = +1 if σ is an even permutation, −1 if σ is an odd permutation
Trunc (x)	= greatest integer ≤x (returns a **real** value)
Floor (x)	= greatest integer ≤x (returns an **integer** value)
exp (x)	= e^x
κ (s)	= curvature function
τ (s)	= torsion function

Let p(u) be a parametrized curve:

$$p_c = p(c), \quad p^u = p' = \frac{dp}{du}, \quad p_c^u = p'(c)$$

Let p(u,v) be a parametrized surface:

$$p_{ab} = p(a,b), \quad p^u = \frac{\partial p}{\partial u}, \quad p_{ab}^u = \frac{\partial p}{\partial u}(a,b), \quad p^v = \frac{\partial p}{\partial v}, \quad p_{ab}^v = \frac{\partial p}{\partial v}(a,b)$$

$$p^{uv} = \frac{\partial^2 p}{\partial u \partial v}, \quad p_{ab}^{uv} = \frac{\partial^2 p}{\partial u \partial v}(a,b)$$

Commutative Diagram. In general, if one has a directed graph where the nodes are sets and the arrows correspond to maps between these sets, then this is said to constitute a commutative diagram if, whenever two directed paths start and end at the same points, the corresponding composition of maps is equal. Commutative diagrams are nice to have and the terminology is useful in many areas of mathematics. As an example, consider the diagram

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ g\downarrow & & \downarrow F \\ C & \xrightarrow[G]{} & D \end{array}$$

If $G(g(a)) = F(f(a))$ for all $a \in A$, then the diagram is said to be *commutative*.

APPENDIX B

Abstract Program Syntax

Control Structures:

Actions (either via simple statements or "procedure" calls): S;

The concatenation of **more** than one action will be surrounded by **begin—end**:

```
begin
    S1; S2; . . .
end;
```

Assignment is indicated via a ":=".
Equality is indicated via "=".

Selection:

```
if  B  then  S;
if  B  then  S1  else  S2;

case  c  of
   c1 : S1;
   .    .
   .  : .
   ck : Sk;
end;
```

Iteration:

```
while  B  do  S;
repeat  S  until  B;
for  i:=m to n  do  S;
```

Other: Constructs that are clearly derived from corresponding *mathematical* expressions are also allowed such as

```
for each  x in X  do  S;
```

Comments are anything between a pair of curly brackets: { }

Standard Abstract Data Types:

```
integer, real, string, etc.,
array, list, tree, graph, etc.
integer array, real list, etc.
```

These are defined by their **operations** (not some lower level data structure). At lower levels, to describe **implementations**, there is also a **record** type that combines known data types to produce other types. For example,

```
employee = record
       string  name;
       integer age;
          .
            .
end;
employee  e;
```

Fields of a record are referenced using the standard dot notation, as in "e.name". Alternatively, we shall use the capitalized field name as an operator to access the field. For example, Name(e) will be used to denote e.name.

All pointer variables will end in "Ptr" and we shall sometimes use the convention that if <z>Ptr is a declared pointer variable, then <z> is the item to which the pointer points.

Pointers are often implementation artifacts, but if really needed in an abstract program, we use operations such as PointerTo, ValueOf, SetValueOf, and PointerToFirst, as in

```
integer i, k;
integer pointer iPtr;
integer list L;

iPtr := PointerTo (i);
k    := ValueOf (iPtr);
SetValueOf (iPtr,7);
iPtr := PointerToFirst (L);
```

If Ptr is a pointer to a record, as in

```
employee pointer Ptr;
```

we use the notation Ptr→age to denote the "age" field of the record pointed to by Ptr. For example,

```
Ptr→age := 45;
i := Ptr→age;
```

Procedures/Functions:

```
procedure p (<parameters>);
<return type> function f (<parameters>);
```

Reference parameters, that is, parameters that are passed by address, are indicated by "**ref**". Any other parameters will be assumed to be value parameters by default. For example,

```
integer function (ref real r, integer i);
```

indicates that the parameter r is passed by address and i, by value.

All procedure/function names will be capitalized. All variables will start with a lowercase letter.

APPENDIX C

IGES

C.1 What Is IGES?

The goal of this appendix is not to provide an authoritative account of the IGES product definition interface but simply to give the reader an idea of what IGES files look like and what it takes to generate them. For anything more, the reader is referred to the official manuals from the National Bureau of Standards. The functionality of IGES has changed as CAD and CAGD has evolved and anyone doing serious work with IGES would have to get the latest manual; however, the core of the standard has stayed the same and the specific reference for the discussion here is [IGES88].

To use IGES one must write and read files that follow a very special format. There are two basic file types consisting of a binary or ASCII format. Binary IGES files are not readable and the less common of the two. We shall have nothing to say about them here. The ASCII file format, which is readable and can be created using a text editor, comes in two flavors. There is a compressed format and one that consists of fixed 80 character lines. We shall restrict ourselves to describing the latter.

C.2 A Sample IGES File

Figure C.1 shows some excerpts of a sample IGES file in ASCII format. A "." at the left end of an otherwise blank line means that one or more lines in the actual file were omitted in the sample listing. Our goal is to describe the parts in this file and a few other aspects of IGES. This should be adequate to give the reader a good idea of what is involved with IGES. The next two sections will discuss a few additional geometric and nongeometric type formats.

To begin with every IGES file in ASCII format consists of a sequence of 80 character lines divided into five nonempty sections that appear in the following order:

Start section
Global section
Directory Entry section
Parameter Data section
Terminate section

```
THIS IS A SIMPLE IGES FILE FOR A SMALL PART                              S      1
                                                                         S      2
,,1H1,15HSMALL-PART.IGES,43HIHIS IS SYSTEM ID STRING FOR GRAPHIC SYSTEM,G0000001
16HIGES VERSION 3.0,16,08,24,08,56,1H1,1.0000,1,4HINCH,,,13H03 522.13554G      2
7,,,3HMAX,7HPRIVATE;                                                     G      3
     124       1       1                                        00000000D      1
     124                       3                                         D      2
.
.
     110      19       1       1       0               1                0D     13
     110                       1                                         D     14
.
.
     100      39       1       1       0               1                0D     43
     100                       2                                         D     44
     112      41       1       1       0               1                0D     45
     112                      13                                         D     46
.
.
     106      70       1               0               1           1     D     51
     106                       3       2                                 D     52
     108      73       1               0               1           1     D     53
     108                       1       1                                 D     54
.
.
     406      95       1               0               0           1 2   D     75
     406                       1    5555                                 D     76
     410      96       1               0              49           1 3   D     77
     410                       1                                         D     76
.
.
124,      1.000000,      0.000000,      0.000000,      0.000000,       1P      1
          0.000000,      1.000000,      0.000000,      0.000000,       1P      2
          0.000000,      0.000000,      1.000000,      0.000000,       1P      3
.
.
110,.0000000,.0000000,.0000000,.0000000,3.000000,.0000000;             13P     19
.
.
100,-0.4000000E+01,2.000000,1.000000,2.500000,1.000000,2.000000,       43P     39
1.500000;                                                              43P     40
112,3,1,2,5,.0000000,1.000000,2.000000,3.000000,4.000000,              45P     41
5.000000,3.000000,.8722851,.0000003576279,-.01331125,3.000000,         45P     42
-0.1481045E+01,-0.2384186E-06,.3784813,.0000000,.0000000,              45P     43
.
.
7.000000,.3864541,-.000002853572,.3735719,1.000000,-.4769966,          45P     52
.0000003278255,1.477365,.0000000,.0000000,.0000000,.0000000;           45P     53
.
.
106,2,5,.0000000,34.00000,-0.1000000,.0000000,34.00000,                51P     70
0.1000000,.0000000,34.10000,0.1000000,.0000000,                        51P     71
34.10000,-0.1000000,.0000000,34.00000,-0.1000000;                      51P     72
108,1.000000,.0000000,.0000000,.0000000,51;                            53P     73
.
.
406,1,2H01;                                                            75P     95
410,1,1.000000,53,57,61,65,69,73,0,1,75;                               77P     96
.
.
S      2G      3D    146P    159                                         T      1
```

Figure C.1. Part of a sample IGES file.

Columns 74–80 of every line must contain a right-justified sequence number that starts with 1 for the first line in every section and increases sequentially by 1 up to the last line in that section. The number can have leading zeros. Column 73 in every line must contain one of the letters "S," "G," "D," "P," or "T" depending on whether that line belongs to the Start, Global, Directory Entry, Parameter Data, or Terminate section, respectively. Check out these parts in Figure C.1. In the rest of the discussion we shall only describe the contents of columns 1–72.

IGES files use two single-character delimiters. One separates parameters within a record and the other separates records. The defaults for these delimiters are the comma and the semicolon, respectively. Real constants consist of basic decimal notation real numbers or such numbers followed by an exponent that is defined by an "E" followed by a signed integer or a "D," for double precision, followed by a signed integer. String constants have the form nHx, where n is a character count and x is a string of n characters. For example, on line 4 of the file in Figure C.1, "4HINCH" represents the string "INCH." Pointer constants of one to seven digit integers refer to a line number in the file.

The Start Section. Columns 1–72 of each line in this section can contain any text whatsoever, formatted in any way whatsoever.

The Global Section. This section consists of a collection of 22 parameters that describe the origin of the file and information needed to interpret the file. In particular, the first two string parameters define the parameter and record delimiter. The two commas at the beginning of the global section in our sample file indicate that we are using the defaults (comma and semicolon). Several of these parameters are described below:

Parameter	Field type	Description
1	String	Parameter delimiter
2	String	End of record delimiter
3	String	Product identification from sending system
4	String	File name
.		
7	Integer	Number of bits for integer representation
8	Integer	Number of bits in a single precision floating point exponent
9	Integer	Number of bits in a single precision floating point mantissa
10	Integer	Number of bits in a double precision exponent
11	Integer	Number of bits in a double precision mantissa
.		
14	Integer	Unit flag
15	Integer	Units: 4HINCH for unit flag = 1 and 2HMM for unit flag = 2 (Other unit flags and units are possible)
.		
18	String	Date and time of file generation of the form 13HYYMMDD.HHNNSS

In our sample file, parameters 7–11 are 16, 8, 24, 8, and 56, respectively.

	1 – 8	9 – 16	17 – 24	25 – 32	33 – 40	41 – 48	49 – 56	57 – 64	65 – 72	73 – 80
LINE 1	ENTITY TYPE NO.	PARA-METER DATA	STRUC-TURE	LINE FONT PATTERN	LEVEL	VIEW	TRANSFOR-MATION MATRIX	LABEL DISPLAY	STATUS	SEQ #
	# 1	> 2	#,> 3	#,> 4	#,> 5	0,> 6	0,> 7	> 8	# 9	D-------- 10
LINE 2	ENTITY TYPE NO.	LINE WEIGHT	COLOR	PARA-METER LINE COUNT	FORM NUMBER	RESERVED	RESERVED	ENTITY LABEL	ENTITY SUB-SCRIPT	SEQ #
	# 11	# 12	#,> 13	# 14	# 15	16	17	18	# 19	D-------- 20

integer
> pointer
#,> integer or pointer (pointer has negative sign)
0 zero

Figure C.2. The Directory Entry Section.

The Directory Entry Section. There is one directory entry for each entity in the file and each consists of two lines that are subdivided into 8-bit fields. The object of the section is to provide a list of all entities and to specify their attributes. The ordering of the entries can be arbitrary except that certain definition entities must precede their instances. In our sample file in Figure C.1 there are actually 73 entities although we have only shown some of them. Figure C.2 shows the fields in the directory entry lines and their names. We describe the more interesting ones.

Entity Type Number. Each entity has a unique number. Numbers 100 through 199 are reserved for geometric entities. The nongeometry entities come in two flavors: annotation entities and structure entities. The entities and their numbers will be discussed in more detail in Sections C.3 and C.4

Parameter Data. This pointer specifies the number of the line in the parameter data section where the data for this entity starts. For example, the data for the 124 entity starts at line 1 and that of the 110 entity starts at line 19 in the parameter data section.

Transformation Matrix. This points to a transformation entity (entity number 124). All geometric points need to be transformed by this transformation. For example, entities 110, 100, 112, 106, and 108 all use the transformation entity specified in directory entry line number 1.

Status. This field consists of four 2-digit values whose meaning is shown in Figure C.3. For example, in our file entity 406 is a physically dependent definition.

Parameter Line Count. The number of lines used by this entity in the parameter data section. In our file, entity 100 used two lines of data and entity 106 used six.

Digits :	1-2	3-4	5-6	7-8
Meaning:	Blank status 00 visible 01 blanked	Subordinate entity switch 00 independent 01 physically dependent 02 logically dependent 03 both (01) and (02)	Entity use flag 00 geometry 01 annotation 02 definition 03 other 04 logical	Hierarchy 00 Global top down 01 Global defer 02 Use hierarchy property

Figure C.3. The Directory Entry Status field.

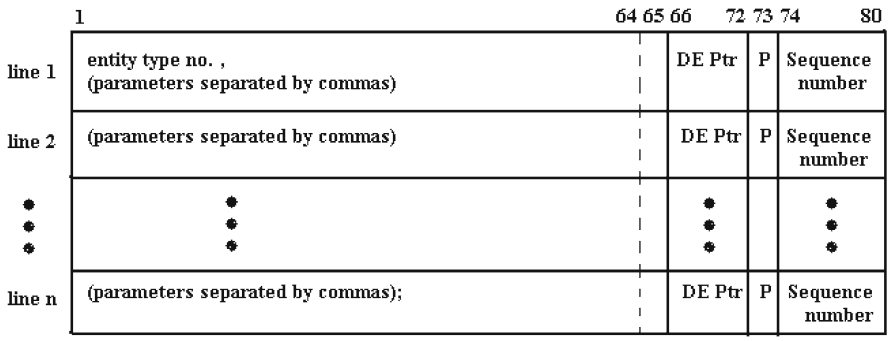


Figure C.4. The Parameter Data Section for one entity.

Form Number. Some entities have different interpretations. The interpretation numbers are specified in the entity descriptions. In our file, entity 106 specified interpretation 2, which means that data points are presented as coordinate triples.

Parameter Data Section. This section contains the data associated to each entity. The data is entered in free format on each line in columns 1–64, except that the first field of the first line of data for an entity must be the entity type number. For example, in our file note the number 124 at the beginning of the first line of the parameter data section indicating that this is data for some entity of type 124 that was specified in the directory entry section. Column 65 should be blank. Columns 66–72 on each parameter data line contains the sequence number of the directory entity that uses this data. In our file, note the number 1 in the first three lines of the parameter data section. Two groups of parameters can come at the end of the specified parameters for each entity unless the record delimiter appears first. The first group of parameters may contain pointers to associativity instances, general notes, and/or text template entities. The second group of parameters may contain pointers to one or more properties. For example, see parameters 8 to 9 + n + m for arc entity number 100 in Figure C.6. Section C.4 will explain these extra parameters more. Any desired comments may be added after the record delimiter and these can run over several lines. See Figure C.4 for an outline of the structure of the parameter data section.

Terminate Section. This section consists of only one line consisting of ten 8-character-long fields. The first four fields start with "S," "G," "D," and "P" corresponding to the start, global, directory entry, and parameter data sections, respectively, followed by the number of lines in that section right-justified in the field. For example, in our file in Figure C.1 we can see that there were 2, 3, 146, and 159 lines in the start, global, directory entry, and parameter data sections, respectively. Columns 33–72 are not used.

C.3 The IGES Geometric Types

This section discusses a few of the geometric entities in IGES and their parameter specification. Figure C.5 lists some of the available entities. We describe those that appear in our sample file in Figure C.1.

Entity Number 124. This transformation matrix entity defines a 3×4 matrix of the form

Entity type #	Entity type
100	Circular arc
102	Composite curve
104	Conic arc
106	Copious data
	Centerline
	Linear path
	Section line
	Simple closed area
	Witness line
108	Plane
110	Line
112	Parametric spline curve
114	Parametric spline surface
116	Point
118	Ruled surface
120	Surface of revolution
122	Tabulated cylinder
124	Transformation matrix
125	Flash
126	Rational B-spline curve
128	Rational B-spline surface
130	Offset curve
132	Connect Point
134	Node
136	Finite element
138	Nodal displacement and rotation
140	Offset surface
142	Curve on a parametric surface
144	Trimmed parametric surface
CSG Types:	
150	Block
152	Right angular wedge
154	Right circular cylinder
156	Right circular cone frustrum
158	Sphere
160	Torus
162	Solid of revolution
164	Solid of linear extrusion
168	Ellipsoid
180	Boolean tree
184	Solid assembly

Figure C.5. Table of some geometric IGES entities.

$$\begin{pmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{pmatrix}$$

with the entries stored in row major form in the parameter data section. The matrix (rij) is assumed to be an orthogonal matrix. The entity corresponds to a transformation defined in terms of column vectors by

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \to \begin{pmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ r31 & r32 & r33 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t1 \\ t2 \\ t3 \end{pmatrix}.$$

Our entity had form number 0. Its data started at line number 1 of the parameter data section, used three lines, and defined the identity transformation. Using other form numbers allows one to pass additional information.

Note. As we list the data for the geometric entities below, keep in mind that normally it would have to be transformed by the transformation matrix associated to the entity to get the "real" data. In our case, we are dealing with the identity transformation so that this is not necessary and we shall not keep pointing that out.

Entity Number 110. A line entity defines a segment between two points (x1,y1,z1) and (x2,y2,z2) whose coordinates are stored in a sequential manner in the parameter data section. In our case the data was stored in parameter data line number 19 and defined the segment [(0,0,0),(0,3,0)].

Entity Number 100. The data for this circular arc entity consists of two lines starting with parameter data line number 39. Figure C.6 describes the meaning of the data. The first number on line 39 is the entity number. The rest of the fields have the following values:

$$zt = -0.4000000E01 = -4.0,$$

$$(x1, y1) = (2.0,1.0), \quad (x2, y2) = (2.5,1.0), \quad (x3, y3) = (2.0,1.5).$$

In other words, our arc lies in the plane z = –4 parallel to the xy-plane. It has center (2,1,–4), starts at (2.5,1,–4), and ends at (2,1.5,–4). The fact that a semicolon follows the x3 and y3 values means that there are no extra pointers, that is, n = m = 0.

Note. The fields 8 through 9 + n + m in Figure C.6 are potentially present in most entities, although we shall no longer bother to mention them if they have not been given any values.

Entity Number 112. The parametric spline curve entity corresponds to a spline curve

$$p(u) = (x(u), y(u), z(u))$$

Parameter	Name	Type	Description
1	zt	Real	Parallel zt displacement of arc from xt, yt plane
2	x1	Real	Arc center x-coordinate
3	y1	Real	Arc center y-coordinate
4	x2	Real	Start point x-coordinate
5	y2	Real	Start point y-coordinate
6	x3	Real	Terminate point x-coordinate
7	y3	Real	Terminate point y-coordinate
8	n	Integer	Number of back pointers (to associativity entities)/ text pointers (to general note entities)
9	DE	Pointer	.
.			Pointers to associativities or general notes
.			.
8 + n	DE	Pointer	.
9 + n	m	Integer	Number of properties
10 + n	DE	Pointer	.
.			Pointers to properties
.			.
9 + n + m	DE	Pointer	.

Figure C.6. Parameter data for circular arc entity #100.

some of whose parameter data fields are described in Figure C.7. In particular,

$$x(u) = AX(i) + BX(i)s + CX(i)s^2 + DX(i)s^3$$
$$y(u) = AY(i) + BY(i)s + CY(i)s^2 + DY(i)s^3$$
$$z(u) = AZ(i) + BZ(i)s + CZ(i)s^2 + DZ(i)s^3$$

for

$$T(i) \le u \le T(i+1), \quad i = 1, 2, \ldots, N,$$

and

$$s = u - T(i).$$

The coefficients D or the coefficients D and C will be zero if the polynomials are of degree 2 or 1, respectively. If the curve is planar, then the Z coefficients will be zero, except that AZ(i) will specify the plane z = AZ(i) that contains the curve. So that one can get the value and the first, second, and third derivative at the end point of the curve without computing the polynomial at u = T(N + 1), these values, divided by appropriate factorials, are included in the parameter data at the end of the coefficient data. The entity in our example has its data in parameter data lines 41 through 53. We see that

Parameter	Name	Type	Description
1	CTYPE	Integer	Spline type 1 = linear 4 = Wilson-Fowler 2 = quadratic 5 = modified Wilson-Fowler 3 = cubic 6 = B-spline
2	H	Integer	Continuity with respect to arc length at breakpoints 1 = curve is continuous and has slope continuity 2 = curve is continuous and has both slope and curvature continuity
3	NDIM	Integer	2 = planar 3 = non-planar
4	N	Integer	Number of segments
5	T(1)	Real	Break points of piecewise polynomial
.			
5+N	T(N+1)	Real	
6+N	AX(1)	Real	x-Coordinate polynomial
7+N	BX(1)		
8+N	CX(1)		
9+N	DX(1)		
10+N	AY(1)	.	y-Coordinate polynomial
.			
13+N	DY(1)		
14+N	AZ(1)	.	z-Coordinate polynomial
.			
18+N	AX(2)		
.	.		
6+13N	TPX0	.	Terminate point x-value
.	TPX1	.	Terminate point x-value of 1st derivative
	TPX2		Terminate point x-value of 2nd derivative/2!
	TPX3		Terminate point x-value of 3rd derivative/3!
	TPY0	.	Terminate point y-value
.	.		

Figure C.7. Parameter data for parametric spline curve entity #112.

$$\text{CTYPE} = 3,\ \text{H} = 1,\ \text{NDIM} = 2,\ \text{N} = 5,\ \text{T(i)} = \text{i},\ \text{i} = 0,1,\ldots,5,$$

$$(\text{AX(1)},\text{BX(1)},\text{CX(1)},\text{DX(1)}) = (3.0,.8722851,.0000003576279,-.01331125),$$

$$(\text{AY(1)},\text{BY(1)},\text{CY(1)},\text{DY(1)}) = (3.0,-0.1481045\text{E+01},-0.2384186\text{E–06},.3784813,0),$$

$$\vdots$$

$$(\text{TPZ0},\text{TPZ1},\text{TPZ2},\text{TPZ3}) = (.0,.0,.0,.0).$$

In other words the entity defines a cubic planar spline that is continuous and has slope continuity at the six breakpoints T(i) = i, i = 0, 1, . . . , 5.

Entity Number 106. The copious data entity has multiple meanings depending on its form number. Figure C.8 describes some of the fields of its parameter data. When the form number is 1, 2, or 3, then the IP field takes on the same value and they both have the same meaning. Its data starts in parameter data line number 70 and consists of 3 lines altogether. In our case

$$\text{form number} = \text{IP} = 2, \quad \text{N} = 5,$$

and we have five data points

$$(0,34,-0.1), (0,34,0.1), (0,34.1,0.1), (0,34.1,-0.1), (0,34,-0.1).$$

How the points are interpreted is usually determined by the entity that refers to this one as we shall see when we discuss the next entity.

Entity Number 108. This is the plane entity. There is one parameter data line for it and that line has number 73. Figure C.9 describes some, but not all, parameters that can be associated to it. In our case we have

Parameter	Name	Type	Description
1	IP	Integer	Interpretation flag IP = 1 x, y pairs, common z IP = 2 x, y, z triples IP = 3 x, y, z coordinates i, j, k vector coordinates
2	N	Integer	Number of n-tuples
Case IP = 2:			
3 → 2+3N			Sequence of x, y, z tuples of data values

Figure C.8. Partial parameter data for copious entity #106.

Parameter	Name	Type	Description
1	A	Real	Corresponds to plane
2	B	Real	
3	C	Real	Ax + By + Cz = D
4	D	Real	
5	DE	Pointer	Pointer to directory entry of closed curve entity or 0

Figure C.9. Partial parameter data for plane entity #108.

$$A = 1, \quad B = 0, \quad C = 0, \quad D = 0, \quad DE = 51.$$

Because DE is nonzero, we have a closed curve in our plane that happens to be the copious data entity described above consisting of five points in the plane x = 0.

C.4 The IGES Nongeometric Types

Figure C.10 lists the nongeometric entity types for IGES version 3.0.

Entity Number 406. Property entities can contain numerical or textual data. The form number specifies the type of property at hand. Low numbers are predefined and numbers 5001–9999 are left for a user to define. In our case, the form number 5555 is a user-defined property. The first number in the parameter data section after the entity number is the number of properties. In our case, it is 1 and the property is the string "01".

Entity Number 410. A view entity specifies how an object should be viewed. The projection is assumed to be a parallel orthographic projection. In the view coordinate system the view plane is assumed to be the plane z = 0 with the origin being the origin of the view plane. The view direction is along the positive z-direction. The positive y-axis is the "up" direction. One can also specify a view volume and scale factor. Figure C.11 shows the layout of the view volume. In the case of our entity, the fact that the status is physically dependent and the entity use flag is "other" means that a drawing entity number 404 points to it. See Figure C.12 for a description of the fields in the parameter data. In our case,

Annotation Entities		Structure Entities	
Entity type #	Entity type	Entity type #	Entity type
202	Angular dimension entity	302	Associativity definition entity
106	Centerline entity	402	Associativity instance entity
206	Diameter dimension entity	404	Drawing entity
208	Flag note entity	304	Line font definition entity
210	General label entity	306	MACRO definition entity
212	General note entity	600-699	MACRO instance entity
214	Leader (arrow) entity	406	Property entity
216	Linear dimension entity	308	Subfigure definition entity
218	Ordinate dimension entity	408	Singular subfigure instance entity
220	Point dimension entity	412	Rectangular array subfigure instance entity
222	Radius dimension entity		
106	Section entity	414	Circular array subfigure instance entity
106	Witness line entity		
		310	Text font definition entity
		410	View entity

Figure C.10. Some IGES annotation and structure entities.

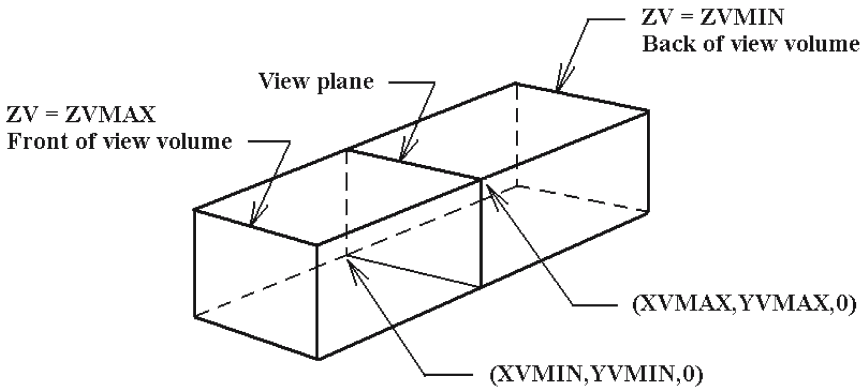


Figure C.11. The view volume for entity number 410.

Parameter	Name	Type	Description
1	VNO	Integer	View number
2	SCALE	Real	Scale factor (default = 1.0)
3	XVMINP	Pointer	Pointer to left side of view volume (XVMIN plane) or 0
4	YVMAXP	Pointer	Pointer to top of view volume (YVMAX plane) or 0
5	XVMAXP	Pointer	Pointer to right side of view volume (XVMAX plane) or 0
6	YVMINP	Pointer	Pointer to bottom of view volume (YVMIN plane) or 0
7	ZVMINP	Pointer	Pointer to back of view volume (ZVMIN plane) or 0
8	ZVMAXP	Pointer	Pointer to front of view volume (ZVMAX plane) or 0

Figure C.12. Partial parameter data for plane entity #410.

Parameter	Name	Type	Description
1	N	Integer	Number of entries
2	DE	Pointer	Pointer to entity 1
.	.	.	.
N+1	DE	Pointer	Pointer to entity N

Figure C.13. Parameter data for associativity instance entity #402 with form number 1.

$$\text{VNO} = 1, \quad \text{SCALE} = 1.0,$$
$$\text{XVMINP} = 53, \quad \text{YVMAXP} = 57, \quad \text{XVMAXP} = 61,$$
$$\text{YVMINP} = 65, \quad \text{ZVMINP} = 69, \quad \text{ZVMAXP} = 73.$$

A view volume has been specified. In general, a zero would indicate no clipping in a particular direction. We analyzed the XVMIN clipping plane corresponding to entity

53 in the last section. Our view entity has no pointers to associativity instances, general notes, or text template entities and one pointer to a property (on directory section line number 75).

In closing, we mention one other handy entity, the associativity instance entity with number 402. There are a number of variants of it depending on the form number. The so-called group associativities are particularly useful because they allow a collection of a set of entities to be maintained as a single, logical entity. A common one of these is the case where the form number is 1. This requires an (unordered) group of back pointers as the parameter data. The general structure for its data is shown in Figure C.13.

Bibliography

Abbreviations

ACM TOG	ACM Transactions on Graphics
AMS	American Mathematical Society
CACM	Communication of the ACM
CAD	Computer Aided Design
CAGD	Computer Aided Geometric Design
CAM-I	Computer Aided Manufacturing International, Inc.
CG&A	IEEE Computer Graphics & Applications
CGIP	Computer Graphics and Image Processing
NCC	Proceedings of the National Computer Conference
SIGGRAPH	Proceedings of yearly SIGGRAPH conference, which is the July or August issue of Computer Graphics
SIAM	Society for Industrial and Applied Mathematics

Advanced Calculus

[Apos58] Apostol, Tom M., *Mathematical Analysis: A Modern Approach to Advanced Calculus*, Addison-Wesley Publ. Co., Inc., 1958

[Spiv65] Spivak, M., *Calculus on Manifolds*, W.A. Benjamin, Inc., 1965.

Algebraic Curves and Surfaces

[AbhB87a] Abhyankar, Shreeram S., and Bajaj, Chanderjit, "Automatic Parameterization of Rational Curves and Surfaces I: Conics and Conicoids," CAD, **19**(1), January/February 1987, 11–14.

[AbhB87b] Abhyankar, Shreeram S., and Bajaj, Chanderjit, "Automatic Parametrization of Rational Curves and Surfaces II: Cubics and Cubicoids," CAD, **19**(9), November 1987, 499–502.

[AbhB87c] Abhyankar, Shreeram S., and Bajaj, Chanderjit, "Automatic Parameterization of Rational Curves and Surfaces III: Algebraic Plane Curves," CAGD, **5**(4), November 1988, 11–14.

[Baja92a] Bajaj, Chanderjit, "Algebraic surface Design and Finite Element Meshes," Technical Report CSD-TR-92-011, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, February 1992.

[Baja92b] Bajaj, Chanderjit, "The Emergence of Algebraic Curves and Surfaces in Geometric Design," Technical Report CSD-TR-92-052, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, September 1992.

[ManC92b] Manocha, Dinesh, and Canny, John F., "Algorithm for Implicitizing Rational Parametric Surfaces," CAGD, **9**(1), May 1992, 25–50.

[Taub94] Taubin, Gabriel, "Rasterizing Algebraic Curves and Surfaces," CG&A, **14**(2), March 1994, 14–23.

Algebraic Geometry

[Arno83] Arnon, D.S., "Topologically Reliable Displays of Algebraic Curves," SIGGRAPH 83, **17**(3), July 1983, 219–227.

[Shaf94] Shafarevich, Igor R., *Basic Algebraic Geometry I*, 2nd Edition, Springer-Verlag, 1994.

Algebraic Topology

(See also Topology).

[BotM58] Bott, R., and Milnor, J., "On the Parallelizability of the Spheres," Bull. AMS, **64**, 1958, 87–89.

[DelE95] Delfinado, Cecil Jose A., and Edelsbrunner, Herbert, "An Incremental Algorithm for Betti Numbers of Simplicial Complexes on the 3-Sphere," CAGD, **12**(7), November 1995, 771–784.

[Matv03] Matveev, S., *Algorithmic Topology and Classification of 3-Manifolds*, Springer-Verlag, 2003.

[Stee51] Steenrod, Norman, *The Topology of Fiber Bundles*, Princeton Univ. Press, 1951.

Analytic Geometry

[Full73] Fuller, Gordon, *Analytic Geometry*, 4th Edition, Addison-Wesley Publ. Co., 1973.

[Limi44] Liming, R.A., *Practical Analytical Geometry with Applications to Aircraft*, MacMillan, 1944.

Antialiasing

(See also Visible Surface Detection)

[AbWW85] Abram, G., Westover, L., and Whitted, T., "Efficient Alias-free Rendering Using Bit-masks and Look-up Tables," SIGGRAPH 85, **19**(3), July 1985, 53–59.

[Crow77a] Crow, Franklin C., "The Aliasing Problem in Computer-generated Shaded Images," CACM, **20**(11), 1977, 799–805.

[Will83] Williams, Lance, "Pyramidal Parametrics," SIGGRAPH 83, **17**(3), July 1983, 1–11.

Blending

[AllD97a] Allen, Seth, and Dutta, Debasish, "Cyclides in Pure Blending I," CAGD, **14**(1), January 1997, 51–75.

[AllD97b] Allen, Seth, and Dutta, Debasish, "Cyclides in Pure Blending II," CAGD, **14**(1), January 1997, 77–102.

[AllD97c] Allen, Seth, and Dutta, Debasish, "SuperCyclides and Blending," CAGD, **14**(7), September 1997, 637–651.

[BloW89a] Bloor, M.I.G., and Wilson, M.J., "Generating Blend Surfaces Using Partial Differential Equations," CAD, **21**(3), April 1989, 165–171.

[BloW89b] Bloor, M.I.G., and Wilson, M.J., "Blend Design as a Boundary-Value Problem," in [StrS89], 221–234.

[Chiy87] Chiyokura, H., "An Extended Rounding Operation for Modeling Solids with Free-Form Surfaces," in *Computer Graphics*, Kunii, T.L., editor, Springer, 1987, 249–268.

[ChiK83] Chiyokura, H., and Kimura, F., "Design of Solids with Free-Form Surfaces," SIGGRAPH 83, **17**(3), July 1983, 289–298.

[ChoJ89] Choi, B.K., and Ju, S.Y., "Constant Radius Blending in Surface Modeling," CAD, **21**(4), May 1989, 213–220.

[ElbC97] Elber, Gershon, and Cohen, Elaine, "Filleting and Rounding Using Trimmed Tensor Product Surfaces," in [HofB97], 206–216.

[HofH85] Hoffmann, Christoph M., and Hopcroft, John E., "Automatic Surface Generation in Computer Aided Design," The Visual Computer, **1**, 1985, 95–100.

[HofH87] Hoffmann, Christoph M., and Hopcroft, John E., "The Potential Method for Blending Surfaces and Corners," in [Fari87], 347–365.

[HsuT98] Hsu, Kun Lung, and Tsay, Der Min, "Corner Blending of Free-form N-Sided Holes," CG&A, **18**(1), January/February 1998, 72–78.

[Prat89] Pratt, M.J., "Cyclide Blending in Solid Modelling," in [StrS89], 235–245.
[RoaM92] Roach, P.A., and Martin, R.R., "Production of Blends and Fairings by Fourier Methods," in *Curves and Surfaces in Computer Vision and Graphics III*, Warren, J.D., editor, SPIE, 1992, 162–173.
[RocO87] Rockwood, Alyn P., and Owen, John C., "Blending Surfaces in Solid Modeling," in [Fari87], 367–384.
[RosR84] Rossignac, A.R., and Requicha, A.A.G., "Constant Radius Blending in Solid Modeling," Computers in Mechanical Engineering, July 1984, 65–73.
[Shen98] Shene, Ching-Kuang, "Blending Two Cones with Dupin Cyclides," CAGD, **15**(7), July 1998, 643–673.
[Shen00] Shene, Ching-Kuang, "Do Blending and Offsetting Commute for Dupin Cyclides?" CAGD, **17**(9), October 2000, 891–910.
[Szil91] Szilvasi-Nagy, M., "Flexible Rounding Operation for Polyhedra," CAD, **23**(9), November 1991, 629–633.
[VaMV89] Várady, T., Martin, R.R., and Vida, J., "Topological Considerations in Blending Boundary Representation Solid Models," in [StrS89], 205–220.
[VaVM89] Várady, T., Vida, J., and Martin, R.R., "Parametric Blending in a Boundary Representation Solid Modeller," in [Hand89], 171–197.
[ViMV94] Vida, Janos, Martin, Ralph R., Várady, Tamas, "A Survey of Blending Methods that Use Parametric Surfaces," CAD, **26**(5), May 1994, 341–365.
[Wood87] Woodwark, J.R., "Blends in Geometric Modelling," in [Mart87], 255–298.

Clipping

[CyrB78] Cyrus, M., and Beck, J., "Generalized Two- and Three-Dimensional Clipping," Computers and Graphics, **3**(1), 1978, 23–28.
[GreH98] Greiner, GÜnther, and Hormann, Kai, "Efficient Clipping of Arbitrary Polygons," ACM TOG, **17**(2), 1998, 71–83.
[LiaB83] Liang, You-Dong, and Barsky, Brian A., "An Analysis and Algorithm for Polygon Clipping," CACM, **26**(11), Nov., 1983, 868–877, and Corrigendum, CACM, **27**(2), February 1984, 151.
[LiaB84] Liang, You-Dong, and Barsky, Brian A., "A New Concept and Method for Line Clipping," ACM TOG, **3**(1), January 1984, 1–22.
[Mail92] Maillot, Patrick-Gilles, "A New, Fast Method for 2D Polygon Clipping: Analysis and Software Implementation," ACM Trans. on Graphics, **11**(3), July 1992, 276–290.
[NiLN87] Nicholl, Tina M., Lee, D.T., and Nicholl, Robin A., "An Efficient New Algorithm For 2-D Line Clipping: Its Development and Analysis," SIGGRAPH 87, **21**(4), July 1987, 253–262.
[SutH74] Sutherland, I.E., and Hodgman, G.W., "Reentrant Polygon Clipping," CACM, **17**(1), January 1974, 32–42.
[Vatt92] Vatti, Bala R., "A Generic Solution to Polygon Clipping," CACM, **35**(7), July 1992, 56–63.
[Weil80] Weiler, K., "Polygon Comparison Using a Graph Representation," SIGGRAPH 80, **14**(3), July 1980, 10–18.

Color

[AgoG87] Agoston, G.A., *Color Theory and its Application in Art and Design*, 2nd Edition, Springer-Verlag, 1987.
[Blin93] Blinn, James F., "NTSC: Nice Technology, Super Color," CG&A, **13**(2), March 1993, 17–23.
[Fish90a] Fishkin, Ken, "A Fast HSL-To-RGB Transform," in [Glas90], 448–449.
[GerP90] Gervautz, Michael, and Purgathofer, Werner, "A Simple Method for Color Quantization: Octree Quantization," in [Glas90], 287–293.
[Hall89] Hall, Roy, *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, 1989.
[Heck82] Heckbert, P.S., "Color Image Quantization for Frame Buffer Display," SIGGRAPH 82, **16**(3), July 1982, 297–307.
[Paet90] Paeth, Alan W., "Mapping RGB Triples Onto Four Bits," in [Glas90], 233–245.
[WuXi92] Wu, Xialin, "Color Quantization by Dynamic Programming and Principal Analysis," ACM TOG, **11**(4), Oct., 1992, 348–372.

Computational Geometry

[Aure91] Aurenhammer, Franz, "Voronoi Diagrams—A Survey of a Fundamental Geometric Data Structure," ACM Computing Surveys, **23**(3), September 1991, 345–405.

[BKOS97] de Berg, Mark, van Kreveld, Marc, Overmars, Mark, and Schwarzkopf, Otfried, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 1997.

[BeMR94] Bern, Marshall, Mitchell, Scott, and Ruppert, Jim, "Linear-Size Nonobtuse Triangulation of Polygons," in *Proc. of the 10th Annual Symp. on Computational Geometry*, Stony Brook, New York, June 6–8, 1994, 221–230.

[BDST92] Boissonnat, J.-D., Devillers, O., Schott, R., Teillaud, M., and Yvinec, M., "Applications of Random Sampling to On-line Algorithms in Computational Geometry," Discrete Comp. Geom., **8**, 1992, 51–71.

[BoiT93] Boissonnat, J.-D., and Teillaud, M., "On the Randomized Construction of the Delaunay Tree," Theoret. Comp. Sci., **112**, 1993, 339–354.

[Chaz91] Chazelle, B., "Triangulating a Simple Polygon in Linear Time," Discrete Comput. Geom., **6**, 1991, 485–524.

[CiMS98] Cignoni, P., Montani, C., and Scopigno, R., "DeWall: A Fast Divide and Conquer Delaunay Triangulation Algorithm in E^d," CAD, **30**(5), April 1998, 333–341.

[ClaS89] Clarkson, K.L., and Shor, P.W., "Applications of Random Sampling in Computational Geometry," Discrete Comp. Geometry, **4**, 1989, 387–421.

[Devi98] Devillers, Olivier, "Improved Incremental Randomized Delaunay Triangulation," in *Proc. of the 14th Annual Symp. on Computational Geometry*, Minneapolis, Minnesota, June 7–10, 1998, ACM Press, 106–115.

[Dwye87] Dwyer, Rex A., "A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations," Algorithmica, **2**(2), 1987, 137–151.

[Edel87] Edelsbrunner, Herbert, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987.

[EtzR99] Etzion, Michal, and Rappoport, Ari, "Computing the Voronoi Diagram of a 3-D Polyhedron by Separate Computation of its Symbolic and Geometric Parts," in [BroA99], 167–178.

[FanP93] Fang, Tsung-Pao, and Piegl, Les A., "Delaunay Triangulation Using a Uniform Grid," CG&A, **13**(3), May 1993, 36–47.

[FanP95] Fang, Tsung-Pao, and Piegl, Les A., "Delaunay Triangulation in Three Dimensions," CG&A, **15**(5), September 1995, 62–69.

[Fort87] Fortune, Stephen J., "A Sweepline Algorithm for Voronoi Diagrams," Algorithmica, **2**(2), 1987, 153–174.

[GJPT78] Garey, M.R., Johnson, D.S., Preparata, F.P., and Tarjan, R.E., "Triangulating a Simple Polygon," Inform. Process. Lett., **7**, 1978, 175–179.

[GreS77] Green, P.J., and Sibson, R., "Computing Dirichlet Tessellations in the Plane," Computer Journal, **21**(2), 1977, 168–173.

[GuiS85] Guibas, Leonidas J., and Stolfi, Jorge, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," ACM Trans. on Graphics, **4**(2), April 1985, 74–123.

[LBDW92] Lavender, David, Bowyer, Adrian, Davenport, James, Wallis, Andrew, and Woodwark, John, "Voronoi Diagrams of Set-Theoretic Solid Models," CG&A, **12**(5), September 1992, 69–77.

[LeeP77] Lee, D.T., and Preparata, F.P., "Location of a Point in a Planar Subdivision and its Application," SIAM J. Comp., 6. 1977, 594–606.

[LinM96] Lin, Ming C., and Manocha, Dinesh, editors, *Applied Computational Geometry: Towards Geometric Engineering*, Springer Verlag, 1996.

[Lisc94] Lischinski, Dani, "Incremental Delaunay Triangulation," in [Heck94], 47–59.

[Mulm94] Mulmuley, Ketan, *Computational Geometry: An Introduction Through Randomized Algorithms*, Prentice-Hall, Inc., 1994.

[NarM95] Narkhede, Atul, and Manocha, Dinesh, "Fast Polygon Triangulation Based on Seidel's Algorithm," in [Paet95], 394–397.

[Orou94] O'Rourke, Joseph, *Computational Geometry in C*, Cambridge Univ. Press, 1994.

[PreS85] Preparata, Franco P., and Shamos, Michael I., *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[Shew96] Shewchuk, Jonathan Richard, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in [LinM96], 203–222.

[SuDr95] Su, Peter, and Drysdale, Robert L. Scot, "A Comparison of Sequential Delaunay Triangulation Algorithms," in *Proc. of the 11th Annual Symp. on Computational Geometry*, Vancouver, British Columbia, Canada, June 5–7, 1995, ACM Press, 61–70.

[ZalC99] Zalik, Borut, and Clapworthy, Gordon J., "A Universal Trapezoidation Algorithm for Planar Polygons," Computers & Graphics, **23**, 1999, 353–363.

Conics

[Blin87] Blinn, James F., "How Many Ways Can You Draw a Circle," CG&A, **7**(8), August 1987, 39–44.

[Blin88a] Blinn, James F., "Jim Blinn's Answer," CG&A, **8**(3), May 1988, 12.

[Galt89] Galton, Ian, "An Efficient Three-Point Arc Algorithm," CG&A, **9**(6), November 1989, 44–49.

[JoLH73] Jordan, B.W., Lennon, W.J., and Holm, B.D., "An Improved Algorithm for the Generation of Nonparametric Curves," IEEE Trans. Computers, Volume **C-22**, 1973, 1052–1060.

[Kap85] Kappel, M.R., "An Ellipse-Drawing Algorithm for Raster Displays," in [Earn85], 257–280.

[Pitt85] Pitteway, M.L.V., "Algorithms of Conic Generation," in [Earn85], 219–237.

[VanA84] Van Aken, J.R., "An Efficient Ellipse-Drawing Algorithm," CG&A, **4**(9), September 1984, 24–35.

[Wils87] Wilson, Peter R., "Conic Representations for Shape Description," CG&A, **7**(4), April 1987, 23–30.

Constructive Solid Geometry

[LaTH86] LaidLaw, David II, Trumbore, W. Benjamin, and Hughes, John F., "Constructive Solid Geometry for Polyhedral Objects," SIGGRAPH 86, **20**(4), August 1986, 161–168.

[RosV89] Rossignac, J., and Voelcker, H., "Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection, and Shading Algorithms," ACM TOG, **8**(1), January 1989, 51–87.

[ShaV91a] Shapiro, Vadim, and Vossler, Donald L., "Construction and Optimization of CSG Representation," CAD, **23**(1), January/February 1991, 4–20.

[ShaV91b] Shapiro, Vadim, and Vossler, Donald L., "Efficient CSG Representations of Two-dimensional Solids," Trans. ASME, Journal of Mechanical Design, **113**(3), September 1991, 292–305.

[ShaV93] Shapiro, Vadim, and Vossler, Donald L., "Separation for Boundary to CSG Conversion," ACM TOG, **12**(1), January 1993, 35–55.

Contours

[DLTW90] Dobkin, David P., Levy, Silvio V.F., Thurston, William P., and Wilks, Allan R., "Contour Tracing by Piecewise Linear Approximations," ACM TOG **9**(4), October 1990, 389–423.

[Dowd85] Dowd, P.A., "Algorithms for Three-dimensional Interpolation Between Planar Slices," in [Earn85], 531–554.

[FuKU77] Fuchs, H., Kedem, Z.M., and Uselton, S.P., "Optimal Surface Reconstruction from Planar Contours," CACM, **20**(10), October 1977, 693–702.

[GanD82] Ganapathy, S., and Dennehy, T.G., "A New General Triangulation Method for Planar Contours," SIGGRAPH 82, **16**(3), July 1982, 69–75.

[MeSS92] Meyers, David, Skinner, Shelley, and Sloan, Kenneth, "Surfaces from Contours," TOG, **11**(3), July 1992, 228–258.

[ParK96] Park, Hyungjun, and Kim, Kwangsoo, "Smooth Surface Approximation to Serial Cross-sections," CAD, **28**(12), December 1996, 995–1005.

[Preu86] Preusser, Albrecht, "Computing Area Filling Contours for Surfaces Defined by Piecewise Polynomials," CAGD, **3**(4), December 1986, 267–279.

[Sabi85] Sabin, M.A., "Contouring—The State of the Art," in [Earn85], 411–482.

[Sutc80] Sutcliffe, D., "Contouring over Rectangular and Skewed Rectangular Grids—An Introduction," in [Brod80], 39–62.

Convex Sets

[Rock70] Rockafellar, R. Tyrrell, *Convex Analysis*, Princeton University Press, Princeton, New Jersey, 1970.

[Vale64] Valentine, Frederick A., *Convex Sets*, McGraw-Hill Book Co., 1964.

Curvature

[Ande93] Andersson, Roger K.E., "Surfaces With Prescribed Curvature I," CAGD, **10**(5), October 1993, 431–452.

[KakG96] Kaklis, P.D., and Ginnis, A.I., "Sectional-Curvature Preserving Skinning Surfaces," CAGD, **13**(7), October 1996, 601–619.

[KrLM98] Krsek, P., Lukács, G., and Martin, R.R., "Algorithms for Computing Curvatures from Range Data," in [Crip98], 1–16.

[MeeW00] Meek, D.S., and Walton, D.J., "On Surface Normal and Gaussian Curvature Approximations Given Data Sampled from a Smooth Surface," CAGD, **17**(6), July 2000, 521–543.

[Miur00] Miura, Kenjiro T., "Unit Quaternion Integral Curve: A New Type of Fair Free-Form Curves," CAGD, **17**(1), January 2000, 39–58.

[Sapi92] Sapidis, Nickolas S., "Controlling the Curvature of a Quadratic Bézier Curve," CAGD, **9**(1), May 1992, 85–91.

[TheF97] Theisel, Holger, and Farin, Gerald, "The Curvature of Characteristic Curves on Surfaces," CG&A, **17**(6), November/December 1997, 88–96.

[Woll00] Wollmann, Christian, "Estimation of the Principle Curvatures of Approximated Surfaces," CAGD, **17**(7), August 2000, 621–630.

[WolT92] Wolter, Franz-Erich, and Tuohy, Séamus T., "Curvature Computations for Degenerate Surface Patches," CAGD, **9**(4), September 1992, 241–270.

[Ye96] Ye, Xiuzi, "The Gaussian and Mean Curvature Criteria for Curvature Continuity Between Surfaces," CAGD, **13**(6), August 1996, 549–567.

Curve Algorithms

[Figu95] de Figueiredo, Luiz H., "Adaptive Sampling of Parametric Curves," in [Paet95], 173–178.

[Grav95] Gravesen, Jens, "The Length of Bézier Curves," in [Paet95], 199–205.

[GueP90] Guenter, Brian, and Parent, Richard, "Computing the Arc Length of Parametric Curves," CG&A, **10**(3), May 1990, 72–78.

[KopM83] Koparkar, P.A., and Mudur, S.P., "A New Class of Algorithms for the Processing of Parametric Curves," CAD, **15**(1), January 1983, 41–45.

[LiCr97] Li, Yong-Ming, and Cripps, Robert J., "Identification of Inflection Points and Cusps on Rational Curves," CAGD, **14**(5), June 1997, 491–497.

[ManC90] Manocha, Dinesh, and Canny, John F., "Polynomial Parameterizations for Rational Curves," in Ferrari, Leonard A., and de Figueiredo, Rui J.P., editors, *Curves and Surfaces in Computer Vision and Graphics*, Proceedings SPIE – The International Society for Optical Engineering, February 13–15, 1990, Santa Clara, CA, Volume 1251, 151–162.

[ManC92a] Manocha, Dinesh, and Canny, John F., "Detecting Cusps and Inflection Points in Curves," CAGD, **9**(1), May 1992, 1–24.

[RouB96a] Roulier, John A., Piper, Bruce, "Prescribing the Length of Parametric Curves," CAGD, **13**(1), February 1996, 3–22.

[RouB96b] Roulier, John A., Piper, Bruce, "Prescribing the Length of Rational Bézier Curves," CAGD, **13**(1), February 1996, 23–43.

[ShaT82] Sharpe, Richard J., and Thorne, Richard W., "Numerical Method for Extracting an Arc Length Parameterization from Parametric Curves," CAD, **14**(2), March 1982, 79–81.

[WolF97] Wolters, Hans J., and Farin, Gerald, "Geometric Curve Approximation," CAGD, **14**(6), August 1997, 499–513.

Cyclides

[AlbD97] Albrecht, Gudrun, and Degen, Wendelin L.F., "Construction of Bézier Rectangles and Triangles on the Symmetric Dupin Horn Cyclide by Means of Inversion," CAGD, **14**(4), May 1997, 349–375.

[Boeh89] Boehm, Wolfgang, "Some Remarks on Cyclides in Solid Modeling," in [StrS89], 247–252.

[Boeh90] Boehm, Wolfgang, "On Cyclides in Geometric Modeling," CAGD, **7**(1–4), June 1990, 243–255.

[ChDH89] Chandru, V., Dutta, D., and Hoffmann, C.M., "On the Geometry of Dupin Cyclides," The Visual Computer, **5**(5), October 1989, 277–290.

[Dege94] Degen, W.L.F., "Generalized Cyclides for Use in CAGD," in [Bowy94], 349–363.

[Dege98] Degen, W.L.F., "On the Origin of SuperCyclides," in [Crip98], 297–312.

[DuMP93] Dutta, Debasish, Martin, Ralph R., and Pratt, Michael J., "Cyclides in Surface and Solid Modeling," CG&A, **13**(1), January 1993, 53–59.

[John93] Johnstone, John K., "A New Intersection Algorithm for Cyclides and Swept Surfaces Using Circle Decomposition," CAGD, **10**(1), February 1993, 1–24.

[KraM00] Krasauskas, R., and Mäurer, C., "Studying Cyclides with Laguerre Geometry," CAGD, **17**(2), February 2000, 101–126.

[Mart82] Martin, R.R., "Principal Patches for Computational Geometry," PhD thesis, Engineering Department, Cambridge University, U.K., 1982.

[MaPS86] Martin, R.R., de Pont, J., and Sharrock, T.J., "Cyclide Surfaces in Computer Aided Design," in [Greg86], 253–267.

[PalB98] Paluszny, Marco, and Boehm, Wolfgang, "General Cyclides," CAGD, **15**(7), July 1998, 699–710.

[Prat90] Pratt, M.J., "Cyclides in Computer Aided Geometric Design," CAGD, **7**(1–4), June 1990, 221–242.

[Prat95] Pratt, M.J., "Cyclides in Computer Aided Geometric Design II," CAGD, **12**(2), March 1995, 131–152.

[Prat96] Pratt, M.J., "Dupin Cyclides and Supercyclides," in [Mull96], 43–66.

[Prat97] Pratt, M.J., "Quartic Supercyclides I: Basic Theory," CAGD, **14**(7), September 1997, 671–692.

Differential Geometry

[Bish75] Bishop, R.L., "There Is More than One Way to Frame a Curve," Am. Math. Monthly, **82**(3), March 1975, 246–251.

[Bloo90] Bloomenthal, Jules, "Calculation of Reference Frames Along a Space Curve," in [Glas90], 567–571.

[Fors12] Forsyth, A.R., *Lectures on Differential Geometry of Curves and Surfaces*, Cambridge Univ. Press, 1912.

[Gray98] Gray, Alfred, *Modern Differential Geometry of Curves and Surfaces with MATHEMATICA*, 2nd Edition, CRC Press, 1998.

[YuPM00] Yu, Guoxin, Patrikalakis, Nicholas M., and Maekawa, Takashi, "Optimal Development of Doubly Curved Surfaces," CAGD, **17**(6), July 2000, 545–577.

Digital Image Processing

[Glas95] Glassner, Andrew S., *Principles of Digital Image Synthesis*, Volume 1 & 2, Morgan Kaufmann Publ., Inc., 1995.

[GonW87] Gonzalez, Rafael C., and Wintz, Paul, *Digital Image Processing*, 2nd Edition, Addison-Wesley Publ. Co., 1987.

[RosK76] Rosenfeld, Azriel, and Kak, Avinash C., *Digital Picture Processing*, Academic Press, 1976.

Engineering Applications

[Crai88] Craig, John J., "CompStation Design: November 88 Report," SILMA Inc. (now part of Adept Technology, Inc.), 1988.

[RosG64] Rosato, D.V., and Grove, C.S., Jr., *Filament Winding: Its Development, Manufacture, Applications, and Design*, Interscience Publishers, 1964.

[Schw97] Schwartz, Mel M., *Composite Materials, Volume II: Processing, Fabrication, and Applications*, Prentice Hall PTR, 1997.

Finite Element Method

[Arms94] Armstrong, Cecil G., "Modelling Requirements for Finite-Element Analysis," CAD, **26**(7), July 1994, 573–578.

[Buch95] Buchanan, George R., *Finite Element Analysis*, Schaum's Outline Series, McGraw-Hill, Inc., 1995.

[Heck93] Heckbert, Paul S., "Introduction to Finite Element Methods," Course Notes, Volume 42, SIGGRAPH 93, August 1993.

[HoLe88] Ho-Le, K., "Finite Element Mesh Generation Methods: A Review and Classification," CAD, **20**(1), January/February 1988, 27–38.

[John87] Johnson, Claes, *Numerical Solutions of Partial Differential Equations by the Finite Element Method*, Cambridge Univ. Press, 1987.

[MitW78] Mitchell, A.R., and Wait, R., *The Finite Element Method in Partial Differential Equations*, John Wiley & Sons, Inc., 1978.

[OttP92] Ottosen, Niels, and Petersson, Hans, *Introduction to the Finite Element Method*, Prentice Hall, 1992.

[PepH92] Pepper, Darrell W., and Heinrich, Juan C., *The Finite Element Method: Basic Concepts and Applications*, Hemisphere Publ. Corp., 1992.

Fourier Series and Transforms

[Brac86] Bracewell, Ronald N., *The Fourier Transform and Its Applications*, 2nd Edition, McGraw-Hill, 1986.

[CooT65] Cooley, James W., and Tukey, John W., "An Algorithm for the Machine Calculation of Complex Fourier Series," Mathematics of Computation, **19**(90), April 1965, 297–301.

[Four95] Fournier, Alain, Organizer, *Wavelets and Their Application to Computer Graphics*, Course Notes, Volume 26, SIGGRAPH, August 1995.

[Frie63] Friedman, Avner, *Generalized Functions and Partial Differential Equations*, Prentice-Hall, Inc., 1963.

[Glas99] Glassner, Andrew., "Fourier Polygons," CG&A, **19**(1), January/February 1999, 84–91.

[GomV98] Gomez, Jonas, and Velho, Luiz, Organizers and Lecturers, *From Fourier Analysis To Wavelets*, Course Notes, Volume 6, SIGGRAPH, July 1998.

[LoDW97] Lounsbery, Michael, DeRose, Tony D., and Warren, Joe, "Multiresolution Analysis for Surfaces of Arbitrary Topological Type," ACM TOG, **16**(1), January 1997, 34–73.

[Seel66] Seeley, Robert, *An Introduction to Fourier Series and Integrals*, W.A. Benjamin, Inc., 1966.

[Widd71] Widder, D.V., *An Introduction to Transform Theory*, Academic Press, 1971.

Fractals

[AlSY97] Alligood, Kathleen T., Sauer, Tim D., and Yorke, James A., *Chaos: An Introduction to Dynamical Systems*, Springer-Verlag, 1997.

[BBGDS92] Banks, J., Brooks, J., Gairns, G., David, G., and Stacey, R., "On Devaney's Definition of Chaos," Amer. Math. Monthly, **99**, 1992, 332–334.

[Barn87] Barnsley, Michael F., "Fractal Modelling of Real World Images," Course Notes, Volume 15, SIGGRAPH, July 1998.

[Barn88] Barnsley, Michael F., *Fractals Everywhere*, Academic Press, 1988.

[Cran95] Crannell, Annalisa, "The Role of Transitivity in Devaney's Definition of Chaos," Amer. Math. Monthly, **102**(9), 1995, 788–793.

[Deva86] Devaney, Robert L., *An Introduction to Chaotic Dynamical Systems*, The Benjamin/Cummings Publ. Co., 1986.

[DevK89] Devaney, Robert L., and Keen, Linda, editors, *Chaos and Fractals: The Mathematics Behind the Computer Graphics*, Proceedings of Symposia in Applied Mathematics, Volume 39, AMS, 1989.

[Falc85] Falconer, K.J., *The Geometry of Fractal Sets*, Cambridge Univ. Press, 1985.

[Fede69] Federer, H., *Geometric Measure Theory*, Springer-Verlag, 1969.

[FoFC82] Fournier, Alain, Fussell, Don, and Carpenter, Loren, "Computer Rendering of Stochastic Models," CACM, **25**(6), June 1982, 371–384.

[Glas92] Glassner, Andrew S., "Geometric Substitutions: A Tutorial," CG&A, **12**(1), January 1992, 22–36.

[Lind68] Lindenmayer, Aristid, "Mathematical Models for Cellular Interactions in Development, Parts I and II, J. of Theoretical Biology, **18**, 1968, 280–315.

[Mand83] Mandelbrot, Benoit B., *The Fractal Geometry of Nature*, W.H. Freeman and Co., 1983.
[Reev83] Reeves, William T., "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects," ACM TOG, **2**(2), 91–108.
[ReBl85] Reeves, William T., and Blau, R., "Approximate and Probabilistic Algorithms for Shading and Rendering Particle Systems," SIGGRAPH 85, **19**(3), July 1985, 313–322.
[Smit84] Smith, Alvy Ray, "Plants, Fractals, and Formal Languages," Course Notes, Volume 15, SIGGRAPH 84, July 1984, 1–10.
[VelB94] Vellekoop, M., and Berglund, R., "On Intervals, Transitivity = Chaos," Amer. Math. Monthly, **101**, 1994, 353–355.

General Computer Graphics

[Ange00] Angel, Edward, *Interactive Computer Graphics: A Top-down Approach with OpenGL*, 2nd Edition., Addison-Wesley Longman, Inc., 2000.
[BalB82] Ballard, D.H., and Brown, C.M., *Computer Vision*, Prentice-Hall, 1982.
[BeaB82] Beatty, J.C., and Booth, K.S., editors, *Tutorial: Computer Graphics*, 2nd Edition, IEEE Comp. Society Press, 1982.
[Boot79] Booth, Kellogg S., *Tutorial: Computer Graphics*, IEEE Computer Society, 1979.
[Earn85] Earnshaw, R.A., editor, *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, 1985.
[FVFH90] Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F., *Computer Graphics: Principles and* Practice, 2nd Edition, Addison-Wesley Publ. Co., 1990.
[Free80] Freeman, H., editor, *Tutorial and Selected Readings in Interactive Computer Graphics*, IEEE Comp. Society Press, 1980.
[Hill90] Hill, F.S. Jr., *Computer Graphics*, MacMillan Publ. Co., 1990.
[Hill01] Hill, F.S. Jr., *Computer Graphics Using OpenGL*, 2nd Edition, Prentice Hall, 2001.
[JGMH88] Joy, Kenneth I., Grant, Charles W., Max, Nelson L., and Hatfield, Lansing, *Tutorial: Computer Graphics: Image Synthesis*, IEEE Computer Society Press, 1988.
[MagT87] Magnenat-Thalmann, Nadia, and Thalmann, Daniel, *Image Synthesis: Theory and Practice*, Springer-Verlag, 1987.
[Miel91] Mielke, Bruce, *Integrated Computer Graphics*, West Publishing Co., 1991.
[PokG89] Pokorny, Cornel K., and Gerald, Curtis F., *Computer Graphics: The Principles Behind the Ar nd Science*, Franklin, Beedle and Associates, 1989.
[Roge98] Rogers, David F., *Procedural Elements for Computer Graphics*, 2nd Edition, McGraw-Hill, 1998.
[Watt90] Watt, A., *Fundamentals of Three Dimensional Computer Graphics*, Addison-Wesley Publ. Co., 1990.

Geodesics

[AgHK00] Agarwal, Pankaj K., Har-Peled, Sariel, and Karia, Meetesh, "Computing Approximate Shortest Paths on Convex Polytopes," in *Proc. of the 16th Annual Symp. on Computational Geometry*, Hong Kong, June 12–14, 2000, ACM Press, 270–279.
[CheH90] Chen, Jindong, and Han, Yije, "Shortest Paths on a Polyhedron," in *Proc. of the 6th Annual Symp. on Computational Geometry*, Berkeley, California, June 6–8, 1990, ACM Press, 360–369.
[Kapo99] Kapoor, S., "Efficient Computation of Geodesic Shortest Paths," *Proc. 31st Annual ACM Symp. Theory of Comput.*, Atlanta, Georgia, 1999, 770–779.
[KSHS03] Kumar, G.V.V., Ravi, Srinivasan, Prabha, Holla, V., Devaraja, Shastry, K.G., and Prakash, B.G., "Geodesic Curve Computations on Surfaces," CAGD, **20**(2), May 2003, 119–133.
[MiMP87] Mitchell, Joseph S.B., Mount, David M., and Papadimitrious, Christos H., "The Discrete Geodesic Problem," SIAM J. Computing, **16**(4), August 1987, 647–668.
[ShaS86] Sharir, M., and Schorr, A., "On Shortest Paths in Polyhedral Spaces," SIAM J. Computing, **15**(1), February 1986, 193–215.

Geometric Modeling Books

[AllG90] Allgower, E.L., and Georg, K., *Numerical Continuation Methods: An Introduction*, Springer Verlag, 1990.
[Barn92] Barnhill, Robert E., editor, *Geometry Processing for Design and Manufacturing*, SIAM, 1992.
[BarB83] Barnhill, Robert E., and Boehm, Wolfgang, editors, *Surfaces in Computer Aided Geometric Design*, North-Holland, 1983.

[BarR74] Barnhill, Robert E., and Riesenfeld, Richard F., editors, *Computer Aided Geometric Design*, Academic Press, 1974.

[Beac91] Beach, Robert C., *An Introduction to the Curves and Surfaces of Computer-Aided Design*, Van Nostrand Reinhold, 1991.

[Bezi72] Bézier, P., *Numerical Control: Mathematics and Applications*, John Wiley & Sons, Inc, 1972.

[BoeP94] Boehm, Wolfgang, and Prautzsch, Hartmut, *Geometric Concepts for Geometric Design*, A K Peters, Ltd., 1994.

[Bowy94] Bowyer, Adrian, editor, *Computer-aided Surface Geometry and Design: The Mathematics of Surfaces IV*, Clarendon Press, Oxford, 1994.

[Brod80] Brodlie, K. W., *Mathematical Methods in Computer Graphics and Design*, Academic Press, 1980.

[BroA99] Bronsvoort, Willem F., and Anderson, David C., editors, *Proceedings of Fifth Symposium on Solid Modeling and Applications*, ACM Press, June 9–11, 1999.

[Chiy88] Chiyokura, Hiroaki, *Solid Modeling with DESIGNBASE: Theory and Implementation*, Addison-Wesley Publ. Co., 1988.

[Crip98] Cripps, Robert, editor, *The Mathematics of Surfaces VIII*, Information Geometers, 1998.

[Earn88] Earnshaw, Rae, A., editor, *Theoretical Foundations of Computer Graphics and CAD*, Springer-Verlag, 1988.

[Fari87] Farin, Gerald, editor, *Geometric Modeling: Algorithms and New Trends*, SIAM, 1987.

[Fari97] Farin, Gerald, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, 4th Edition, Academic Press, Inc., 1997.

[FauP79] Faux, I.D., and Pratt, M.J., *Computational Geometry for Design and Manufacture*, Halsted Press, a division of John Wiley & Sons, Inc., 1979.

[Fish94] Fisher, R.B., editor, *Computer-aided Surface Geometry and Design: The Mathematics of Surfaces V*, Clarendon Press, Oxford, 1994.

[Gall00] Gallier, Jean, *Curves and Surfaces in Geometric Modeling: Theory and Algorithms*, Morgan Kaufmann Publ., 2000.

[Gass83] Gasson, Peter C., *Geometry of Spatial Forms*, Ellis Horwood, a division of John Wiley and Sons, 1983.

[GHSV93] de Miranda Gomez, Jonas, Hoffmann, Christoph, Shapiro, Vadim, and Velho, Luiz, Organizers, *Modeling in Computer Graphics*, Course Notes, Volume 40, SIGGRAPH, August 1993.

[Greg86] Gregory, J.A., editor, *The Mathematics of Surfaces*, Clarendon Press, Oxford, 1986.

[HaFN95] Hagen, H., Farin, G., and Noltemeier, H., editors, *Geometric Modelling: Dagstuhl 1993*, Computing Suppl. 10, Springer-Verlag, 1995.

[Hand89] Handscomb, D.C., editor, *Computer-aided Surface Geometry and Design: The Mathematics of Surfaces III*, Clarendon Press, Oxford, 1989.

[Hoff89] Hoffmann, Christoph M., *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann Publ. Inc., 1989.

[HofB97] Hoffmann, Christoph, and Bronsvort, Wim, editors, *Proceedings of Fourth Symposium on Solid, Modeling Foundations and Applications*, ACM, May 14–16, 1997.

[HofR95] Hoffmann, Chris, and Rossignac, Jarek, editors, *Proceedings of Third Symposium on Solid Modeling and Applications*, ACM Press, May 17–19, 1995.

[HosL93] Hoschek, Josef, and Lasser, Dieter, *Fundamentals of Computer Aided Geometric Design*, A. K. Peters, Wellesley, Mass., 1993.

[LanS86] Lancaster, Peter, and Salkauskas, Kestutis, *Curve and Surface Fitting: An Introduction*, Academic Press, 1986.

[LorW86] Lord, E. A., and Wilson, C. B., *The Mathematical Description of Shape and Form*, Ellis Horwood Limited, 1986.

[LycS89] Lyche, Tom, and Schumaker, Larry L., *Mathematical Methods in Computer Aided Geometric Design*, Acacemic Press, Inc., 1989.

[Mant88] Mäntylä, Martti, *An Introduction to Solid Modeling*, Computer Science Press, 1988.

[Mart87] Martin, R.R., editor, *The Mathematics of Surfaces II*, Clarendon Press, Oxford, 1987.

[Mort85] Mortenson, Michael E., *Geometric Modeling*, John Wiley & Sons, Inc., 1985.

[Mull96] Mullineux, G., editor, *The Mathematics of Surfaces VI*, Oxford Univ. Press, 1996.

[RogE90] Rogers, David F., and Earnshaw, Rae, A., editors, *Computer Graphics Techniques: Theory and Practice*, Springer-Verlag, 1990.

[RosT91] Rossignac, Jaroslaw, and Turner, Joshua, editors, *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, ACM Press, June 5–7, 1991.

[Salo99] Salomon, David, *Computer Graphics and Geometric Modeling*, Springer-Verlag, 1999.
[ShaM95] Shah, Jami J., and Mäntylä, Martti, *Parametric and Feature-Based CAD/CAM: Concepts, Techniques, and Applications*, John Wiley & Sons, Inc., 1995.
[Snyd92] Snyder, John M., *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*, Academic Press, Inc., 1992.
[StrS89] Strasser, Wolfgang, and Seidel, Hans-Peter, editors, *Theory and Practice of Geometric Modeling*, Springer Verlag, 1989.
[SuLi89] Su, Bu-qing, and Liu, Ding-yuan, *Computational Geometry: Curve and Surface Modeling*, Academic Press, Inc., 1989.

Geometric Modeling Papers

[AllG93] Allgower, E.L., and Georg, K., "Continuation and Path Following," Acta Numerica, 1993, 1–64.
[Amat96] Amato, Nancy M., "Equipping CAD/CAM Systems with Geometric Intelligence," ACM Computing Surveys, **28**(4es), December 1996, Article #17.
[BagW95] Bagali, Siddarameshwar, and Waggenspack, Warren N., Jr., "A Shortest Path Approach to Wireframe to Solid Model Conversion," in [HofR95], 339–349.
[Baum72] Baumgart, B.G., "Winged-edge Polyhedron Representation," Technical Report STAN-CS-320, Computer Science Dept., Stanford University, Stanford, CA, 1972.
[Baum75] Baumgart, B.G., "A Polyhedron Representation for Computer Vision," NCC 75, 589–596.
[Bézi71] Bézier, P.E., "Example of an Existing System in the Motor Industry," Proc. Royal Soc. London Ser. A, **321**, 1971, 207–218.
[Bier95] Bieri, H., "Nef Polyhedra: A Brief Introduction," in [HaFN95], 43–60.
[Binf71] Binford, T.O., "Visual Perception by Computer," in Proceedings of the IEEE Conference on Systems and Control, Miami, Florida, December 1971.
[Blin82] Blinn, James F., "A Generalization of Algebraic Surface Drawing," ACM TOG, **1**(3), 235–256.
[BlSS89] Blum, Lenore, Shub, Mike, and Smale, Stephen, "On a Theory of Computation and Complexity over the Real Numbers: NP-Completeness, Recursive Functions and Universal Machines," Bulletin of the AMS, **21**(1), July 1989, 1–46.
[BoFK84] Böhm, Wolfgang, Farin, Gerald, and Kahmann, Jürgen, "A Survey of Curve and Surface Methods in CAGD," CAGD, **1**(1), January 1984, 1–60.
[Boeh87] Boehm, Wolfgang, "Smooth Curves and Surfaces," in [Fari87], 175–184.
[BrHS80] Braid, I.C., Hillyard, R.C., and Stroud, I.A., "Stepwise Construction of Polyhedra in Geometric Modeling," in [Brod80], 123–141.
[CavM89] Cavaretta, Alfred, and Micchelli, Charles A., "The Design of Curves and Surfaces by Subdivision Algorithms," in [LycS89], 115–153.
[Coon67] Coons, S.A., "Surfaces for Computer Aided Design of Space Forms," MIT Project Mac, TR-41, MIT, Cambridge, MA, June 1967.
[DehZ91] DeHaemer, Jr., M.J., and Zyda, M.J., "Simplification of Objects Rendered by Polygonal Approximations," Computer & Graphics, **15**, 1991, 175–184.
[EdaL99] Edalat, Abbas, and Lieutier, André, "Foundation of a Computable Solid Modeling," in [BroA99], 278–284.
[EdeM90] Edelsbrunner, Herbert, and Mücke, Ernst Peter, "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms," ACM TOG, **9**(1), January 1990, 66–104.
[Elsa83] Elsaesser, Fritz, "Surfaces and Their Applications at Opel," in [BarB83], 157–162.
[Fari83] Farin, Gerald, "Some Aspects of Car Body Design at Daimler-Benz," in [BarB83], 93–98.
[Ferg64] Ferguson, J.C., "Multivariable Curve Interpolation," J. ACM, **11**, 1964, 221–228.
[FilB89] Filip, Daniel J., and Ball, Thomas W., "Procedurally Representing Lofted Surfaces," CG&A, **9**(6), November 1989, 27–33.
[FiMM86] Filip, D., Magedson, R., and Markot, R., "Surface Algorithms Using Bounds on Derivatives," CAGD, **3**(4), 1986, 295–311.
[FolR93] Foley, Thomas, and Rockwood, Alyn, Organizers, *Curve and Surface Design: From Geometry to Applications*, Course Notes, Volume 82, SIGGRAPH 93, August 1993.
[Fort95] Fortune, Steven, "Polyhedral Modelling with Exact Arithmetic," in [HofR95], 225–233.
[GeCG99] Ge, Jian-Xin, Chou, Shang-Ching, and Gao, Xiao-Shan, "Geometric Constraint Satisfaction Using Optimization Methods," CAD, **31**(14), December 1999, 867–879.

[Gord69] Gordon, William J., "Distributive Lattices and the Approximation of Multivariate Functions," in *Approximations with Special Emphasis on Spline Functions*, edited by I.J. Schoenberg, Academic Press, 1969, 223–277.

[Gord71] Gordon, William J., "Blending Function Methods of Bivariate and Multivariate Interpolation and Approximation," SIAM J. Numer. Anal., **8**(1), 1971, 158–177.

[GorR74a] Gordon, W.J., and Riesenfeld, R.F., "Bernstein-Bézier Methods for the Computer Aided Design of Free-form Curves and Surfaces, J.ACM, **21**, 1974, 293–310.

[Greg89] Gregory, John A., "Geometric Continuity," in [LycS89], 353–371.

[Heck97] Heckbert, Paul S., Organizer, *Multiresolution Surface Modeling*, Course Notes, Volume 27, SIGGRAPH 97, August 1997.

[HecG97] Heckbert, Paul S., and Garland, Michael, "Survey of Polygonal Surface Simplification Algorithms," in [Heck97].

[Hoch83] Hochfeld, Hans-Joachim, "Surface Description in the Application at Volkswagen," in [BarB83], 35–42.

[HoHK89] Hoffmann, Christoph M., Hopcroft, John E., and Karasick, Michael S., "Robust Set Operations on Polyhedral Solids," CG&A, **9**(6), 1989, 50–59.

[Hopp96] Hoppe, Hugues., "Progressive Meshes," SIGGRAPH 96, August 1996, 99–108.

[HuaM02] Huang, J., and Menq, C.H., "Combinatorial Manifold Mesh Reconstruction and Optimization from Unorganized Points with Arbitrary Topology," CAD, **34**(2), February 2002, 149–165.

[HutH96] Hutchinson, Dave, and Hewitt, Terry, "Rapidly Visualizing Isophotes," J. of Graphics Tool, **1**(3), 1996, 7–12.

[JiMa97] Ji, Qiang, and Marefat, Michael M., "Machine Interpretation of CAD Data for Manufacturing Applications," ACM Computing Surveys, **29**(3), September 1997, 264–311.

[Just92] Juster, N.P., "Modelling and Representation of Dimensions and Tolerances: A Survey," CAD, **24**(1) January 1992, 3–17.

[KimK03] Kim, Ku-Jin, "Minimum Distance Between a Canal Surface and a Simple Surface," CAD, **35**(10), September 2003, 871–879.

[Klas80] Klass, R., "Correction of Local Surface Irregularities Using Reflection Lines," CAD, **12**(2), 1980, 73–77.

[Kobb96] Kobbelt, Leif, "A Variational Approach to Subdivision," CAGD, **13**(8), November 1996, 743–761.

[Kost91] Kosters, "Curvature-dependent Parameterization of Curves and Surfaces," CAD, **23**(8), October 1991, 569–578.

[Kypr80] Kyprianou, L.K., "Shape Classification in Computer Aided Design," Ph.D. Dissertation, University of Cambridge, Cambridge, England.

[LanR80] Lane, J.M., and Riesenfeld, R.F., "A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces," IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-2(1), January 1980, 35–46.

[LiON02] Li, W.D., Ong, S.K., and Nee, A.Y.C., "Recognizing Manufacturing Features from a Design-by-Feature Model," CAD, **34**(11), September 2002, 849–868.

[LiHS02] Li, Yan-Tao, Hu, Shi-Min, and Sun, Jia-Guang, "A Constructive Approach to Solving 3-D Geometric Constraint Systems Using Dependence Analysis," CAD, 34(2), February 2002, 97–108.

[LicS87] Lichten, Larry, and Samek, Marcel, "Integrating Sculptured Surfaces into a Polyhedral Solid Modeling System," in [Fari87], 109–121.

[LWZL02] Liu, G.H., Wong, Y.S., Zhang, Y.F., and Loh, H.T., "Adaptive Fairing of Digitized Point Data with Discrete Curvature," CAD, **34**(4), April 2002, 309–320.

[Lueb01] Luebke, David P., "A Developer's Survey of Polygonal Simplification Algorithms," CG&A, **21**(3), May/June 2001, 24–35.

[MäNS96] Mäntylä, Martti, Nau, Dana, and Shah, Jami, "Challenges in Feature-based Manufacturing Research," CACM, **39**(2), February 1996, 77–85.

[Mart94] Martin, R.R., "The Geometry of the Helical Canal Surface," in [Bowy94], 17–32.

[MarS89] Martin, R.R., and Stephenson, P.C., "Swept Volumes in Solid Modellers," in [Hand89], 391–404.

[Mill86] Miller, James R., "Sculptured Surfaces in Solid Models: Issues and Alternative Approaches," CG&A, **6**(12), December 1986, 37–48.

[Nasr87] Nasri, Ahmad. H., "Polyhedral Subdivision Methods for Free-Form Surfaces," ACM TOG, **6**(1), January 1987, 29–73.

[NiBl94] Ni, Xiujun, and Bloor, M. Susan, "Performance Evaluation of Boundary Data Structures," CG&A, **14**(6), November 1994, 66–77.

[NowR83] Nowacki, Horst, and Reese, Dirk, "Design and Fairing of Ship Surfaces," in [BarB83], 121–134.

[OckS84] Ocken, S., and Schwartz, J.T., "Precise Implementation of CAD Primitives Using Rational Parametrizations of Standard Surfaces," in Pickett, M.S., and Boyse, J.W., editors, *Solid Modeling by Computers: From Theory to Applications*, Plenum Press, 1984, 259–273.

[PaPV95] Paoluzzi, Alberto, Pascucci, Valerio, and Vicentino, Michele, "Geometric Programming: A Programming Approach to Geometric Design," ACM TOG, **14**(3), July 1995, 266–306.

[PePR99] Peternell, Martin, Pottmann, Helmut, and Ravani, Bahram, "On the Computational Geometry of Ruled Surfaces," CAD, **31**(1), January 1999, 17–32.

[Podg02] Podgorelec, D., "A New Constructive Approach to Constraint-Based Geometric Design," CAD, **34**(11), September 2002, 769–785.

[Pösc84] Pöschl, T., "Detecting Surface Irregularities Using Isophotes," CAGD, **1**(2), 1984, 163–168.

[Prat87a] Pratt, M.J., "Form Features and Their Applications in Solid Modeling," Course Notes, Volume 26, SIGGRAPH 87, July 1987.

[Prat87b] Pratt, M.J., "Recent Research in Form Features," Course Notes, Volume 26, SIGGRAPH 87, July 1987.

[PraW85] Pratt, M.J., and Wilson, P.R., "Requirements for the Support of Form Features in a Solid Modeling System," Report No. R-85-ASPP-01, CAM-I, Arlington, Texas.

[RanR91] Rando, T., and Roulier, J.A., "Designing Faired Parametric Surfaces," CAD, **23**, 1991, 492–497.

[RenE03] Renner, Gábor, and Ekárt, Anikó, "Genetic Algorithms in Computer Aided Design," CAD, **35**(8), July 2003, 709–726.

[Requ80] Requicha, A.A.G., "Representations for Rigid Solids: Theory, Methods, and Systems," ACM Computing Surveys, **12**(4), December 1980, 437–464.

[Requ96] Requicha, Aristides A.G., "Geometric Reasoning for Intelligent Manufacturing," CACM, **39**(2), February 1996, 71–76.

[ReqV82] Requicha, A.A.G., and Voelcker, H.B., "Solid Modeling: A Historical Summary and Contemporary Assessment," CG&A, **2**(2), March 1982, 9–24.

[ReqV83] Requicha, A.A.G., and Voelcker, H.B., "Solid Modeling: Current Status and Research Directions," CG&A, **3**(7), October 1983, 25–37.

[ReqV85] Requicha, A.A.G., and Voelcker, H.B., "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," Proc. of the IEEE, **73**(1), January 1985, 30–44.

[Roll95] Roller, D., "Solid Modeling with Constrained Form Features," in [HaFN95], 275–284.

[Sabi90] Sabin, Malcolm, "Sculptured Surface Definitions—A Historical Survey," in [RogE90], 285–337.

[SaRE76] Samuel, N.M., Requicha, A.A.G., and Elkind, S.A., "Methodology and Results of an Industrial Part Survey," TM-21, Production Automation Project, Univ. of Rochester, July 1976.

[Sede87] Sederberg, Thomas W., "Algebraic Geometry for Surface and Solid Modeling," in [Fari87], 29–42.

[SeWZ89] Sederberg, Thomas W., White, S.C., and Zundel, A.K., "Fat Arcs: A Bounding Region with Cubic Convergence," CAGD, **6**, 1989, 205–218.

[SéCM95] Séquin, C.H., Chang, P.-Y., and Moreton, H.P., "Scale-Invariant Functionals for Smooth Curves and Surfaces," in [HaFN95], 303–321.

[Shap91] Shapiro, Vadim, "Representations of Semi-Algebraic Sets in Finite Algebras Generated by Space Decompositions," Ph.D. thesis, Cornell University, Cornell Programmable Automation, Ithaca, NY, 1991.

[ShaV95] Shapiro, Vadim, and Vossler, Donald L., "What Is a Parametric Family of Solids?" in [HofR95], 43–54.

[ShiS98] Shin, Byeong-Seok, and Shin, Yeong Gil, "Fast 3D Solid Model Reconstruction from Orthographic Views," CAD, **30**(1), January 1998, 63–76.

[ShiK91] Shinagawa, Yoshihisa, and Kunii, Tosiyasu L., "Constructing a Reeb Graph Automatically from Cross Sections," CG&A, **11**(6), November 1991, 44–52.

[ShKK91] Shinagawa, Yoshihisa, Kunii, Tosiyasu L., and Kergosieu, Yannick L., "Surface Coding Based on Morse Theory," CG&A, **11**(5), September 1991, 66–78.

[SodT94] Sodhi, Rajneeet, and Turner, Joshua U., "Towards Modelling of Assemblies for Product Design," CAD, **26**(2),February 1994, 85–97.

[SuHH99] Suri, S., Hubbard, P.M., and Hughes, J.F., "Analyzing Bounding Boxes for Object Intersection," TOG, **18**(3), July 1999, 257–277.

[Tilo80] Tilove, R.B., "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," IEEE Trans. on Computer, **C-29**(10), October 1980, 874–883.

[VanW96] Van Overveld, C.W.A.M., and Wyvill, B., "Banishing Bad Buckling," J. of Graphics Tools, **1**(3), 1996, 13–28.

[VSBJ98] Volpin, O., Sheffer, A., Bercovier, M., and Joskowicz, L., "Mesh Simplification with Smooth Surface Reconstruction," CAD, **30**(11), September 1998, 875–882.

[Wats89] Watson, L.T., "Globally Convergen Homotopy Methods: A Tutorial," Appl. Math. Comput., **31**, 1989, 369–396.

[Weil85] Weiler, Kevin, "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments," CG&A, **5**(1), January 1985, 21–40.

[WooT85] Woo, T.C., "A Combinatorial Analysis of Boundary Data Structure Schemata," CG&A, **5**(3), March 1985, 19–27.

[Yu92] Yu, Jiaxun, "Exact Arithmetic Solid Modeling," PhD thesis, Technical Report CSD-TR-92-037, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, June 1992.

Graphical User Interfaces

[Micr94] Microsoft Corp., *The Windows Interface Guidelines for Software Design*, Microsoft Press, 1995.

[Pedd92] Peddie, Jon, *Graphical User Interfaces and Graphic Standards*, McGraw-Hill, Inc., 1992.

Graphics Pipeline

[Blin88b] Blinn, James F., "Where Am I? What Am I Looking At," CG&A, **8**(4), July 1988.

[Blin91a] Blinn, James F., "A Trip Down the Graphics Pipeline: Line Clipping," CG&A, **11**(1), January 1991.

[Blin91b] Blinn, James F., "A Trip Down the Graphics Pipeline: Pixel Coordinates," CG&A, **11**(4), July 1991.

[Blin91c] Blinn, James F., "A Trip Down the Graphics Pipeline: Subpixelic Particles," CG&A, **11**(5), September 1991.

[Blin92] Blinn, James F., "A Trip Down the Graphics Pipeline: Grandpa, What Does 'Viewport' Mean," CG&A, **12**(1), January 1992.

[FauL01] Faugeras, Olivier, and Luong, Quang-Tuan, *The Geometry of Multiple Images*, the MIT Press, 2001.

Graphics Standards

[ANSI85] ANSI (American National Standards Institute), *American National Standard for Information Processing Systems – Computer Graphics – Graphical Kernel System (GKS) Functional Description*, ANSI X3.124-1985, ANSI, New York, 1985.

[ANSI88] ANSI (American National Standards Institute), *American National Standard for Information Processing Systems – Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File*, ANSI X3.144-1988, ANSI, New York, 1988.

[BarD98] Bargen, Bradley, and Donnelly, Peter, *Inside DirectX*, Microsoft Press, 1998.

[BeBF78] Bergeron, R.D., Bono, P.R., and Foley, J.D., "Graphics Programming Using the Core System," ACM Computing Surveys, **10**(4), December 1978, 389–443.

[BDDH95] Brodlie, Ken W., Damnjanovic, Ljiljana B., Duce, David A., and Hopgood, F. Robert, "GKS-94: An Overview," CG&A, **15**(6), November 1995, 64–71.

[Cars98] Carson, George S., "The History of Computer Graphics Standards Development," Computer Graphics, **32**(1), 1998, 34–38.

[EnKP84] Enderle, G., Kansy, K., and Pfaff, G., *Computer Graphics Programming: GKS – The Graphical Kernel System*, Springer-Verlag, 1984.

[Glid97] Glidden, Rob, *Graphics Programming with Direct3D*, Addison-Wesley Developers Press, 1997.

[GSPC77] Graphics Standards Planning Committee, "Status Report of the Graphics Standards Planning Committee," Computer Graphics, **11**, 1977.

[GSPC79] Graphics Standards Planning Committee, "Status Report of the Graphics Standards Planning Committee," Computer Graphics, **13**(3), 1979.

[IGES88] Initial Graphics Exchange Specification (IGES), Version 4.0, U.S. Department of Commerce, National Bureau of Standards, Washington, DC, 1988.

[ISO 88] International Standards Organization, *International Standard Information Processing Systems Computer Graphics – Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description*, ISO Document Number 8805:1988(E), American National Standards Institute, New York, 1988.

[KemF97] Kempf, Renate, and Frazier, Chris, Editors, *OpenGL Reference Manual*, 2nd Edition, Addison-Wesley Developers Press, 1997.

[Timm96] Timmins, Bret, *DirectDraw Programming*, M&T Books, 1996.

[VanD88] van Dam, A., "PHIGS + Functional Description, Revision 3.0," Computer Graphics, **22**(3), July 1988, 125–218.

[WNDS99] Woo, Mason, Neider, Jackie, Davis, Tom, and Shreiner, Dave, *OpenGL Programming Guide*, 3rd Edition, Addison Wesley Longman, 1999.

[WriS00] Wright, Richard S., Jr., and Sweet, Michael, *OPENGL SuperBible*, 2nd Edition, Waite Group Press, 2000.

Hodographs

[Faro92] Farouki, Rida T., "Pythagorean-Hodograph Curves in Practical Use," in [Barn92], 3–33.

[KimD93] Kim, Deok-Soo, "Hodograph Approach to Geometric Characterization of Parametric Cubic Curves," CAD, **25**(10), October 1993, 644–654.

[Moon99] Moon, Hwan Pyo, "Minkowski Pythagorean Hodographs," CAGD, **16**(8), September 1999, 739–753.

[SaWS95] Saito, Takafumi, Wang, Guo-Jin, and Sederberg, Thomas W., "Hodographs and Normals of Rational Curves and Surfaces," CAGD, **12**(4), June 1995, 417–430.

[SedW87] Sederberg, T., and Wang, X., "Rational Hodographs," CAGD, **4**(4), 1987, 333–335.

Implicit Curves and Surfaces

[AllG87] Allgower, E.L., and Gnutzmann, S., "An Algorithm for Piecewise Linear Approximations of an Implicitly Defined Two-Dimensional Surfaces," SIAM J. Numerical Analysis, **24**(2), April 1987, 452–469.

[AllG91] Allgower, E.L., and Gnutzmann, S., "Simplicial Pivoting for Mesh Generation of Implicitly Defined Surfaces," CAGD, **8**(4), October 1991, 305–325.

[AllS85] Allgower, E.L., and Schmidt, P.H., "An Algorithm for Piecewise-Linear Approximations of an Implicitly Defined Manifold," SIAM J. Numerical Analysis, **22**(2), April 1985, 322–346.

[Bloo88] Bloomenthal, Jules, "Polygonization of Implicit Surfaces," CAGD, **5**(4), November 1988, 341–355.

[Bloo97] Bloomenthal, Jules, editor, *Introduction to Implicit Surfaces*, Morgan Kaufmann Publ., 1997.

[Chan88] Chandler, Richard E., "A Tracking Algorithm for Implicitly Defined Curves," CG&A, **8**(2), March 1988, 83–89.

[GarZ79] Garcia, C.B., and Zangwill, W.I., "Finding All Solutions to Polynomial Systems and Other Systems of Equations," Math. Programming, **16**(1979), 159–176.

[GonN02] Gonzalez-Vega, Laureano, and Necula, Ioana, "Efficient Topology Determination of Implicitly Defined Algebraic Plane Curves," CAGD, **19**(9), December 2002, 719–743.

[Hoff93] Hoffmann, Christoph M., "Implicit Curves and Surfaces in CAGD," CG&A, **13**(1), Jan., 1993, 79–88.

[Morg83] Morgan, A.P., "A Method for Computing All Solutions to Systems of Polynomial Equations," ACM Trans. on Math. Software, **9**(1983), 1–17.

[NinB93] Ning, Paul, and Bloomenthal, Jules, "An Evaluation of Implicit Surface Tilers," CG&A, **13**(4), November 1993, 33–41.

[SeZZ89] Sederberg, Thomas W., Zhao, Junwu, and Zundel, Alan K., "Approximate Parameterization of Algebraic Curves," in [StrS89], 33–54.

[VeVC94] Verschelde, Jan, Verlinden, Pierre, and Cools, Ronald, "Homotopies Exploiting Polytopes for Solving Sparse Polynomial Systems," SIAM J. Numer. Anal., **31**(3), June 1994, 915–930.

[Wats86] Watson, Layne T., "Numerical Linear Algebra Aspects of Globally Convergent Homotopy Methods," SIAM Review, **28**(4), December 1986, 529–545.

[Wrig85] Wright, A.H., "Finding All Solutions to a System of Polynomial Equations," Math. of Computation, **44**(1985), 125–133.

Intersection Algorithms

[AbdY96] Abdel-Malek, Karim, and Yeh, Harn-Jou, "Determining Intersection Curves Between Surfaces of Two Solids," CAD, **28**(6/7), June/July 1996, 539–549.

[AbdY97] Abdel-Malek, Karim, and Yeh, Harn-Jou, "On the Determination of Starting Points for Parametric Surface Intersections," CAD, **29**(1), January 1997, 21–35.

[Aste88] Asteasu, C., "Intersection of Arbitrary Surfaces," CAD, **20**(9), November 1988, 533–538.

[AzBB90] Aziz, Nadim. M., Bata, Reda., and Bhat, Sudarshan, "Bézier Surface/Surface Intersection," CG&A, **10**(1), 1990, 50–58.

[BHLH88] Bajaj, C.L, Hoffmann, C.M., Lynch, R.E., and Hopcroft, J.E.H., "Tracing Surface Intersections," CAGD, **5**(4), November 1988, 285–307.

[BajX92] Bajaj, Chanderjit, and Xu, Gualiang, "NURBS Approximation of Surface/Surface Intersection Curves," Technical Report CSD-TR-92-036, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, June 1992.

[BFJP87] Barnhill, R.E., Farin, G., Jordan, M., and Piper, B.R., "Surface/Surface Intersection," CAGD, **4**(1), January 1987, 3–16.

[BarK90] Barnhill, R.E., and Kersey, S.N., "A Marching Method for Parametric Surface/Surface Intersection," CAGD, **7**(1–4), June 1990, 257–280.

[Boen91] Boender, E., "A Survey of Intersection Algorithms for Curved Surfaces," Computers and Graphics, **15**, 1991, 109–115.

[BurS93] Bürger, Heiko, and Schaback, Robert, "A Parallel Multistage Method for Surface/Surface Intersection," CAGD, **10**, 1993, 277–291.

[Carl82] Carlson, Wayne E., "An Algorithm and Data Structure for 3D Object Synthesis Using Surface Patch Intersections," Computer Graphics, **16**(3), July 1982, 255–263.

[ChaK87] Chandru, Vidaya, and Kochar, Bipin S., "Analytic Techniques for Geometric Intersection Problems," in [Fari87], 305–318.

[ChBA94] Chang, Long Chyr, Bein, Wolfgang W., and Angel, Edward, "Surface Intersection Using Parallelism," CAGD, **11**(1), February 1994, 39–69.

[CheO88] Chen, J.J., and Ozsoy, T.M., "Predictor-Corrector Type of Intersection Algorithm for C^2 Parametric Surfaces," CAD, **20**(6), July/Aug, 1988, 347–352.

[Chen89] Cheng, Koun-Ping, "Using Plane Vector Fields to Obtain All the Intersection Curves of Two General-Surfaces," in [StrS89], 187–204.

[CycW92] Cychosz, Joseph M., and Waggenspack, Warren N. Jr., "Intersecting a Ray with a Quadric Surface," in [Kirk92], 275–283.

[CycW94] Cychosz, Joseph M., and Waggenspack, Warren N. Jr., "Intersecting a Ray with a Cylinder," in [Heck94], 356–365.

[DoSY89] Dokken, T., Skytt, V., Ytrehus, A.-M., "Recursive Subdivision and Iteration in Intersections and Related Problems," in [LycS89], 207–214.

[Fari92b] Farin, Gerald, "An SSI Bibliography," in [Barn92], 205–207.

[Faro87] Farouki, Rida T., "Direct Surface Section Evaluation," in [Fari87], 319–334.

[FaNO89] Farouki, R.T., Neff, C.A., and O'Connor, M.A., Automatic Parsing of Degenerate Quadric-Surface Intersection," TOG, **8**(3), July 1989, 174–203.

[GarW89] Garrity, Thomas, and Warren, Joe, "On Computing the Intersection of a Pair of Algebraic Surfaces," CAGD, **6**(2), May 1989, 137–153.

[GolM87] Goldman, Ronald N., and Miller, James R., "Combining Algebraic Rigor with Geometric Robustness for the Detection and Calculation of Conic Sections in the Intersection of Two Natural Quadric Surfaces," in [Fari87], 221–231.

[GolS87] Goldman, R.N., and Sederberg, T.W., "Analytic Approach to Intersection of All Piecewise Parametric Rational Cubic Curves," CAD, **19**(6), July/August 1987, 282–292.

[GraK97] Grandine, T.A., and Klein IV, F.W., "A New Approach to the Surface Intersection Problem," CAGD, **14**(2), February 1997, 111–134.

[Gree94] Greene, Ned, "Detecting Intersection of a Rectangular Solid and a Convex Polyhedron," in [Heck94], 74–82.

[HaAG83] Hanna, S.L., Abel, J.F., and Greenberg, D.P., "Intersection of Parametric Surfaces by Means of Look-Up Tables," CG&A, **3**(5), October 1983, 39–48.

[HeKE99] Heo, Hee-Seok, Kim, Myung-Soo, and Elber, Gershon, "The Intersection of Two Ruled Surfaces," CAD, **31**(1), January 1999, 33–50.

[Hohm91] Hohmeyer, Michael E., "A Surface Intersection Algorithm Based on Loop Detection," in [RosT91], 197–207.

[HEFS85] Houghton, Elizabeth G., Emnett, Robert F., Factor, James D., Sabharwal, Chaman L., "Implementation of a Divide-and-Conquer Method for Intersection of Parametric Surfaces," CAGD, **2**(1–3), 1985, 173–183.

[HMPY97] Hu, Chun-Yi, Maekawa, Takashi, Patrikalakis, Nicholas M., and Ye, Xiuzi, "Robust Interval Algorithm for Surface Intersections," CAD, **29**(9), September 1997, 617–627.

[HMSP96] Hu, Chun-Yi, Maekawa, Takashi, Sherbrooke, Evan C., and Patrikalakis, Nicholas M., "Robust Interval Algorithm for Curve Intersections," CAD, **28**(6/7), June/July 1996, 495–506.

[KlaK92] Klass, Reinhold, and Kuhn, Bernhard, "Fillet and Surface Intersections Defined by Rolling Balls," CAGD, **9**(3), August 1992, 185–193.

[Klas94] Klassen, R. Victor, "Intersecting Parametric Cubic Curves by Midpoint Subdivision," in [Heck94], 261–277.

[Kopa91] Koparkar, P., "Surface Intersection by Switching from Recursive Subdivision to Iterative Refinement," The Visual Computer, **8**(1991), 47–63.

[KrPW92] Kriezis, G.A., Patrikalakis, N.M., and Wolter, F-E., "Topological and Differential-Equation Methods for Surface Intersections," CAD, **24**(1), January 1992, 41–55.

[KrPP90] Kriezis, G.A., Prakash, P.V., and Patrikalakis, N.M., "Method for Intersecting Algebraic Surfaces with Rational Polynomial Patches," CAD, **22**(10), December 1990, 645–654.

[KriM97] Krishnan, Shankar, and Manocha, Dinesh, "An Efficient Surface Intersection Algorithm Based on Lower-Dimensional Formulation," ACM TOG, **16**(1), January 1997, 74–106.

[LamM95] Lamure, Hervé, and Michelucci, Dominique, "Solving Geometric Constraints by Homotopy," in [HofR95], 263–269.

[LamM96] Lamure, Hervé, and Michelucci, Dominique, "Solving Geometric Constraints by Homotopy," IEEE Trans. on Visualization and Comp. Graphics, **2**(1), March 1996, 28–34.

[Luka89] Lukács, Gábor, "The Generalized Inverse Matrix and the Surface-Surface Intersection Problem," in [StrS89], 167–185.

[LuMM95] Luo, Ren C., Ma, Yawei, and McAllister, David F., "Tracing Tangential Surface-Surface Intersections," in [HofR95], 255–262.

[MaLe98] Ma, Yawei, and Lee, Yuan-Shin, "Detection of Loops and Singularities of Surface Intersections," CAD, **30**(14), December 1998, 1059–1067.

[ManD94] Manocha, Dinesh, and Demmel, J., "Algorithms for Intersecting Parametric and Algebraic Curves I: Simple Intersections," TOG, **13**(1), January 1994, 73–100.

[ManK97] Manocha, Dinesh, and Krishnan, Shankar, "Algebraic Pruning: A Fast Technique for Curve and Surface Intersection," CAGD, **14**(9), December 1997, 823–845.

[MarM89] Markot, R.P., and Magedson, R.L., "Solutions of Tangential Surface and Curve Intersections," CAD, **21**(7), September 1989, 421–429.

[MarM91] Markot, R.P., and Magedson, R.L., "Procedural Method for Evaluating the Intersection Curves of Two Parametric Surfaces," CAD, **23**(6), July/Aug, 1991, 395–404.

[Mill87] Miller, J.R., "Geometric Approaches to Nonplanar Quadric Surface Intersection Curves," TOG, **6**(4), 1987, 274–307.

[OweR87] Owen, J.C., and Rockwood, A.P., "Intersection of General Implicit Surfaces," in [Fari87], 335–345.

[Patr92] Patrikalakis, Nicholas M., "Interrogation of Surface Intersections," in [Barn92], 161–185.

[Patr93] Patrikalakis, Nicholas M., "Surface-to-Surface Intersections," CG&A, **13**(1), Jan., 1993, 89–95.

[PhiO84] Phillips, M.B., and Odell, G.M., "An Algorithm for Locating and Displaying the Intersection of Two Arbitrary Surfaces," CG&A, **4**(9), September 1984, 48–58.

[Pieg92] Piegl, Les A., "Constructive Geometric Approach to Surface-Surface Intersection," in [Barn92], 137–159.

[Powe72] Powell, M.J.D., "Problems Related to Unconstrained Optimisation," in Murray, W., editor, *Numerical Methods for Unconstrained Optimisation*, Academic Press, 1972.

[PraG86] Pratt, M.J., and Geisow, A.D., "Surface/Surface Intersection Problems," in [Greg86], 117–142.

[Rock90] Rockwood, A.P., "Accurate Display of Tensor Product Isosurfaces," in Kaufman, A., editor, *Visualization '90*, IEEE Computer Society Press, 1990, 353–360.

[Sede89] Sederberg, T.W., "Algorithm for Algebraic Curve Intersection," CAD, **21**(9), November 1989, 547–554.

[SeCK89] Sederberg, T.W., Christiansen, H.N., and Katz, S., "Improved Test for Closed Loops in Surface Intersections," CAD, **21**(8), October 1989, 505–508.

[SedM88] Sederberg, Thomas W., and Meyers, Ray J., "Loop Detection in Surface Patch Intersections," CAGD, **5**(2), July 1988, 161–171.

[SedN90] Sederberg, Thomas W., and Nishita, T., "Curve Intersection Using Bézier Clipping," CAD, **22**(9), November 1990, 538–549.
[SedP86] Sederberg, Thomas W., and Parry, S.R., "A Comparison of Three Curve Intersection Algorithms," CAD, **18**(1), January/February 1986, 58–63.
[SheJ87] Shene, Ching-Kuang, and Johnstone, John K., "On the Planar Intersection of Natural Quadrics," in [Fari87], 233–242.
[Shen94] Shene, Ching-Kuang, "Computing the Intersection of a Line and a Cylinder," in [Heck94], 353–355.
[Shen95] Shene, Ching-Kuang, "Computing the Intersection of a Line and a Cone," in [Paet95], 227–231.
[Stoy92] Stoyanov, Tz.E., "Marching Along Surface/Surface Intersection Curves with an Adaptive Step Length," CAGD, **9**(6), December 1992, 485–489.
[Timm77] Timmer, H.G., "Analytical Background for Computation of Suface Intersections," Douglas Aircraft Company Technical Memorandum, C1-250-CAT-77-036, April 1977.
[Turn88] Turner, Joshua U., "Accurate Solid Modeling Using Polyhedral Approximations," CG&A, **8**(3), May 1988, 14–28.
[Wang92] Wang, K.Y., "Parametric Surface Intersections," in [Barn92], 187–204.
[WilM93] Wilf, Itzhak, and Manor, Yehuda, "Quadric-Surface Intersection Curves: Shape and Structure," CAD, **25**(10), October 1993, 633–643.
[WuAn99] Wu, Shin-Ting, and Andrade, Lenimar N., "Marching Along a Regular Surface/Surface Intersection with Circular Steps," CAGD, **16**(4), May 1999, 249–268.
[YeMa99] Ye, Xiuzi, and Maekawa, Takashi, "Differential Geometry of Intersection Curves of Two Surfaces," CAGD, **16**(8), September 1999, 767–788.
[ZhoS99] Zhou, Y., and Suri, S., "Analysis of a Bounding Box Heuristic for Object Intersection," J. of the ACM, **46**(6), November 1999, 833–857.

Interval Analysis

(See also [Snyd92])

[AleH83] Alefeld, Götz, and Herzberger, Jürgen, *Introduction to Interval Computations*, Academic Press, 1983.
[Garl85] Garloff, J., "Interval Mathematics. A Bibliography," Freiburger Interval-Berichte, 1985, Volume **6**, 1–122.
[Garl87] Garloff, J., "Bibliography on Interval Mathematics. Continuation," Freiburger Interval-Berichte, 1987, Volume **2**, 1–50.
[HuPY96a] Hu, Chun-Yi, Patrikalakis, Nicholas M., and Ye, Xiuzi, "Robust Interval Solid Modelling, Part I: Representations," CAD, **28**(10), October 1996, 807–817.
[HuPY96b] Hu, Chun-Yi, Patrikalakis, Nicholas M., and Ye, Xiuzi, "Robust Interval Solid Modelling, Part II: Boundary Evaluation," CAD, **28**(10), October 1996, 819–830.
[Moor66] Moore, Ramon E., *Interval Analysis*, Prentice-Hall, Inc., 1966.
[Moor79] Moore, Ramon E., *Methods and Applications of Interval Analysis*, SIAM, 1979.
[Snyd92a] Snyder, John M., "Interval Analysis for Computer Graphics," SIGGRAPH 92, **26**(2), July 1992, 121–130.

Mathematics for Geometric Modeling

[AgoM05] Agoston, Max K., *Computer Graphics and Geometric Modeling: Mathematics*, Springer, 2005.
[BowW83] Bowyer, A., and Woodwark, J., *A Programmer's Geometry*, Butterworths, 1983.
[Hogg92] Hoggar, S.G., *Mathematics for Computer Graphics*, Cambridge Univ. Press, 1992.
[Mort89] Mortenson, M.E., *Computer Graphics: An Introduction to the Mathematics and Geometry*, Industrial Press, 1989.
[RogA90] Rogers, D.F., and Adams, J.A., *Mathematical Elements for Computer Graphics*, 2nd Edition, McGraw-Hill, 1990.

Medial Axes

[BBGS99] Blanding, Robert, Brooking, Cole, Ganter, Mark, and Storti, Duane, "A Skeletal-Based Solid Editor," in [BroA99], 141–150.

[Blum67] Blum, H., "A Transformation for Extracting New Descriptors of Shape," in Wathen-Dunn, Weinant, editor, *Models for the Perception of Speech and Visual Form*, MIT Press, 1967, 362–381.

[Blum73] Blum, H., "Biological Shape and Visual Science, Part I," J. Theoretical Biology, **38**, 1973, 205–287.

[BluN78] Blum, H., and Nagel, R.N., "Shape Description Using Weighted Symmetric Axis Features," Pattern Recognition, **10**, 1978, 167–180.

[Bran92] Brandt, J.W., "Describing a Solid with the Three-Dimensional Skeleton," in Warren, J.D., editor, *Proceedings of the International Society for Optical Engineering Volume 1830, Curves and Surfaces in Computer Vision and Graphics III*, SPIE, Boston, Massachusetts, 1992, 258–269.

[CalH68] Calabi, L., and Hartnett, W.E., "Shape Recognition, Prairie Fires, Convex Deficiencies and Skeletons," Amer. Math. Monthly, **75**, 1968, 335–342.

[Chia92] Chiang, Ching-Shoei, "The Euclidean Distance Transform," PhD thesis, Technical Report CSD-TR 92-050, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, August 1992.

[ChCM97] Choi, Hyeong In, Choi, Sung Woo, and Moon, Hwan Pyo, "Mathematical Theory of Medial Axis Transform," Pacific J. of Math., **181**(1), November 1997, 57–88.

[CuKM99] Culver, Tim, Keyser, John, and Manocho, Dinesh, "Accurate Computation of the Medial Axis of a Polyhedron," in [BroA99], 179–190.

[ElbK99] Elber, Gershon, and Kim, Myung-Soo, "Rational Bisectors of CSG Primitives," in [BroA99], 159–166.

[FarJ94] Farouki, R.T., and Johnstone, J.K., "Computing Point/Curve and Curve/Curve Bisectors," in [Fish94], 327–354.

[FarR98] Farouki, Rida T., and Ramamurthy, Rajesh, "Degenerate Point/Curve and Curve/Curve Bisectors Arising in Medial Axis Computations for Planar Domains with Curved Boundaries," CAGD, **15**(6), June 1998, 615–635.

[GelD95] Gelston, Sean M., and Dutta, Debasish, "Boundary Surface Recovery from Skeleton Curves and Surfaces," CAGD, **12**(1), February 1995, 27–51.

[GibB85] Giblin, P.J., and Brassett, S.A., "Local Symmetry of Plane Curves," Amer. Math. Monthly, **92**(10), December 1985, 689–707.

[Hoff91] Hoffmann, Christoph M., "Computer Vision, Descriptive Geometry, and Classical Mechanics," Technical Report CSD-TR-91-073, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, October 1991, also in Proceedings of the Eurographics Workshop, *Computer Graphics and Mathematics*, edited by Falcidieno, B., Hermann, I., and Pienovi, C., Genoa, Italy, October 1991, 229–244.

[Hoff94] Hoffmann, Christoph M., "How to Construct the Skeleton of CSG Objects," in [Bowy94], 421–437.

[LazV99] Lazarus, Francis, and Verroust, Anne, "Level Set Diagrams of Polyhedral Objects," in [BroA99], 130–140.

[LaCJ94] Lazarus, Francis, Coquillart, Sabine, and Jancène, "Axial Deformations: An Intuitive Deformation Technique," CAD, **26**(8), August 1994, 607–613.

[Nack82] Nackman, Lee R., "Curvature Relations in Three-Dimensional Symmetric Axes," CGIP, **20**(1), September 1982, 43–57.

[RamG03] Ramanathan, M., and Gurumoorthy, B., "Constructing Medial Axis Transform of Planar Domains with Curved Boundaries," CAD, **35**(7), June 2003, 619–632.

[RedT95] Reddy, Jayachandra, and Turkiyyah, George M., "Computation of 3D Skeletons Using a Generalized Delaunay Triangulation Technique," CAD, **27**(9), September 1995, 677–694.

[ShAR95] Sheehy, D.J., Armstrong, C.G., and Robinson, D.J., "Computing the Medial Surface of a Solid from a Domain Delaunay Triangulation," in [HofR95], 201–212.

[ShAR96] Sheehy, Damian J., Armstrong, Cecil G., and Robinson, Desmond J., "Shape Description by Medial Surface Construction," IEEE Trans. on Visualization and Comp. Graphics, **2**(1), March 1996, 62–72.

[ShPB95] Sherbrooke, Evan C., Patrikalakis, Nicholas M., and Brisson, Erik, "Computation of the Medial Axis Transform of 3-D Polydedra," in [HofR95], 187–200.

[ShPB96] Sherbrooke, Evan C., Patrikalakis, Nicholas M., and Brisson, Erik, "An Algorithm for the Medial Axis Transform of 3-D Polyhedral Solids," IEEE Trans. on Visualization and Comp. Graphics, **2**(1), March 1996, 44–61.

[STGLS97] Storti, D., Turkiyyah, G., Ganter, M., Lim, C.T., and Stal, D., "Skeleton-based Modeling Operations on Solids," in [HofB97], 141–154.
[TaJS99] Tate, S.J., Jared, G.E.M., Swift, K.G., "Detection of Symmetry and Primary Axes in Support of Proactive Design for Assembly," in [BroA99], 151–158.
[TSGCV97] Turkiyyah, George M., Storti, Duane W., Ganter, Mark, Chen, Hao, and Vimawala, Munikumar, "An Accelerated Triangulation Method for Computing the Skeletons of Free-form Solid Models," CAD, **29**(1), January 1997, 5–19.
[Verm94] Vermeer, P.J., "Medial Axis Transform to Boundary Representation Conversion," PhD thesis, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, 1994.
[Wolt95] Wolter, F.E., "Cut Locus and Medial Axis in Global Shape Interrogation and Representation," CAGD, 1995.
[YuGD91] Yu, Xinhua, Goldak, John, and Dong, Lingxian, "Constructing 3-D Discrete Medial Axis," in [RosT91], 481–492.

Miscellaneous

[AbeD81] Abelson, Harold, and diSessa, Andrea A., *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*, the MIT Press, 1981.
[Arvo91] Arvo, James, editor, *Graphics Gems II*, Academic Press, 1991.
[Bado90] Badouel, Didier, "An Efficient Ray-Polygon Intersection," in [Glas90], 390–393.
[Chas78] Chasen, Sylvan H., *Geometric Principles and Procedures for Computer Graphic Applications*, Prentice-Hall, Inc., 1978.
[Chin95] Chin, Normal, "A Walk Through BSP Trees," in [Paet95], 121–138.
[Ferw01] Ferwerda, James A., "Elements of Early Vision for Computer Graphics," CG&A, **21**(5), September/October 2001, 22–33.
[Fium89] Fiume, E.L., *The Mathematical Structure of Raster Graphics*, Academic Press, 1989.
[Geor92] Georgiades, Príamos, "Signed Distance from Point to Plane," in [Kirk92], 223–224.
[Glas90] Glassner, A.S., editor, *Graphics Gems*, Academic Press, 1990.
[Gold90] Goldman, Ronald, "Matrices and Transformations," in [Glas90], 472–475.
[Hain94] Haines, Eric, "Point in Polygon Strategies," in [Heck94], 24–46.
[Heck94] Heckbert, Paul S., editor, *Graphics Gems IV*, Academic Press, 1994.
[Herm98] Herman, Gabor T., *Geometry of Digital Spaces*, Birkhäuser,1998.
[Hodg92] Hodges, Larry F., "Tutorial: Time-Multiplexed Stereoscopic Computer Graphics," CG&A, **12**(2), March 1992, 20–30.
[Kirk92] Kirk, David, editor, *Graphics Gems III*, Academic Press, 1992.
[Mill99] Miller, James R., "Applications of Vector Geometry for Robustness and Speed," CG&A, **19**(4), July/August 1999, 68–73.
[Morr91] Morrison, Jack C., "Distance from a Point to a Line," in [Arvo91], 10–13.
[Paet95] Paeth, Alan W., editor, *Graphics Gems V*, Academic Press, 1995.
[Pavl82] Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, 1982.
[Pras91] Prasad, Mukesh, "Intersection of Line Segments," in [Arvo91], 7–9.
[Rose79] Rosenfeld, A., *Picture Languages*, Academic Press, 1979.
[SchS95] Schlick, Christophe, and Subrenat, Gilles, "Ray Intersection of Tessellated Surfaces: Quadrangles Versus Triangles," in [Paet95], 232–241.
[VanG95] Van Gelder, Allen, "Efficient Computation of Polygon Area and Polyhedron Volume," in [Paet95], 35–41.
[WatP98] Watt, Alan, and Policarpo, Fabio, *The Computer Image*, Addison-Wesley, 1998.
[Weil94] Weiler, Kevin, "An Incremental Angle Point in Polygon Test," in [Heck94], 16–23.

Numerical Methods

[ConD72] Conte, S.D., and de Boor, C., *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw-Hill Book Co., 1972.
[DahB74] Dahlquist, G., and Bjorck, A., *Numerical Methods*, Prentice-Hall, Inc., 1974.
[Horn75] Hornbeck, Robert W., *Numerical Methods*, Quantum Publishers, Inc., 1975.
[McCa98] McCartin, Brian J., "Seven Deadly Sins of Numerical Computation," Amer. Math. Monthly, **105**(10), December 1998, 929–941.

[PFTV86] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes: The Art of Scientific Computing*, Cambridge Univ. Press, 1986.

[Wall90] Wallis, Bob, "Tutorial on Forward Differencing," in [Glas90], 594–603.

Offset Curves and Surfaces

[BarF95] Barnhill, R.E., and Frost, T.M., "Parametric Offset Surface Approximation," in [HaFN95], 1–34.

[Brec92] Brechner, Eric L., "General Offset Curves and Surfaces," in [Barn92], 101–121.

[ElLK97] Elber, Gershon, Lee, In-Kwon, and Kim, Myung-Soo, "Comparing Offset Curve Approximation Methods," CG&A, **17**(3), May–June 1997, 62–71.

[Faro85] Farouki, R.T., "Exact Offset Procedures for Simple Solids," CAGD, **2**(4), December 1985, 257–279.

[Faro86] Farouki, R.T., "The Approximation of Non-degenerate Offset Surfaces," CAGD, **3**(1), May 1986, 15–43.

[FarN90a] Farouki, R.T., and Neff, C.A., "Analytic Properties of Plane Offset Curves," CAGD, **7**(1990), 83–99.

[FarN90b] Farouki, R.T., and Neff, C.A., "Algebraic Properties of Plane Offset Curves," CAGD, **7**(1990), 101–127.

[FarS95] Farouki, Rida T., and Sederberg, Thomas W., "Analysis of the Offset to a Parabola," CAGD, **12**(6), September 1995, 639–645.

[Fors95] Forsyth, Mark, "Shelling and Offsetting Bodies," in [HofR95], 373–381.

[HMSV99] Hartquist, E.E., Menon, J.P., Suresh, K., Voelcker, H.B., and Zagajac, J., "A Computing Strategy for Applications Involving Offsets, Sweeps, and Minkowski Operations," CAD, **31**(3), March 1999, 175–183.

[KuSP02] Kumar, G.V.V. Ravi, Shastry, K.G., and Prakash, B.G., "Computing Non-self-intersecting Offsets of NURBS surfaces," CAD, **34**(3), March 2002, 209–228.

[KuSP03] Kumar, G.V.V. Ravi, Shastry, K.G., and Prakash, B.G., "Computing Constant Offsets of a NURBS B-Rep," CAD, **35**(10), 935–944.

[Lü95] Lü, Wei, "Offset-rational Parametric Plane Curves," CAGD, **12**(6), September 1995, 601–616.

[Maek99] Maekawa, Takashi, "An Overview of Offset Curves and Surfaces," CAD, **31**(3), March 1999, 165–173.

[MaeP93] Maekawa, Takashi, and Patrikalakis, Nicholas M., "Computation of Singularities and Intersections of Offsets of Planar Curves," CAGD, **10**(5), October 1993, 407–429.

[MeeW90] Meek, D.S., and Walton, D.J., "Offset Curves of Clothoidal Splines," CAD, **22**(4), May 1990, 199–201.

[Pham92] Pham, B., "Offset Curves and Surfaces: A Brief Survey," CAD, **24**(4), April 1992, 223–229.

[Pott95] Pottmann, H., "Rational Curves and Surfaces with Rational Offsets," CAGD, **12**(2), March 1995, 175–192.

[RosR86] Rossignac, Jaroslaw R., and Requicha, Aristides A.G., "Offsetting Operations in Solid Modeling," CAGD, **3**(1986), 129–148.

[SaPD88] Saeed, S.E.O., de Pennington, A., and Dodsworth, J.R., "Offsetting in Geometric Modelling," CAD, **20**(2), March 1988, 67–74.

[TilH84] Tiller, W., and Hanson, E., "Offsets of Two Dimensional Profiles," CG&A, **4**(9), September 1984, 36–46.

PC Oriented Computer Graphics

[Ferr94] Ferraro, R.F., *Programmer's Guide to the EGA, VGA, and Super VGA Cards*, 3rd Edition, Addison-Wesley Publ. Co., 1994.

[Wilt87] Wilton, R., *Programmer's Guide to PC & PS/2 Video Systems*, Microsoft Press, 1987

Physically Based Modeling

[BaraW98] Baraff, David, and Witkin, Andrew, Organizers, *Physically Based Modeling*, Course Notes, Volume 13, SIGGRAPH 98, July 1998.

[BarrA87] Barr, Alan H., Organizer, *Topics in Physically-Based Modeling*, Course Notes, Volume 16, SIGGRAPH 87, July 1987.

[BarrA89] Barr, Alan H., editor, *Topics in Physically Based Modeling*, Addison-Wesley, 1989.
[Barze92] Barzel, Ronen, *Physically-Based Modeling for Computer Graphics*, Academic Press, 1992.
[KassB93] Kass, Michael, and Baraff, David, Organizers, *An Introduction to Physically Based Modeling*, Course Notes, Volume 60, SIGGRAPH 93, August 1993.

Polygonization Algorithms

(See also Implicit Curves and Surfaces)

[Cuil98] Cuillière, J.C., "An Adaptive Method for the Automatic Triangulation of 3D Parametric Surfaces," CAD, **30**(2), February 1998, 139–150.
[DeSB92] Dey, Tamal K., Sugihara, Kokichi, and Bajaj, Chanderjit, "Triangulations in Three Dimensions with Finite Precision Arithmetic," Technical Report CSD-TR-92-001, Comp. Science Dept., Purdue Univ., West Lafayette, Indiana, 47907-1398, USA, January 1992.
[Fili86] Filip, Daniel J., "Adaptive Subdivision Algorithms for a Set of Bézier Triangles," CAD, **18**(2), March 1986, 74–78.
[HerB87] Von Herzen, B., and Barr, A.H., "Accurate Triangulations of Deformed, Intersecting Surfaces," SIGGRAPH 87, **21**(4), July 1987, 103–110.
[Hiro74] Hironaka, H., "Triangulations of Algebraic Sets," in *Algebraic Geometry, ARCATA 1974*, Proc. of Symposia in Pure Mathematics, AMS, Providence, R.I., 1975.
[LiSH92] Lindgren, Terence, Sanchez, Juan, and Hall, Jim, "Curve Tessellation Criteria Through Sampling," in [Kirk92], 262–265.
[Schu93] Schumaked, Larry L., "Triangulations in CAGD," CG&A, **13**(1), January 1993, 47–52.
[ShiG95] Shimada, Kenji, and Gossard, David C., "Bubble Mesh: Automated Triangular Meshing of Non-Manifold Geometry by Sphere Packing," in [HofR95], 409–419.
[StaH97] Stander, Barton T., and Hart, John C., "Guaranteeing the Topology of an Implicit Surface Polygonization for Interactive Modeling," SIGGRAPH 97, August 1997, 279–286.
[VeDG99] Velho, L., De Figueiredo, L.H., and Gomes, J., "A Unified Approach for Hierarchical Adaptive Tessellation of Surfaces," TOG, **18**(4), October 1999, 329–360.
[ZheS00] Zheng, Jianmin, and Sederberg, Thomas W., "Estimating Tessellation Parameter Intervals for Rational Curves and Surfaces," TOG, **19**(1), January 2000, 56–77.

Projective Geometry and Transformations

[Egga98] Eggar, M.H., "Pinhole Cameras, Perspective, and Projective Geometry," Amer. Math. Monthly, **105**(7), August–September 1998, 618–630.
[PenP86] Penna, M.A., and Patterson, R.R., *Projective Geometry and its Applications to Computer Graphics*, Prentice-Hall, 1986.

Quadrics

[Barr81] Barr, A.H., "Superquadrics and Angle-Preserving Transformations," CG&A, **1**(1), January 1981.
[Barr92] Barr, A.H., "Rigid Physically Based Superquadrics," in [Kirk92], 137–159.
[Gold83] Goldman, Ronald N., "Quadrics of Revolution," CG&A, **3**(3), March/April 1983, 68–76.

Quaternions

[Baez02] Baez, John C., "The Octonions," Bull. of the AMS, **39**(2), April 2002, 145–205.
[BCGH92] Barr, Alan H., Currin, Bena, Gabriel, Steven, and Hughes, John F., "Smooth Interpolation of Orientations with Angular Velocity Constraints Using Quaternions," SIGGRAPH 92, **26**(2), July 1992, 313–320.
[Brad82] Brady, Michael, "Trajectory Planning," in *Robot Motion: Planning and Control*, edited by Michael Brady, John M. Hollerbach, Timothy L. Johnson, Tomas Lozano-Perez, and Matthew T. Mason, The MIT Press, 1982.
[Brou84] Brou, Philippe, "Using the Gaussian Image to Find the Orientation of Objects," The International Journal of Robotics Research, **3**(4), Winter, 1984, 89–125.
[CouH53] Courant, R., and Hilbert, D., *Methods of Mathematical Physics, Volume I*, Interscience Publishers, Inc., New York, 1953.

[Hami69] Hamilton, William R., *Elements of Quaternions*, 3rd Edition, Chelsea Publ. Co., New York, 1969. Also in *Lectures on Quaternions* (1853): Republished in *The Mathematical Papers of Sir William Rowan Hamilton*, Volume III, *Algebra*, Cambridge Univ. Press, London, 1967.

[HaMa95] Hanson, Andrew J., and Ma, Hui, "Quaternion Frame Approach to Streamline Visualization," IEEE Trans. on Visualization and Computer Graphics, **1**(2), June 1995, 164–174.

[HarA02] Harada, Koichi, and Anzai, Takeshi, "Multiple Sweeping Using Quaternion Operations," CAD, **34**(11), September 2002, 815–822.

[Hers75] Herstein, I.N., *Topics in Algebra*, 2nd Edition, John Wiley and Sons, Inc., New York, 1975.

[KaLL83] Kane, Thomas R., Likins, Peter W., and Levinson, David A., *Spacecraft Dynamics*, McGraw-Hill, Inc., 1983.

[Kuip99] Kuipers, Jack B., *Quaternions and Rotation Sequences*, Princeton Univ. Press, 1999.

[MacB79] MacLane, Saunders, and Birkhoff, Garrett, *Algebra*, 2nd Edition, MacMillan Publ. Co., Inc., New York, 1979.

[MiTW73] Misner, Charles W., Thorne, Kip S., and Wheeler, John Archibald, *Gravitation*, W. H. Freeman and Co., San Francisco, Chapter 41 – Spinors, 1973.

[MitR68] Mitchell, E.E.L., and Rogers, A.E., "Quaternion Parameters in the Simulation of a Spinning Rigid Body," in *Simulation the Dynamic Modeling of Ideas and Systems with Computers*, John McLeod, P.E., editor, 1968.

[PicS83] Pickert, G. and Steiner, H.-G., "Chapter 8 – Complex Numbers and Quaternions," in *Fundamentals of Mathematics, Volume I – Foundations of Mathematics: The Real Number System and Algebra*, H. Behnke, F. Bachmann, K. Fladt, and W. Suss, editors, Translated by S.H. Gould, 1983.

[Port81] Porteous, Ian R., *Topological Geometry*, 2nd Edition, Cambridge Univ. Press, Cambridge, 1981.

[Shoe85] Shoemake, Ken, "Animating Rotation with Quaternion Curves," Computer Graphics, SIGGRAPH 85, **19**(3), July 1985, 245–254.

[Shoe91] Shoemake, Ken, "Quaternions and 4 × 4 Matrices," in [Arvo91], 351–354.

[Shoe93] Shoemake, Ken, "Quaternions," Course Notes, Volume 60, SIGGRAPH 93, August 1993.

[Stil98] Stillwell, John, "Exceptional Objects," Amer. Math. Monthly, **105**(9), November 1998, 850–858.

[Tayl79] Taylor, Russell H., "Planning and Execution of Straight Line Manipulator Trajectories," IBM J. Res. Develop. 23, No. 4, July 1979, 424–436. Also in *Robot Motion: Planning and Control*, edited by Michael Brady, John M. Hollerbach, Timothy L. Johnson, Tomas Lozano-Perez, and Matthew T. Mason, The MIT Press, 1982.

[YanF64] Yang, A.T., and Freudenstein, F., "Application of Dual-Number Quaternion Algebra to the Analysis of Spatial Mechanisms," J. Appl. Mech., Trans. ASME 86, 1964, 300–308.

Radiosity

[CCWG88] Cohen, M.F., Chen, S.E., Wallace, J.R., and Greenberg, D.P., "A Progressive Refinement Approach to Fast Radiosity Image Generation," SIGGRAPH 88, **22**(4), August 1988, 75–84.

[CohG85] Cohen, M.F., and Greenberg, D.P., "A Radiosity Solution for Complex Environments," SIGGRAPH 85, **19**(3), July 1985, 31–40.

[CohW93] Cohen, Michael F., and Wallace, John R., *Radiosity and Realistic Image Synthesis*, Academic Press, 1993.

[NeuN95] Neumann, Laszlo, and Neumann, Attila, "Radiosity and Hybrid Methods," ACM TOG, **14**(3), July 1995, 233–265.

[WaCG87] Wallace, J.R., Cohen, M.F., and Greenberg, D.P., "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," SIGGRAPH 87, **21**(4), July 1987, 311–320.

Raster Algorithms

[AckW81] Ackland, Bryan, and Weste, Neil, "The Edge Flag Algorithm—A Fill Method for Raster Scan Displays," IEEE Trans. on Computers, Vol **C-30**, January 1981.

[ArcM75] Arcelli, C., and Massarotti, A., "Regular Arcs in Digital Contours," CGIP, **4**(1975), 339–360.

[BoLZ75] Bongiovanni, G., Luccio, F., and Zorat, A., "The Discrete Equation of the Straight Line," IEEE Trans. Comp., **24**(1975), 310–313.

[BoyB00] Boyer, Vincent, and Bourdin, Jean-Jacques, "Auto-Adaptive Step Straight-Line Algorithm," CG&A, **20**(5), September/October 2000, 67–69.
[Bres65] Bresenham, J.E., "Algorithm for Computer Control of a Digital Plotter," IBM Systems Journal, **4**(1), 1965, 25–30.
[Bres77] Bresenham, J.E., "A Linear Algorithm for Incremental Digital Display of Circular Arcs," CACM, **20**(2), February 1977, 100–106.
[DeFL87] DeFanti, Tom, Frankel, Rick, and Leske, Larry, "A Call for the Publishing of Blt-Stones," CG&A, **7**(10), October 1987, 39–49.
[Fish90b] Fishkin, Ken, "Filling a Region in a Frame Buffer," in [Glas90], 278–284.
[FisB85] Fishkin, K.P., and Barsky, B.A., "An Analysis and Algorithm for Filling Propagation," Proc. Graphics Interface 1985, 203–212.
[Free69] Freeman, H., "A Review of Relevant Problems in the Processing of Line-drawing Data," in *Automatic Interpretation and Classification of Images*, edited by A. Grasselli, Academic Press, 1969.
[GupS81] Gupta, S., and Sproull, R.E., "Filtering Edges for Gray-Scale Displays," SIGGRAPH 81, **15**(3), August 1981, 1–5.
[Heck90a] Heckbert, Paul S., "What Are the Coordinates of a Pixel?" in [Glas90], 246–248.
[Heck90b] Heckbert, Paul S., "A Seed Fill Algorithm," in [Glas90], 275–277.
[Heck90c] Heckbert, Paul S., "Digital Line Drawing," in [Glas90], 99–100.
[Mcil92] McIlroy, M. Douglas, "Getting Raster Ellipses Right," ACM TOG, **11**(3), July 1992, 259–275.
[Morr90] Morrison, Jack C., "Fast Anti-aliasing Polygon Scan Conversion," in [Glas90], 76–83.
[Pitt67] Pitteway, M.L.V., "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter," Computer J., **10**(3), November 1967, 282–289.
[PitW80] Pitteway, M.L.V., and Watkinson, D.J., "Bresenham's Algorithm with Grey-Scale," CACM, **23**(11), November 1980, 625–626.
[RoWW90] Rokne, J.G., Wyvill, Brian, and Wu, Xiaolin, "Fast Line Scan-Conversion," ACM TOG, **9**(4), October 1990, 376–388.
[Smit79] Smith, A.R., "Tint Fill," SIGGRAPH 79, **13**(2), August 1979, 276–283.
[SteL00] Stephenson, Peter, and Litow, Bruce, "Why Step When You Can Run," CG&A, **20**(6), November/December 2000, 76–84.
[Thom90] Thompson, Kelvin, "Rendering Anti-Aliased Lines," in [Glas90], 105–106.
[VanN85] Van Aken, Jerry, and Novak, Mark, "Curve-Drawing Algorithms for Raster Displays," ACM TOG, **4**(2), April 1985, 147–169.
[Wüth98] Wüthrich, Charles A., "A Model for Curve Rasterization in n-Dimensional Space," Computers & Graphics, **22**(2–3), 1998, 153–160.
[Wyvi90] Wyvill, Brian, "Symmetric Double Step Line Algorithm," in [Glas90], 101–104.

Ray Tracing

[Aman84] Amantides, J., "Ray Tracing with Cones," SIGGRAPH 84, **18**(3), July 1984, 129–136.
[Cook86] Cook, Robert L., "Stochastic Sampling in Computer Graphics," ACM TOG, **5**(1), January 1986, 51–72.
[Cook89] Cook, Robert L., "Stochastic Sampling and Distributed Ray Tracing," in [Glas89], 161–199.
[CoPL84] Cook, Robert L., Porter, Thomas, and Carpenter, Loren, "Distributed Ray Tracing," SIGGRAPH 84, **18**(3), July 1984, 137–145.
[FuTI86] Fujimoto, A., Tanaka, T., and Iwata, K., "ARTS: Accelerated Ray Tracing System," CG&A, **6**(4), April 1986, 16–26.
[Glas84] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing," CG&A, **4**(10), October 1984, 15–22.
[Glas86] Glassner, Andrew S., "An Overview of Ray Tracing," in [Glas89], 1–31.
[Glas89] Glassner, A.S., editor, *An Introduction to Ray Tracing*, Academic Press, 1989.
[Hain89] Haines, Eric., "Essential Ray Tracing," in [Glas89], 33–77.
[Hanr89] Hanrahan, Pat., "A Survey of Ray-Surface Intersection Algorithms," in [Glas89], 79–119.
[HKBZ97] Havran, Vlastimil, Kopal, Tomáš, Bittner, Jiři, and Žára, "Fast Robust BSP Tree Traversal Algorithm for Ray Tracing," J. of Graphics Tools, **2**(4), 1997, 15–23.
[HecH84] Heckbert, Paul S., and Hanrahan, Pat, "Beam Tracing Polygon Objects," SIGGRAPH 84, **18**(3), July 1984, 119–128.
[Kapl85] Kaplan, Michael R., "Space-Tracing, a Constant Time Ray-Tracer," Course Notes, Volume 11, SIGGRAPH 85, July 1985.

[KayK86] Kay, Timothy L., and Kajiya, James T., "Ray Tracing Complex Scenes," SIGGRAPH 86, **20**(4), August 1986, 269–278.
[Lind92] Lindley, Craig A., *Practical Ray Tracing in C*, John Wiley & Sons, Inc., 1992.
[Ritt90] Ritter, Jack, "A Simple Ray Rejection Test," in [Glas90], 385–386.
[Roth82] Roth, Scott D., "Ray Casting for Modeling Solids," CGIP, **18**(2), February 1982, 109–144.
[SunS92] Sung, Kelvin, and Shirley, Peter, "Ray Tracing with the BSP Tree," in [Kirk92], 271–274.
[WaCF92] Watkins, Christopher D., Coy, Stephen B., and Finlay, Mark, *Photorealism and Ray Tracing in C*, M&T Publ., Inc., 1992.

Real Analysis

[Nata61] Natanson, I.P., *Theory of Functions of a Real Variable*, translated from the Russian by Leo F. Boron, Frederick Ungar Publ. Co., 1961.
[Spie69] Spiegel, Murray R., *Theory and Problems of Real Variables*, Schaum's Outline Series, McGraw-Hill, Co., 1969.

Rendering

[Debe99] Debevec, Paul, organizer, *Image-Based Modeling, Rendering, and Lighting*, Course Notes, Volume 39, SIGGRAPH 99, August 1999.
[ElbC96] Elber, Gershon, and Cohen, Elaine, "Adaptive Isocurve-based Rendering for Freeform Surfaces," ACM TOG, **15**(3), July 1996, 249–263.
[Gree99] Greenberg, Donald P., "A Framework for Realistic Image Synthesis," CACM, **42**(8), August 1999, 45–53.
[Kaji86] Kajiya, James T., "The Rendering Equation," SIGGRAPH 86, **20**(4), August 1986, 143–150.
[MölH99] Möller, Tomas, and Haines, Eric, *Real-Time Rendering*, A.K. Peters, Ltd., 1999.
[WatW92] Watt, Alan, and Watt, Mark, *Advanced Animation and Rendering Techniques: Theory and Practice*, ACM Press, Addison-Wesley Publ. Co., 1992.
[Whit85] Whitted, T., "The Hacker's Guide to Making Pretty Pictures," Course Notes, Volume 12, SIGGRAPH 85, July 1985.

Robotics

[Crai89] Craig, John J., *Introduction to Robotics: Mechanics & Control*, 2nd Edition, Addison-Wesley Publ. Co., 1989.
[DenH55] Denavit, J., and Hartenberg, R.S., "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," Journal of Applied Mechanics, June 1955, 215–221.
[Feat87] Featherstone, Roy, *Robot Dynamics Algorithms*, Kluwer Academic Publishers, 1987.
[HwaA92] Hwang, Yong K., and Ahuja, Narendra, "Gross Motion Planning – A Survey," ACM Computin urveys, **24**(3), September 1992, 219–291.
[Paul82] Paul, Richard P., *Robot Manipulators: Mathematics, Programming and Control*, The MIT Press, 1982.

Shading and Illumination (Early Work)

[AtWG78] Atherton, P.R., Weiler, K., and Greenberg, D., "Polygon Shadow Generation," SIGGRAPH 78, **12**(3), August 1978, 275–281.
[BisW86] Bishop, G., and Weimer, D.M., "Fast Phong Shading," SIGGRAPH 86, **20**(4), August 1986, 103–106.
[Blin77] Blinn, J.F., "Models of Light Reflection for Computer Synthesized Pictures," SIGGRAPH 77, **11**(2), Summer, 1977, 192–198. Also in [Free80], 316–322.
[Bouk70] Bouknight, W.J., "A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Presentations," CACM, **13**(9), September 1970, 527–536. Also in [Free80], 292–301.
[BuiT75] Bui-Tuong, Phong, "Illumination for Computer Generated Pictures," CACM, **18**(6), June 1975, 311–317. Also in [BeaB82], 449–455.
[CooT82] Cook, Robert L., and Torrance, K.E., "A Reflectance Model for Computer Graphics," ACM TOG, **1**(1), January 1982, 7–24. Also in [JGMH88], 244–253.
[Crow77b] Crow, F.C., "Shadow Algorithms for Computer Graphics," SIGGRAPH 77, **11**(2), Summer, 1977, 242–247. Also in [BeaB82], 442–448.

[Duff79] Duff, T., "Smoothly Shaded Renderings of Polyhedral Objects on Raster displays," SIGGRAPH 79, **13**(2), August 1979, 270–275.

[GoTG84] Goral, C., Torrance, K.E., and Greenberg, D.P., "Modeling the Interaction of Light Between Diffuse Surfaces," SIGGRAPH 84, **18**(3), July 1984, 212–222.

[Gour71] Gouraud, H., "Continuous Shading of Curved Surfaces," IEEE Trans. on Computers, **C-20**(6), June 1971, 623–629. Also in [Free80], 302–308.

[TorS67] Torrance, K.E., and Sparrow, E.M., "Theory for Off-Specular Reflection from Roughened Surfaces," J. of the Optical Society of America, **56**(7), September 1967, 1105–1114.

[Whit80] Whitted, T., "An Improved Illumination Model for Shaded Display," CACM, **23**(6), June 1980, 343–349. Also in [JGMH88], 132–138.

[Will78] Williams, L., "Casting Curved Shadows on Curved Surfaces," SIGGRAPH 78, **12**(3), August 1978, 270–274.

[WoPF90] Woo, Andrew, Poulin, Pierre, and Fournier, Alain, "A Survey of Shadow Algorithms," CG&A, **10**(6), November 1990, 13–32.

Spatial Data Structures

[DocT81] Doctor, L., and Torborg, J., "Display Techniques for Octree-Encoded Objects," CG&A, **1**(3), 29–38.

[Meag82a] Meagher, D., "Geometric Modeling Using Octree Encoding," CGIP, **19**(2), June 1982, 129–147.

[Meag82b] Meagher, D., "Efficient Synthetic Image Generation of Arbitrary 3-D Objects," in Proc. of th EEE Computer Society Conference on Pattern Recognition and Image Processing, IEEE Computer Society Press, 1982.

[Same84] Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures," ACM Computing Surveys, **16**(2), June 1984, 187–260.

[Same90a] Samet, Hanan, *Design and Analysis of Spatial Data Structures*, Addison-Wesley Publ. Co., 1990.

[Same90b] Samet, Hanan, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley Publ. Co., 1990.

[SamW88] Samet, Hanan, and Webber, Robert E., "Hierarchical Data Structures and Algorithms for Computer Graphics: Part I: Fundamentals," CG&A, **8**(3), 1988, 48–68.

[YKFT84] Yamaguchi, K., Kunii, T.L., Fujimura, K., and Toriya, H., "Octree Related Data Structures and Algorithms," CG&A, **4**(1), 1984, 53–59.

Splines

[BarG89] Barry, P.J., and Goldman, R.N., "What Is the Natural Generalization of a Bézier Curve?," in [LycS89], 71–86.

[BaDD87] Barsky, B.A., DeRose, T.D., and Dippe, M.D., "An Adaptive Subdivision Method with Crack Prevention for Rendering Beta-spline Objects," Univ. of California, Berkeley, Computer Science Division, Technical Report UCB/CSD 87/348, 1987.

[Bars88] Barsky, Brian A., *Computer Graphics and Geometric Modeling Using Beta-splines*, Springer-Verlag, 1988.

[BarD89] Barsky, Brian A., and DeRose, Tony D., "Geometric Continuity of Parametric Curves: Three Equivalent Characterizations," CG&A, **9**(6), November 1989, 60–68.

[BarD90] Barsky, Brian A., and DeRose, Tony D., "Geometric Continuity of Parametric Curves: Construction of Geometrically Continuous Splines, CG&A, **10**(1), January 1990, 60–68.

[BaBB87] Bartels, Richard H., Beatty, John C., and Barsky, Brian A., *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, 1987.

[Bézi74] Bézier, P., "Mathematical and Practical Possibilities of UNISURF," in [BarR74], 127–152.

[Blin89a] Blinn, James F., "How many different cubic curves are there?," CG&A, **9**(3), May 1989, 78–83.

[Blin89b] Blinn, James F., "Cubic curve update," CG&A, **9**(6), November 1989, 70–73.

[Blin99] Blinn, James F., "How many rational parametric cubic curves are there? Part 1: Inflection points," CG&A, **19**(4), July/August 1999, 84–87.

[Blin00a] Blinn, James F., "How many different parametric cubic curves are there? Part 2: The same game," CG&A, **19**(6), November/December 1999, 88–92. Correction in CG&A, **20**(1), JanuaryFebruary 2000, 69.

[Blin00b] Blinn, James F., "How many rational parametric cubic curves are there? Part 3: The Catalog," CG&A, **20**(2), March/April 2000, 85–88.

[BloK02] Blomgren, Robert M., and Kasik, David J., "Early Investigation, Formulation and Use of NURBS at Boeing," Computer Graphics, **36**(3), August 2002, 27–32.

[Boeh80] Boehm, Wolfgang, "Inserting New Knots into B-spline Curves," CAD, **12**(4), July 1980, 199–201.

[CatR74] Catmull, Edwin, and Rom, Raphael, "A Class of Local Interpolating Splines," in [BarR74], 317–326.

[CoLR80] Cohen, E., Lyche, T., and Riesenfeld, R.F., "Discrete B-splines and Subdivision Techniques in Computer Aided Geometric Design and Computer Graphics," CGIP, **14**(2), October 1980, 87–111.

[DanD89] Daniel, M., and Daubisse, J., "The Numerical Problem of Using Bézier Curves and Surfaces in the Power Basis," CAGD, **6**(2), 1989, 121–128.

[deBo78] de Boor, Carl, *A Practical Guide to Splines*, Springer-Verlag, 1978.

[Fari89] Farin, Gerald, "Rational Curves and Surfaces," in [LycS89], 215–238.

[Fari92a] Farin, Gerald, "Degree Reduction Fairing of Cubic B-spline Curves," in [Barn92], 87–99.

[Fari95] Farin, Gerald, *NURB Curves and Surfaces: From Projective Geometry to Practical Use*, A.K. Peters, 1995.

[Faro91] Farouki, Rida T., "Computing with Barycentric Polynomials," The Mathematical Intelligencer, **13**(4), 1991, 61–69.

[FarR87] Farouki, R.T., and Rajan, V.T., "On the Numerical Condition of Polynomials in Bernstein Form," CAGD, **4**(1987), 191–216.

[FarR88] Farouki, R.T., and Rajan, V.T., "Algorithms for Polynomials in Bernstein Form," CAGD, **5**, 1988, 1–26.

[Forr72] Forrest, A., "Interactive Interpolation and Approximation by Bézier Polynomials," The Computer J., **15**(1), 1972, 71–79. Also in CAD **22**(9), 1990, 527–537.

[GorR74b] Gordon, William J., and Riesenfeld, Richard F., "B-Spline Curves and Surfaces," in [BarR74], 95–126.

[Greg74] Gregory, John A., "Smooth Interpolation Without Twist Constraints," in [BarR74], 71–87.

[LanR83] Lane, J.M., and Riesenfeld, R.F., "A Geometric Proof for the Variation Diminishing Property of B-spline Approximation," J. Approx. Theory, **37**, 1983, 1–4.

[LeeE82] Lee, E.T.Y., "A Simplified B-Spline Computation Routine," Computing, **29**, 1982, 365–371.

[LiuW02] Liu, Ligang, and Wang, Guojin, "Explicit Matrix Representation for NURBS Curves and Surfaces," CAGD, **19**(6), June 2002, 409–419.

[LooD89] Loop, Charles T., and DeRose, Tony D., "A Multisided Generalization of Bézier Surfaces," ACM TOG, **8**(3), July 1989, 204–234.

[LukC96] Luken, William L., and Cheng, Fuhua (Frank), "Comparison of Surface and Derivative Evaluation Methods for the Rendering of NURB Surfaces," ACM TOG, **15**(2), April 1996, 153–178.

[Malc77] Malcolm, Michael A., "On the Computation of Nonlinear Spline Functions," SIAM J. Numer. Anal., **14**(2), April 1977, 254–282.

[Mehl74] Mehlum, Even, "Nonlinear Splines," in [BarR74], 173–207.

[Niel74] Nielson, Gregory M., "Some Piecewise Polynomial Alternatives to Splines Under Tension," in [BarR74], 209–235.

[Pieg91] Piegl, Les, "On NURBS: A Survey," CG&A, **11**(1), January 1991, 55–71.

[PieT95] Piegl, Les, and Tiller, Wayne, *The NURBS Book*, Springer, 1995.

[PieT00] Piegl, Les, and Tiller, Wayne, "Reducing Control Points in Surface Interpolation," CG&A, **20**(5), September/October 2000, 70–74.

[PraG92] Prautzsch, H., and Gallagher, T., "Is There a Geometric Variation Diminishing Property for B-spline or Bézier Surfaces?," CAGD, **9**(2), 1992, 119–124.

[Rams88] Ramshaw, Lyle, "Béziers and B-Splines as Multiaffine Maps," in [Earn88], 757–776.

[Rams89] Ramshaw, Lyle, "Blossoms Are Polar Forms," CAGD, **6**(4), 1989, 323–359.

[Rasa90] Rasala, Richard, "Explicit Cubic Spline Interpolation Formulas," in [Glas90], 579–584.

[Rock93] Rockwood, Alyn, "A Brief Introduction to Blossoming," Course Notes, Volume 82, SIGGRAPH 93, August 1993.

[Roge01] Rogers, David F., *An Introduction to NURBS with Historical Perspective*, Morgan Kaufmann Publ., 2001.

[Scho46] Schoenberg, I.J., "Contributions to the Problem of Approximation of Equidistant Data by Analytic Functions," Quart. Appl. Math., **4**, 1946, 45–99.

[Scho67] Schoenberg, I.J., "On Spline Functions," in *Inequalities*, edited by O. Shisha, Academic Press, 1967, 255–291.

[Seid89] Seidel, Hans-Peter, "A New Multiaffine Approach to B-Splines," CAGD, **6**(1), 1989, 23–32.
[Seid93] Seidel, Hans-Peter, "An Introduction to Polar Forms," CG&A, **13**(1), January 1993, 38–46.
[ShaB84] Shani, U., and Ballard, D.H., "Splines as Embeddings for Generalized Cylinders," Computer Vision, Graphics and Image Processing, **27**, 1984, 129–156.
[StoD89] Stone, Maureen C., and DeRose, Tony D., "A Geometric Characterization of Parametric Cubic Curves," ACM TOG, **8**(3), July 1989, 147–163.
[SuLi83] Su, Bu-qing, and Liu, Ding-yuan, "An Affine Invariant and Its Application in Computational Geometry," Scientia Sinica, **26**, 1983, 259–272.
[Till83] Tiller, Wayne, "Rational B-Splines for Curve and Surface Representation," CG&A, **3**(5), September 1983, 61–69.
[Wang81] Wang, C.Y., "Shape Classification of the Parametric Cubic Curve and Parametric B-spline Cubic Curve," CAD, **13**(4), 1981, 199–206.
[Wern79] Werner, Helmut, "An Introduction to Non-Linear Splines," in *Polynomial and Spline Approximation: Theory and Applications*, Sahney, B.N., D. Reidel Publ. Co., 1979, 247–307.

Subdivision Curves and Surfaces

[CatC78] Catmull, E., and Clark, J., "Recursively Generated B-spline Surfaces on Arbitrary Topological Meshes," CAD, **10**(6), Nov. 78, 350–355.
[Chai74] Chaikin, G.M., "An Algorithm for High Speed Curve Generation," CGIP, **3**, Dec. 74, 346–349.
[DooS78] Doo, D.W.H., and Sabin, M.A., "Behavior of Recursive Subdivision Surfaces Near Extraordinary Points," CAD, **10**(6), Nov. 78, 356–360.
[Loop87] Loop, Charles, "Smooth Subdivision Surfaces Based on Triangles," Masters thesis, University of Utah, Department of Mathematics, 1987.
[Nasr00] Nasri, Ahmad H., "Recursive Subdivision of Polygonal Complexes and Its Applications in Computer-Aided Geometric Design," CAGD, **17**(7), August 2000, 595–619.
[Pete95] Peters, Jörg, "Smoothing Polyhedra Made Easy," ACM TOG, **14**(2), April 1995, 162–170.
[PetR97] Peters, Jörg, and Reif, Ulrich, "The Simplest Subdivision Scheme for Smoothing Polyhedra," ACM TOG, **16**(4), October 1997, 420–431.
[Reif95] Reif, Ulrich, "A Unified Approach to Subdivision Algorithms Near Extraordinary Vertices," CAGD, **12**(2), March 1995, 153–174.
[Ries75] Riesenfeld, R., "On Chaikin's Algorithm," CGIP, **4**(3), 1975, 304–310.
[Stam98] Stam, Jos, "Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values," SIGGRAPH 98, July 1998, 395–404.
[ZorS99] Zorin, Denis, and Schröder, Peter, Organizers, *Subdivision for Modeling and Animation*, Course Notes, Volume 37, SIGGRAPH 99, August 1999.

Surfaces and Manifolds

[ACDL00] Amenta, N., Choi, S., Dey, T.K., and Leekha, N., "A Simple Algorithm for Homeomorphic Surface Reconstruction," in *Proc. of the 16th Annual Symp. on Computational Geometry*, Hong Kong, June 12–14, 2000, ACM Press, 213–222.
[BeFH86] Beck, James M., Farouki, Rida T., and Hinds, John K., "Surface Analysis Methods," CG&A, **6**(12), December 1986, 18–36.
[BoiC00] Boissonnat, Jean-Daniel, and Cazals, Frédéric, "Smooth Surface Reconstruction via Natural Neighbour Interpolation of Distance Functions," in *Proc. of the 16th Annual Symp. on Computational Geometry*, Hong Kong, June 12–14, 2000, ACM Press, 223–232.
[HagH95] Hagen, H., and Hahmann, St., "Stability Concept for Surfaces," in [HaFN95], 189–198.
[Hage92] Hagen, H., Hahmann, S., Schreiber, T., Nakajima, Y., Wördenweber, B., and Hollemann-Grundstedt, P., "Surface Interrogation Algorithms," CG&A, **12**(5), September 1992, 53–60.
[MorS92] Moreton, Henry P., and Séquin, Carlo H., "Functional Optimization for Fair Surface Design," SIGGRAPH 92, **26**(2), July 1992, 167–176.
[Sarr98] Sarraga, Ramon F., "Recent Methods for Surface Shape Optimization," CAGD, **15**(5), May 1998, 417–436.
[YuMS01] Yu, Xiaohua, Morse, Bryan S., and Sederberg, Thomas W., "Image Reconstruction Using Data-Dependent Triangulation," CG&A, **21**(3), May/June 2001, 62–68.

Texture

[BieS86] Bier, Eric A., and Sloan, Kenneth R., Jr., "Two-Part Texture Mappings," CG&A, **6**(9), September 1986, 40–53.

[BliN76] Blinn, J.F., and Newell, M.E., "Texture and Reflections in Computer Generated Images," CACM, **19**(10), 1976, 542–547.

[Blin78] Blinn, J.F., "Simulation of Wrinkled Surfaces," SIGGRAPH 78, **12**(3), August 1978, 286–292. Also in [JGMH88].

[Gree86] Greene, Ned., "Environment Mapping and Other Applications of World Projections," CG&A, **6**(11), November 1986, 21–29.

[Hara79] Haralick, R.M., "Statistical and Structural Approaches to Texture," Proc. IEEE, **67**(5), May 1979, 786–804.

[Heck86] Heckbert, Paul S., "Survey of Texture Mapping," CG&A, **6**(11), November 1986, 56–67. Also in [JGMH88].

[Jule62] Julesz, B., "Visual Pattern Discrimination," IRE Trans. on Information Theory, Volume **8**, February 1962, 84–92.

[JulB81] Julesz, B., and Bergen, J.R., "Textons, The Fundamental Elements in Preattentive Vision and Perception of Textures," in *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, edited by Martin A. Fischler and Oscar Firschein, Morgan Kaufman Publ., Inc., 1987, 243–256.

[Neva82] Nevatia, Ramakant, *Machine Perception*, Prentice-Hall, Inc., 1982.

[WeiD97] Weinhaus, Frederick M., and Devarajan, Venkat, "Texture Mapping 3d Models of Real-World Scenes," ACM Computing Surveys, **29**(4), December 1997, 325–365.

[WooA98] Woo, Andrew, "Chordlength Texturing of Spline Surfaces," Journal of Graphics Tools, **3**(2), 1998, 15–19.

Topology

(See also Algebraic Topology)

[BurM71] Burgesser, H., and Mani, P., "Shellable Decompositions of Cells and Spheres," Math. Scand., **29**, 1971, 197–205.

[HurW48] Hurewicz, Witold, and Wallman, Henry, *Dimension Theory*, Princeton Univ. Press, 1948.

Trimmed Surfaces

[AnGC99] Anglada, Marc Vigo, Garcia, Núria Pla, and Crosa, Pere Brunet, "Directional Adaptive Surface Triangulation," CAGD, **16**(2), February 1999, 107–126.

[Brun95] Brunnett, G., "Geometric Design with Trimmed Surfaces," in [HaFN95], 101–115.

[Casa87] Casale, Malcom S., "Free-form Solid Modeling with Trimmed Surface Patches," CG&A, **7**(1), January 1987, 33–43.

[CaBU92] Casale, M.S., Bobrow, J.E., and Underwood, R., "Trimmed-patch Boundary Element: Bridging the Gap Between Solid Modeling and Engineering Analysis," CAD, **24**(4), 1992, 193–198.

[Chew93] Chew, L.P., "Guaranteed Quality Mesh Generation for Curved Surfaces," in Proceedings of the ACM Symposium on Computational Geometry, 1993, 274–280.

[CMPP99] Cho, W., Maekawa, T., Patrikalakis, N.M., and Peraire, J., "Topologically Reliable Approximation of Trimmed Polynomial Surface Patches," Graphical Models and Image Processing, **61**(1), 1999.

[ChPP98] Cho, Wonjoon, Patrikalakis, Nicholas M., and Peraire, Jaime, "Approximate Development of Trimmed Patches for Surface Tessellation," CAD, **30**(14), December 1998, 1077–1087.

[KumM94] Kumar, S., and Manocha, D., "Interactive Display of Large Scale Trimmed NURBS Models," Technical Report TR94-008, Dept. of Comp. Sci., University of North Carolina, USA, 1994.

[KumM95] Kumar, S., and Manocha, D., "Efficient Rendering of Trimmed NURBS Surfaces," CAD, **27**(7), 1995, 509–521.

[LasB95] Lasser, D., and Bonneau, G.P., "Bézier Representation of Trim Curves," in [HaFN95], 227–242.

[Luke96] Luken, William L., "Tessellation of Trimmed NURB Surfaces," CAGD, **13**(2), March 1996, 163–177.

[Peter94] Peterson, John W., "Tesselation of NURB Surfaces," in [Heck94], 286–320.

[PieR95] Piegl, L.A., and Richard, A.M., "Tessellating Trimmed NURBS Surfaces," CAD, **27**(1), January 1995, 16–26.

[PieT98] Piegl, L.A., and Tiller, W., "Geometry-Based Triangulation of Trimmed NURBS Surfaces," CAD, **30**(1), January 1998, 11–18.

[RoHD89] Rockwood, A., Heaton, K., and Davis, T., "Real-Time Rendering of Trimmed Surfaces," SIGGRAPH 89, **23**(3), July 1989, 107–117.

[SheH92] Sheng, X., and Hirsch, B.E., "Triangulation of Trimmed Surfaces in Parametric Space," CAD, **24**(8), August 1992, 437–444.

[VigB95] Vigo, M., and Brunet, P., "Piecewise Linear Approximation of Trimmed Surfaces," in [HaFN95], 341–356.

Virtual Reality

[Broo99] Brooks, Frederick P., Jr., "What's Real About Virtual Reality," CG&A, **19**(6), November/December 1999, 16–27.

[CMBZ00] Capps, Michael, McGregor, Don, Brutzman, Don, and Zyda, Michael, "NPSNET-V:A New Beginning for Dynamically Extensible Virtual Environments," CG&A, **20**(5), September/October 2000, 12–15.

[CrSD93] Cruz-Neira, C., Sandin, D.J., and DeFanti, T.A., "Surround-Screen Projection-Based Virtual Reality:The Design and Implementation of the CAVE," SIGGRAPH 93, August 1993, 135–142.

[HalM63] Hall, M.R., and Miller, J.W., "Head-Mounted Electro-Ocular Display: A New Display Concept for Specialized Environments," Aerospace Medicine, **34**(4), April 1963, 316–318.

[Suth65] Sutherland, I.E., "The Ultimate Display," invited lecture, IFIP Congress 65, see also *Proceedings IFIP Congress 65*, Volume 2, Kalenich, W.A., editor, Spartan Books and MacMillan, 506–508.

[VFLL00] Van Dam, Andries, Forsberg, Andrew S., LaidLaw, David H., LaViola, Joseph J., Jr., and Simpson, Rosemary M., "Immersive VR For Scientific Visualization: A Progress Report," CG&A, **20**(6), November/December 2000, 26–52.

Visible Surface Detection

[Appe67] Appel, Arthur, "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," Proc. ACM Nat. Conf., 1967, 387–393. Also in [Free80], 214–220.

[Blin81] Blinn, James, F., "A Scan Line Algorithm for Displaying Parametrically Defined Surfaces," SIGGRAPH 81 tutorials #L, also in Computer Graphics, Vol 12 (supplement to SIGGRAPH 78), also in [Boot79], 348–354.

[Carp84] Carpenter, L., "The A-buffer, an Antialiased Hidden Surface Method," SIGGRAPH 84, **18**(3), July 1984, 103–108.

[Catm74] Catmull, Edwin, "A Subdivision Algorithm for Computer Display of Curved Surfaces," Ph.d. thesis, University of Utah, 1974. Also as UTEC-CSc-74-133, Computer Science Dept., University of Utah, Salt Lake City, UT, December 1974.

[Catm75] Catmull, Edwin, "Computer Display of Curved Surfaces," in Proc. IEEE Conf. on Computer Graphics, Pattern Recognition and Data Structures, May 1975. Also in [Free80], 309–315.

[Catm78] Catmull, Edwin, "A Hidden-Surface Algorithm with Anti-aliasing," SIGGRAPH 78, **12**(3), August 1978, 6–11.

[Clar79] Clark, J.H., "A Fast Algorithm for Rendering Parametric Surfaces," SIGGRAPH 79, **13**(2), August 1979, 174 (abstract only). Also in [JGMH88], 88–93.

[FuAG83] Fuchs, H., Abram, G.D., and Grant, E.D., "Near Real-Time Shaded Display of Rigid Objects," SIGGRAPH 83, **17**(3), July 1983, 65–69.

[FuKN80] Fuchs, H., Kedem, Z.M., and Naylor, B.F., "On Visible Surface Generation by a Priori Tree Structures," SIGGRAPH 80, **14**(3), July 1980, 124–133.

[Grif75] Griffiths, J.G., "A Data-Structure for the Elimination of Hidden Surfaces by Patch Subdivision," CAD, **7**, July 1975, 171–178.

[Grif78a] Griffiths, J.G., "A Surface Display Algorithm," CAD, **10**(1), January 1978, 65–73.

[Grif78b] Griffiths, J.G., "Bibliography of Hidden-Line and Hidden-Surface Algorithms," CAD, **10**(3), 1978, 203–206.

[LCWB80] Lane, Jeffrey M., Carpenter, Loren C., Whitted, Turner, and Blinn, James F., "Scan Line Methods for Displaying Parametrically Defined Surfaces," CACM, **23**(1), January 1980, 23–34. Also in [JGMH88], 94–105.

[NeNS72] Newell, M.E., Newell, R.G., and Sancha, T.L., "A New Approach to the Shaded Picture Problem", Proc. ACM National Conference, 1972.

[Robe63] Roberts, L.G., "Machine Perception of Three Dimensional Solids," Lincoln Laboratory, TR 315, MIT, Cambridge, MA, May 1963. Also in Tippet, J.T., et al., editors, *Optical and Electro-Optical Information Processing*, MIT Press, Cambridge, MA, 1964, 159–197.

[SBGS69] Schumacker, R.A., Brand, B., Gilliland, M., and Sharp, W., *Study for Applying Computer-Generated Images to Visual Simulation*, Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, September 1969.

[SuSS74] Sutherland, I.E., Sproull, R.F., and Schumacker, R.A., "A Characterization of Ten Hidden Surface Algorithms," ACM Computing Surveys, **6**(1), March 1974, 1–55. Also in [BeaB82].

[Warn69] Warnock, J., *A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures*, Technical Report TR 4–15, NTIS AD-753 671, Computer Science Department, University of Utah, Salt Lake City, UT, June 1969.

[Watk70] Watkins, G.S., *A Real Time Visible Surface Algorithm*, Ph.D. thesis, Technical Report UTEC-CSc-70-101, NTIS AD-762 004, Computer Science Department, University of Utah, Salt Lake City, UT, June 1970.

[WeiA77] Weiler, K., and Atherton, P., "Hidden Surface Removal Using Polygon Area Sorting," SIGGRAPH 77, **11**(2), Summer, 1977, 214–222. Also in [JGMH88], 209–217.

Visualization

[Banc95] Banchoff, Thomas F., *Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions*, Scientific American Library, 1996.

[Gunn93] Gunn, Charlie, "Discrete Groups and Visualization of Three-Dimensional Manifolds," SIGGRAPH 93, August 1993, 255–262.

[HanH92] Hanson, Andrew J., and Heng, Pheng A., "Illuminating the Fourth Dimension," CG&A, **12**(4), July 1992, 54–62.

[HaMF94] Hanson, Andrew J., Munzner, Tamara, and Francis, George, "Interactive Methods for Visualizable Geometry," Computer, **27**(7), July 1994, 73–83.

[Week85] Weeks, J., "Hyperbolic Structures on 3-Manifolds," Ph.D. Dissertation, Princeton University, 1985.

Volume Rendering

[CohK97] Cohen-Or, Daniel, and Kaufman, Arie, "3D Line Voxelization and Connectivity Control," CG&A, **17**(6), November/December 1997, 80–87.

[DrCH88] Drebin, Robert A., Carpenter, Loren, and Hanrahan, Pat, "Volume Rendering," Computer Graphics, **22**(4), August 1988, 65–74.

[Elvi92] Elvins, T. Todd, "A Survey of Algorithms for Volume Visualization," Computer Graphics, **26**(3), August 1992, 194–201.

[FTAT00] Fujishiro, Issei, Takeshima, Yuriko, Azuma, Taeko, and Takahashi, Shigeo, "Volume Data Mining Using 3D Field Topology Analysis," CG&A, **20**(5), September/October 2000, 46–51.

[Kalv92] Kalvin, A.D., "A Survey of Algorithms for Constructing Surfaces from 3D Volume Data," IBM Research Report RC 17600 (#77606), IBM, Yorktown Heights, NY, January 1992.

[Kauf98] Kaufman, Arie, Organizer, *Advances in Volume Visualization*, Course Notes, Volume 24, SIGGRAPH 98, July 1998.

[KaCY93] Kaufman, Arie, Cohen, Daniel, and Yagel, Roni, "Volume Graphics," Computer, **26**(7), July 1993, 51–64.

[LacL94] Lacroute, P., and Levoy, Marc, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," SIGGRAPH 94, July 1994, 451–458.

[Levo88] Levoy, Marc, "Display of Surfaces from Volume Data," CG&A, **8**(3), May 1988, 29–37.

[Levo90] Levoy, Marc, "A Hybrid Ray Tracer for Rendering Polygon and Volume Data," CG&A, **10**(2), March 1990, 33–40.

[LiCN98] Lichtenbelt, Barthold, Crane, Randy, and Naqvi, Shaz, *Introduction to Volume Rendering*, Prentice Hall PTR, 1998.

[LorC87] Lorensen, William E., and Cline, Harvey E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," SIGGRAPH 87, **21**(4), July 1987, 163–169.

[NFMD90] Ney, Derek R., Fishman, Elliot K., Magid, Donna, and Drebin, Robert A., "Volumetric Rendering," CG&A, **10**(2), March 1990, 24–32.

[RhyT01] Rhyme, Theresa-Marie, and Treinish, Lloyd, editors, "Visualization Viewpoints: The Transfer Function Bake-Off," CG&A, **21**(3), May/June 2001, 16–22.

[ScML98] Schroeder, Will, Martin, Ken, and Lorensen, Bill, *The Visualization Toolkit*, 2nd Edition, Prentice Hall PTR, 1998.

[StFF91] Stytz, M.R., Frieder, G., and Frieder, O., "Three-Dimensional Medical Imaging: Algorithms and Computer Systems," ACM Computing Surveys, **23**(4), December 1991, 421–499.

[THBP90] Tiede, Ulf, Hoehne, Karl Heinz, Bomans, Michael, Pommert, Andreas, Riemer, Martin, and Wiebecke, Gunnar, "Investigation of Medical 3d-Rendering Algorithms," CG&A, **10**(2), March 1990, 41–53.

[West90] Westover, L., "Footprint Evaluation for Volume Rendering," SIGGRAPH 90, **24**(4), August 1990, 367–376.

[WilV90a] Wilhelms, Jane, and van Gelder, Allen, "Octrees for Faster Isosurface Generation," Computer Graphics, **24**(5), November 1990, 57–62.

[WilV90b] Wilhelms, Jane, and van Gelder, Allen, "Topological Considerations in Isosurface Generation," Computer Graphics, **24**(5), November 1990, 79–86.

[WyMW86] Wyvill, B., McPheeters, C., and Wyvill, G., "Data Structures for Soft Objects," The Visual Computer, **2**(4), August 1986, 227–234.

[YaCK92] Yagel, Roni, Cohen, Daniel, and Kaufman, Arie, "Discrete Ray Tracing," CG&A, **12**(5), 1992, 19–28.

[YooT98] Yoo, Terry, Organizer, *3D Visualization in Medicine*, Course Notes, Volume 23, SIGGRAPH 98, July 1998.

Index

A
Absolute curvature, 525
A-buffer algorithm, 276
Accuracy, 211
Active edge list, 49, 100, 276, 285, 606
Adapted frame field,
  relatively parallel, 463
Adaptive subdivision, 177, 178, 589
Adaptive supersampling, 343
Address
  of attractor point, 812
Adini twist vector, 499
Adjacency relation
  between cells, 200
Adjacent
  $(3^n - 1)$-, 24
  2n-, 24
  k-, 24
Adjacent cells, 200
Admissibility criterion
  in trimming, 603
Admissible
  triangle, 603
  triangulation, 603, 604
AEL, 49, 100
Affine map
  between vector spaces, 418
Affinely invariant, 386, 403, 466, 491, 499, 502, 505, 528, 630
Algebraic coefficients
  of bicubic patch, 496
  of cubic curve, 391
Algebraic geometry
  in curve tracing, 615
  issues with, 619
Algorithmic modeling, 215
Aliasing, 32, 45, 342, 357, 368, 767, 790
Angle at vertex
  of face, 662
  of pwl curve, 662
Angle counting test, 235
Angle deficit method, 651
Angle of incidence, 312, 315, 320
Angle of refraction, 320
Angle parameterization, 593
Animation, 61, 141
Annotating, 223
Annulus, 476
Antarctic region, 667
Antialiasing, 32, 45, 49, 291, 343
  A-buffer algorithm, 291
  bump maps, 330
  in marching cube algorithm, 368
  in textures, 327
  in visible surface determination, 290
  in volume rendering, 362
  lines, 46
  polygons, 46, 53
  radiosity, 357
  Warnock algorithm, 291
  with mip-mappings, 327
  with postfiltering, 47
  with prefiltering, 46
  with supersampling, 47
Aperture color, 296
API, 11, 13, 15, 16, 64
Application programming interface, 11
Approximation problem
  general, 375
Aqueous humor, 295

AR, 687
Arc
  in medial axis, 184
Arc length, 633, 638
  C^k continuity, 455
  computation of, 635
  for Bézier curves, 637
Arc-length parametrization, 634, 637, 638, 652
Arctic region, 667
Area
  of parallelogram, 249
  of set in picture, 25
  of polygon, 250
  of subtended region, 251
  of triangle, 250
Area sampling, 265
Artificial intelligence, 688
Artzy's algorithm
  for volume rendering, 362
Aspect ratio, 120
  hardware, 120
Assembly features, 194
Assignment
  of points on trimlines, 680
Asymptotic curve, 526
Attractor
  of iterated function system, 810
Augmented reality, 687
Axis-parallel box, 697
Axonometric projection, 133
Axonometric view, 133

B
Back clipping plane, 5
Back edge, 266
Back face, 267
  removal of, 267
Background
  of set in picture, 25
Balanced binary search tree, 698, 718
Band-limited function, 44, 788
Barnhill-Kersey algorithm, 561, 569, 573
  comparison with others, 566
  data structures for, 565
  hunting phase for, 561
  relaxing points in, 562
  sorting phase for, 566
  tolerances in, 562, 563, 565
  tracing phase for, 563
Barycentric coordinate property
  of affine maps, 418
Barycentric coordinate test, 233
Base frame, 142
Base surface, 679
Basic fill algorithm, 29
Basin of attraction, 220, 579, 800
Basis function, 375
Basis functions, 385, 455, 751
  Haar wavelet, 793
  properties of, 387
  wavelet, 792
Beam tracing, 343
Bernstein polynomial, 398, 507, 549
Bessel end condition, 388
Beta constraint, 454
Beta-spline, 455
Beveling, 222
Bézier basis function, 399
Bézier clipping, 547
Bézier coefficients, 397, 501
Bézier control net, 508
Bézier curve, 398, 403, 432, 466
  2nd derivative of, 402
  advantages of, 403
  affinely invariant, 403
  as B-spline curve, 418
  blossom of, 422, 469
  convex hull property of, 399
  cubic, 398
  degree elevation of, 452
  derivative of, 402, 459
  derivatives of, 424
  differences with B-spline curve, 466
  disadvantages of, 403
  invariant under affine parameter transformations, 403
  numerical stability of, 467
  piecewise, 404
  properties of, 399, 402, 403
  rational, 432
  similarities with B-spline curve, 466
  subdivision, 451
  use of, 403
  variation diminishing property of, 466
Bézier matrix, 397
Bézier patches
  geometric continuity of, 522
Bézier point, 399, 422
Bézier polygon, 399

Bézier polynomials, 422
Bézier representation
  of polynomial, 422
Bézier surface, 487, 502
  affinely invariant, 502
  as B-spline surface, 505
  control points of, 502
  convex hull property of, 502
  cubic, 501
  partial derivatives of, 502
  rational, 512
  triangular, 508
Bézout's theorem, 572
BFA, 29
BGI, 10, 13, 15
Bias, 454
Bicubic patch, 496, 499
  affinely invariant, 499
Bicubic patches
  geometric continuity of, 523
Bilinear surface, 486, 488
Binary search tree
  balanced, 698
Binary space partitioning, 270
Biquintic Bézier mesh, 645
Bit block transfer, 60
Bit map, 59
Bit plane extraction, 65
BitBlt, 60, 66, 181
Blending, 222, 504, 517, 598, 672, 673
  based on parametric surfaces, 679, 680
  as boundary value problem, 682
  extent constraints of, 673
  Fourier methods for, 682
  global, 673
  of atoms, 673
  polyhedral, 673, 681
  potential method for, 676
  range constrained, 676
  rolling ball, 678, 681
  superelliptic, 676
  superficial, 672
  surface, 672
  using conics, 674, 677
  variable radius rolling ball, 678
  volume bounded, 674
  volumetric, 673
Blending arc, 680
Blending curve, 222
Blending function, 386
Blending functions, 392
Blending surface, 222, 679
Blinn visible surface algorithm, 266
  summary of, 289
Blossom, 419, 420, 521
  of Bézier curve, 469
  tensor product, 505
  triangular, 506
Blossoming theorem, 419
Body sheet
  in medial axis, 184
Boolean sum operator, 490
Border
  of set in picture, 25
Border following algorithm, 26
Borland graphics interface, 10
Bouknight reflectance model, 312, 333
Boundary, 29
Boundary data structures, 199
  comparison of, 202
Boundary evaluation
  of CSG object, 170, 557
Boundary fill, 28
Boundary representation, 166, 175, 183, 221, 224
  advantages of, 220, 198
  conditions for validity, 167
  data structures of, 199
  disadvantages of, 220
Boundary value problem
  for geodesics, 655
  in blending, 682
Bounded rational arithmetic, 213
Bounded variation, 779, 783
Bounding box, 228, 340, 544, 545, 562, 573
  generalized, 230
  oriented, 573
Bounding object, 228, 340
Bounding object hierarchy, 340
Bounding sphere, 230, 322
Box, 228
  axis-parallel, 697
Box function, 783, 784, 785, 790, 791, 792
Branch node
  of machine, 218
Branch point, 564
Branching problem
  for skinning, 631
B-rep, 166, 187, 188, 194, 195, 198, 210, 673
B-rep-to-CSG conversion, 208, 210

Bresenham
  circle-drawing algorithm, 56
  improved circle-drawing algorithm, 58
  line-drawing algorithm, 39, 41, 362
Bridging, 668
Brightness
  perceived, 296
Brilliance
  perceived, 296
BSP algorithm, 265, 266, 270, 332
  orthogonal, 273
BSP tree, 270, 273, 342
  construction of, 270
  orthogonal, 273
  traversal of, 271
B-spline, 408
  barycentric coordinate property of, 418
  compact support of, 411
  definitions of, 404
  desirable properties of, 406
  differentiability of, 411
  linear, 405
  of degree m, 408
  of order k, 408
  properties of, 411
  recursive definition of, 407
B-spline basis function, 408
  see B-spline
B-spline coefficients, 415
B-spline curve, 411, 432, 466
  as Bézier curves, 427, 428
  clamped, 408, 411
  closed cubic uniform, 416
  closed quadratic uniform, 416
  control polygon, 411
  convex hull property of, 417
  derivatives of, 438
  differences with Bézier curve, 466
  differences with Hermite curve, 467
  domain of, 411
  evaluation of, 438
  knot insertion, 428
  local control property, 411
  local convex hull property, 417
  matrix of, 414
  nonperiodic, 409
  nonuniform, 408, 411
  nonuniform rational, 433
  of degree m, 411
  of order k, 411
  open uniform, 408
  periodic, 408, 411
  properties of, 417
  rational, 432
  segment of, 411
  similarities with Bézier curve, 466
  unclamped, 408, 411
  uniform, 408, 411
  variation diminishing property of, 430
B-spline matrix
  cubic periodic/uniform, 414
  quadratic periodic/uniform, 414
B-spline surface, 504
  bicubic, 504
  convex hull property of, 504
  degree, 504
  domain of, 504
  evaluation of, 514
  local control of, 504
  local convex hull property of, 504
  order, 504
  partial derivatives of, 515
  rational, 512
  u-knot, 504
  v-knot, 504
Bubble meshing, 603
Buckle
  in interpolating spline, 459
Buffer
  frame, 8
  refresh, 8
  stencil, 9
  texture, 9, 325
  z-, 9, 275
BUILD, 172, 194, 196
Bump map, 328, 329
Buttock
  of ship, 533
By-polygon rendering, 330
By-scan-line rendering, 330

C

CAD, v, 156, 169, 180, 181, 190, 191, 212, 445, 467, 588, 601, 614, 649, 686, 687, 688, 714, 822
CADD, 175
CAGD, v, 181, 373, 377, 378, 445, 456, 480, 487, 495, 517, 520, 521, 649, 674, 687, 714, 822
CAM, 180, 588, 687, 688

Camera, 4, 113
  pinhole, 113
  synthetic, 113
Camera calibration, 139, 141
Camera coordinate system, 4, 113, 131
Camera data, 113
Camera frame, 113
Camera-to-clip-space transformation, 122, 124, 622
CAM-I, 194, 195
Canal surface, 678
Canonical intersection term, 209
Canonical subset, 706
Carpenter-Lane visible surface algorithm, 290
Cartesian product surface, 472, 495
Category, 225
Cathode ray tube, 7
Catmull visible surface algorithm, 284
Catmull-Clark subdivision algorithm, 529, 530
Cauchy-Riemann equations, 768
CAVE, 686
Cayley numbers, 756
Cel painter, 28
Cell decomposition, 178
Cell decomposition approach
  in surface tiling, 595
CGA, 8, 9
Chaikin curve subdivision algorithm, 465, 527
Chamfer, 672
Chamfering, 222
Chaos game, 21, 810
Chaotic map, 798, 799
Characteristic function
  of a set, 817
Characteristic point, 545, 641
Characteristic polygon, 399
Chordal deviation, 589
Chord length, 632, 633
Chromaticity diagram
  CIE, 299, 300
  Maxwell triangle, 298
Chromaticity values, 297, 298
CIE color specification, 299
CIE illuminants, 297
CIE tri-stimulus values, 297
Circle
  parametrization of, 54
Circle drawing algorithms
  Bresenham, 55
  DDA approach to, 54
  other, 57
Circle of latitude
  of surface of revolution, 475
C^k
  function vs set, 374
Clamped end condition, 388
Clark visible surface algorithm, 290
Clip, 69
Clip coordinate system, 122
  homogeneous, 122
Clip polygon, 69
Clip space, 122
  homogeneos, 122
Clipping, 5
  Bézier, 547
  Blinn, 125
  bogus edges in, 84, 85, 87, 98
  Cohen-Sutherland, 71
  Cyrus-Beck, 73
  generalized Cohen-Sutherland, 73
  Greiner-Hormann, 106
  Liang-Barsky, 77, 86
  Maillot, 89
  Nicholl-Lee-Nicholl, 81
  Sutherland-Hodgman, 84
  text, 110
  Vatti, 98
  Weiler, 85
Clipping plane
  back, 5
  far, 5, 138
  front, 5, 130
  hither, 5
  near, 5, 138
  yon, 5
Clothoid, 447, 643, 650
  application of, 447
  generalized, 447
Cluster, 268
CMY color model, 299
Coefficients
  algebraic, 391, 496
  geometric, 391, 496
  Hermite, 391
Cohen-Sutherland line clipping, 71, 80, 89, 93, 97, 125, 126, 356

Coherence, 49, 285, 291, 632
  area, 273
  depth, 291
  edge, 276, 291
  face, 291
  frame, 291
  geometric, 291
  object, 291
  point-to-point, 278
  scan line, 51, 291
  spatial, 342
Collage theorem, 811
Color
  achromatic, 296
  aperture, 296
  chromatic, 296
  definition of, 295
  film, 296
  misconceptions of, 294
  nonspectral, 297
  perceived, 295
  spectral, 297
Color cube, 299
Color cycling, 65
Color model
  CMY, 299
  cube, 299, 300
  HSL triangle, 301
  HSV hexcone, 301
  RGB, 299, 300
  YIQ, 300
Color models
  conversion between, 303
Color response, 297
Colorimetry, 295, 297
Combinatorial topology, 167, 171
Combinatorial validity, 171, 173
Commutative diagram, 818
Compatibility condition, 494, 504
Complement
  of set in picture, 25
Component, 24
Composite
  k-fold, 798
  of two curves, 453
Composite functions
  representation of, 614
Composite material, 655, 667
Compositing, 360
Computable map
  over ring R, 219
Computation, 219
  length of, 219
Computation node
  of machine, 218
Computational geometry, 199, 697
Compute function
  with machine, 219
Computed tomography, 181
Computer vision, 338
Condensation set, 810
Condensation transformation, 810
Condition number
  of matrix, 570, 619
Cone, 483
  infinite, 664
  truncated, 476
Cone receptor cell
  in eye, 295
Cone tracing, 343
Conic
  as projection of parabola, 431
  as rational curve, 431
  general equation of, 59
  representation of, 215
Conic drawing algorithms, 57
Conjugation map
  of quaternions, 758
Connected, 24
Connecting pwl curve, 660
Constrained minimization, 619, 742
Constraint-based modeling, 191, 193
Constraint graph, 198
Constraints
  interpolatory, 376
  orthogonality, 377
  shape, 377
  variational, 377
Constraint solutions, 735
Constructive solid geometry, 157, 167
Contact curve, 598, 674
Continuation method, 570
Contour, 361, 362, 624
  as implicit representation, 624
  of set, 553
Contour algorithm, 362, 554, 625
  higher dimensional, 630
Contour line, 526

Contour problem, 624
  discrete, 625
Contraction map, 809
  contractility factor of, 809
Contraction mapping theorem, 809
Control net
  Bézier, 508
  triangular, 508
Control points
  of Bézier curve, 399
  of Bézier surface, 502
  of B-spline curve, 411
  of B-spline surface, 504
  of Gregory patch, 503
  of polynomial, 422
  of rational Bézier curve, 433
  of rational Bézier surface, 512
  of rational B-spline curve, 433
  of rational B-spline surface, 512
Control polygon, 544
  Bézier curve, 399
  B-spline curve, 411
Convex hull property
  see Bézier/B-spline/NURBS curve/surface
  for Doo-Sabin surfaces, 528
Convexity test
  for polygon, 239
Convolution, 186, 786
  properties of, 788
Convolution theorem, 788
Coons patch, 489
  bicubic, 491
  bilinearly blended, 489, 499
  generalized bicubic, 492
  triangular, 494
Coons surface, 487, 489
  affinely invariant, 491
  bicubic, 491
  bilinearly blended, 489, 491
  generalized bicubic, 492
Coordinate system, 111
Core, 11
Cornea, 295
Corner cutting
  subdivision algorithm, 465, 526, 527
Corner point
  of superellipse, 448
Cornu spiral, 447
  generalized, 447
Correpondence problem
  for skinning, 631
Correspondence rule
  for directrix-generator representation, 473
Cox-de Boor B-spline definition, 407, 438, 516
Coxeter triangulation, 627
Cranfield object, 679
Critical angle, 322
Cross-ratio, 435
Crossing curve
  for blending surface, 680
Crossings test, 234
CRT, 7, 46, 63, 66, 121, 334
CSG, 157, 167, 177, 181, 193, 195, 210, 220, 557, 598, 673
CSG object, 186, 206, 207
  medial axis representation, 186
  ray tracing, 348
  Voronoi diagram, 186
CSG representation, 168, 198, 208, 209, 221, 224
  advantages of, 169, 220
  disadvantages of, 170, 220
Csg-rep, 168, 178
CSG-to-b-rep conversion, 205
CSG-tree, 168, 226
CT, 181, 362, 593, 594
Curry-Schoenberg theorem, 430
Cursor, 9, 61
Curtain fold, 288
Curvature, 188, 455, 460, 526, 590, 599, 604, 639, 645, 649, 668
  and curves, 650
  and surfaces, 651
  for degenerate patches, 651
  for polygonal objects, 652, 657
  from discrete data, 650
  line, 520
Curvature continuous curve, 455
Curvature continuous surfaces, 651
  along curve, 651
Curve, 24, 373
  bias of, 454
  cubic, 377, 390
  curvature continuous, 455
  digital/discrete, 24
  evolute of, 641
  fair, 460
  four point, 395

G^k continuous, 453
globally parameterizable, 741
Hermite, 385, 391
implicit, 177
interrogation of, 460
Lagrange, 378, 379
linear, 377
offset, 177
parametric, 373
piecewise linear, 657
polynomial, 377
projection on surface, 671
pwl, 657
quadratic, 377
recursive subdivision, 465
spline, 377
subdivision algorithm for, 465
subdivision problem for, 449
tension of, 454
tracing, 615
unit quaternion integral, 650
Curve-bounding area, 546
Curve fitting problem, 461
Curve intersection algorithm
algebraic, 551
divide-and-conquer, 547
Newton-Raphson, 546
planar Bézier, 547
recursive subdivision, 547
Curve simplification, 597
Cusp, 456, 645
extraordinary, 641
ordinary, 641
Cusp line, 458
Cutoff frequency
of band-limited function, 788
Cutting and pasting, 171, 172
Cyclical overlap of figures, 270
Cyclide, 517, 678
Bézier parameterization of, 521
central, 517, 519, 520
characteristic lines of, 520
horned, 520
intersection of, 579
inversion of, 520
principal patch of, 520
ring, 519, 520
spindle, 520
Cylinder, 476, 483
Cyrus-Beck line clipping, 73, 125, 346

D
Data classification
in volume rendering, 358
Data structures
for boundary representations, 199
for cell complexes, 199
for volumes, 203
D-border, 26
DDA, 37, 54, 58
simple, 38
symmetric, 38
de Boor algorithm, 426
de Boor point, 411, 425
de Boor polygon, 411
de Casteljau algorithm, 401, 452, 469, 550, 593
for triangular patches, 508, 509, 521
Decidable set, 219
Decimation, 597, 687
Decomposition scheme, 178
adaptive, 178
object based, 178
space based, 178, 180
uniform, 178
Degree elevation, 632
Degrees of freedom, 142
Delaunay cell complex, 723, 724
face of, 723
Delaunay graph, 723, 724
Delaunay tetrahedralization, 603
Delaunay triangulation, 185, 201, 603, 604, 722, 723, 724, 725
direct computation of, 725
De Moivre theorem, 761
Denavit-Hartenberg notation, 146
Depth sorting, 269, 274, 292
Design
by feature, 196
extensional view of, 687
intentional view of, 687
Design stage
for boat, 532
DESIGNBASE, 173
Detail coefficients, 795
Developable surface, 483, 668
Deviation criterion
for surface triangulation, 600
Device independent, 10
Device independent code, 10

Diagonal
  of polygon, 715
Diagonal map, 419, 420
Diagonally dominant, 390
Differential equation
  first order, 36
  for circle, 54
  for ellipse, 58
  for straight line, 37
Digital differential analyzer, 37
Digital image processing, 767, 792
Dimension, 802
  box-counting, 804
  effective, 806
  fractal, 803, 804
  Hausdorff-Besicovitch, 804
  similarity, 813
  topological, 802
Dimensioning, 223
Dirac delta function, 783
Direct access
  to adjacency information, 200
Direct3D, 11
DirectDraw, 11
Directrix
  of surface, 473
Directrix-generator representation
  of surfaces, 473
DirectX, 11, 14, 15, 64
Dirichlet problem, 772, 778, 782
Discrete geodesic problem, 659
Discrete line
  attributes of, 35
Discrete topology, 22, 362
Discretize, 588
Disk
  n-dimensional, 816
Distance, 25
  between L^∞ functions, 775
  between L^p functions, 774
  Euclidean, 25
  from line to line, 248
  from point to line, 245
  from point to plane, 246
  max, 25
  taxicab, 25
Distance algorithms, 245
  curve-curve, 583
  curve-surface, 586
  overview of, 583
  point-curve, 583
  point-surface, 585
  surface-surface, 586
Distance function
  for medial axis, 184, 186
Distributed ray tracing, 344
Dither
  ordered, 48
Dither matrix, 48
Dithering, 48
Divide-and-conquer, 169, 551, 566, 631, 725
Division algebra, 755
D-move, 55
Domain
  of B-spline curve, 411
  of B-spline surface, 504
Doo-Sabin subdivision algorithm, 527
  improved, 529
  properties of, 528
DOS, 10, 11, 13, 14, 15
Double buffering, 66
Double-Blutel surface, 521
Doubly ruled surface, 483, 489
Douglas-Peucker algorithm, 598
Draw procedure, 15, 16
DrawLine procedure, 16
Dropping curve
  on surface, 670
Dupin cyclide, 517, 521, 650
Dupin indicatrix, 651
Dynamical system, 797, 806

E

Edge fill algorithms, 32, 291
Edge sequence, 660
  defined by pwl curve, 660
  shortest, 663
  simple, 660
Edge sequence tree, 665
Edge tracker, 286
Edge-adjacent faces, 660
  sequence of, 660
  simple sequence of, 660
Editing operations
  in modelers, 222
  with medial axis, 187
EGA, 8, 9
Eigenvalue, 578
Element
  in FEM, 746

Elementary collapse, 171
Elementary expansion, 171
Elementary interval, 706
Elementary subdivision
  of pwl curve, 658
  proper, 658
Ellipse
  parametrization of, 58
Ellipse drawing algorithm, 57
Elliptic paraboloid, 480, 514
Embedding method, 576
End condition
  Bessel, 388
  clamped, 388
  natural, 388
  periodic, 388
End conditions
  comparison of, 389
End-effector, 142
Endoskeleton, 186
Energy function approach
  to curve fitting, 461
Entering list
  in Weiler polygon clipping, 85
Entry value
  for Liang-Barsky line clipping, 78
Envelope, 646
  of curves, 646
Environment mapping, 327
Equator, 667
Erase procedure, 15
Erep, 221
Error function, 55
Euclidean metric, 26
Euclidean space
  n-dimensional, 815
Euler angle, 765
Euler characteristic, 171, 172
Euler operation representation, 172
Euler operations, 171, 681
Euler theorem, 650
Euler's method, 36
Evolute, 641
Exact arithmetic, 185
Exit value
  for Liang-Barsky line clipping, 78
Exoskeleton, 186
Expert system, 688
Exponential notation
  of pure quaternion, 761
Extended real numbers, 815
Extreme point
  of convex set, 714
Extrusion, 222, 483
Eye coordinate system, 131

F
Face
  back, 267
  definition of, 170
  front, 267
  representation of, 171
  shadowed, 664
Faceted display, 165
Faceted representation, 165, 223
Faceted surface sectioning algorithm, 555
Fair
  curve, 460
  set of points, 461
Fairing curves, 460, 650
  quadratic Bézier, 650
Fairing stage
  for boat, 534
Fairing surfaces, 521, 525, 526, 650, 651, 672
  discrete surface, 526
  isophote method, 526
  reflection line method, 525
  subdivision surface, 526
  using variational approaches, 525
Far clipping plane, 5, 138
Far plane, 229
Fast Fourier transform, 786
Fat arc, 547
Fat line, 547
Fat plane, 547, 573
FEA, 745
Feasable region, 735
Feature, 194
  characteristics of, 194
Feature-based modeling, 192, 687
Feature mapping, 195, 198
Feature model, 194
Feature recognition, 198
Feature validation, 195, 196
FEM, 745
Ferguson patch, 499, 681
FFT, 786
Filament winding, 667
  helical, 668

polar, 668
Fill algorithms
antialiasing in, 32, 53
boundary fill, 28
comparison of, 53
edge filling, 28, 32
flood fill, 28
ordered edge list, 51
parity check, 32
pixel based, 28
polygon based, 28, 50
seed fill, 28
types of,
Fillet, 222, 598, 672, 678
Filleting, 222
Film color, 296
Filter, 791
band-pass, 792
high-pass, 792
low-pass, 792
reconstruction, 792
Finite difference method, 745
Finite element
in FEM, 746
Finite element analysis, 592, 718
Finite element mesh
generation of, 188, 592
in FEM, 746
Finite element method, 745
advantages of, 754
basic steps of, 753
Galerkin, 749, 750
variational, 748
Finite element modeling, 179, 588
Fishkin seed fill algorithm, 32, 33
Fixed point
attractive, 800
repelling, 800
Flatness of curves/surfaces, 561, 573, 589, 591, 601, 602, 635, 649
Flood fill, 28
Floor function, 817
Floyd-Steinberg algorithm, 48
Fluid
circulation free, 771
circulation of, 771
incompressible, 771
irrotational, 771
nonviscous, 771
Fluid flow
stationary, 771
steady, 771
Folded edge, 286
Font, 109
bit-mapped, 109
outline, 109
vector, 109
Form factors, 351, 355
Form features, 194
Forward difference operator
rth, 423
Forward differences, 423
Forward kinematics, 142
Four point matrix, 396
Fourier coefficients, 777, 781
Fourier methods
in blending and fairing, 682
Fourier series
convergence of, 780
for a function, 777
Fourier transform, 44, 324
discrete, 786
of function of one variable, 781
of function of two variables, 785
Fractal, 3, 188, 216, 579, 805
Fractal dimension, 803, 804
Fractal space, 809
Fractional cascading, 702, 703
Frame
camera, 113
in $\mathbf{R}^n$, 111
link, 143
Frame buffer, 8, 64
Free-form surface, 643
Frenet frame, 461, 484, 485, 643
Frenet frame continuity, 456
Frequency domain, 785
Fresnel integral, 447
Fresnel reflection law, 314, 315
Front clipping plane, 5, 130
Front face, 267
Functional features, 194
Functor, 225

G

Galerkin method, 746, 749, 753
Gamma correction, 335
Gamut, 299
natural, 299

Gauss curvature, 484, 525, 591, 644, 645, 650, 651, 668, 669
Gaussian function, 46, 314, 783
  standard deviation, 314, 783
  variance, 783
Gauss map, 580
Gaussian quadrature, 635, 636
Gauss-Seidel algorithm, 352, 353
Gauss-Seidel radiosity algorithm, 353
GDI, 11, 14
General sweep, 174
Generalized bounding box, 230, 348
Generalized cylinder, 175
Generalized function, 783
Generalized geodesic, 670
Generalized inverse matrix,
Generative model, 175, 484
Generative modeling, 177
Generative modeling representation, 177
Generator
  for blending surface, 680
  generative model, 175
Generator rule
  for directrix-generator representation, 473
Generic halfspace, 159
Generic inclusion function, 733
Genetic algorithms, 199
GENMOD, 172, 175, 177, 726
Geodesic, 652, 668, 669, 670, 671
  discrete, 659
  from tessellation, 655
  generalized, 670
  kinematic definition, 652
  static force definition, 652
Geodesic path, 652, 654
GEOMED, 172
Geometric coefficients
  of bicubic patch, 496
  of cubic curve, 391
Geometric continuity
  kth order, 453, 522
  of Bézier curves, 455
  of Bézier patches, 522, 523
  of bicubic patches, 523
Geometric features, 194
Geometric matrix
  for bicubic patch, 496
Geometric modeling, 157
  geometrically intelligent, 688
  intrinsic, 684
  things left to do in, 687, 688
Geometrical optics, 310
G^k continuity, 453, 522
GKS, 11
Global scaling, 135, 136
Global shape function, 749
Globally parametrizable, 741
  in ith coordinate, 741
Gordon surface, 494
Gouging
  of NC cutter, 639
Gouraud shading, 316, 331, 352
GQ, 635
Graftal, 190, 216
Grammar-based model, 190
Graphical user interface, 13, 375
Graphics device interface, 11
Graphics mode, 13
Graphics standards
  hardware, 8
Grassfire algorithm, 185
Gregory patch, 503, 531, 681
  control points of, 503
  convex hull property of, 503
Gregory square, 494
Greiner-Hormann polygon clipping, 70, 71, 98, 106
Grid method
  for sections, 554
Grid test, 238
Gröbner basis, 575, 619
GUI, 13
Gupta-Sproull algorithm, 46
GWB, 173

H

Haar transform, 796
Haar wavelet, 793
Haar wavelet basis, 793
Halfplane, 711, 720
  lower, 815
  open, 711
  upper, 815
Halfspace, 159
  generic, 159
Halftoning, 48
Halting set, 219
Handle decomposition
  of manifold, 179, 188, 624
Haptic system, 182, 686
Hard area flooding, 32
Harmonic function, 768

Hat function, 405, 751, 795
Hausdorff metric, 728, 809
Hausdorff p-dimensional measure, 804, 813
Hausdorff-Besicovitch dimension, 804
Head-mounted display, 685
Heat flow equation
  strong form, 750
  weak form, 751
Heaviside function, 784
Hemicube method, 355
Hercules graphics standard, 8, 9
Hermite
  basis functions, 383, 384, 491
  coefficients, 391
  interpolating curve, 385, 391
  interpolation problem, 383
  matrix, 383, 392, 501
Hessian, 288
Hexcone color model, 301
Hidden line removal, 266
Hidden surface removal, 264
  see visible surface removal
Hilbert space, 774
  inner product of, 775
Hither clipping plane, 5
HMD, 685
Hodograph, 459
Holes
  of set in picture, 25
Homogeneous clip coordinate system, 122
Homogeneous clip space, 122
Homogeneous coordinates
  pro and con, 134
  problems with, 137
Homotopy continuation method, 570
Homotopy method, 570
Horizontal retrace, 7
HSL triangle color model, 301
HSV hexcone color model, 301
Hue
  nonspectral, 297
  perceived, 296
  spectral, 297
  unique, 296
  unitary, 296
Hunting phase
  in surface intersection computations, 558, 559, 561
Hyperbolic paraboloid, 483, 484, 514
Hyperboloid, 481
  of one sheet, 480, 483, 484
  of revolution, 480
Hypercube, 693

I

IBM, PC, 8, 59
Ideal function
  in interval analysis, 732
Identity matrix, 817
IFS, 810
IGES, 205, 211, 467, 822
Illumination model, 308
Illumination models
  overview, 309
Illumination pipelines
  global, 332
  local, 330
Image-based lighting, 338
Image-based modeling, 338
Image-based rendering, 338
Image enhancement, 788
Image precision algorithm, 264
  area sampling, 265
  point sampling, 265
Image reconstruction, 633, 788
Image synthesis, 308
Implicit curve algorithm
  marching method, 615
  problems with, 621
  rasterization, 614
  using algebraic geometry, 615
  via interval analysis, 740
Implicit function theorem, 259
Implicit representation
  and contours, 624
  versus parametrization, 259
Implicit surface
  parameterization of, 574
  nendering of, 622
Implicit surface algorithm
  marching, 624
  using interval analysis, 624
  via handle decomposition, 624
Impulse signal or function, 783, 792
In-betweening, 147
Inclusion function, 731
  convergent, 732
  excess width of, 732
  generic, 733

Inclusion isotone, 728
Inclusion monotonic, 728, 732
  isotonic, 732
  order k, 732
Indeterminate region, 735
Index of refraction, 315, 321
Induced function
  in interval analysis, 729
Inf, 816
Infeasable region, 735
Infinite precision arithmetic, 213
Inflection point, 456, 641
Inner product
  on $L^2([a,b])$, 775
Input node
  of machine, 218
Input-output map, 219
Instancing, 169
Integral curve, 36
  unit quaternion, 650
Interior
  of set in picture, 25
Interior angle
  at vertex of polygon, 716
Intermediate surface
  for texture mapping, 326, 327
Interpolating plane, 487
Interpolating spline
  buckling, 459
Interpolating surface
  four point, 486
  plane, 487
Interpolation
  bicubic, B-spline, 516
  cubic Bézier, 444
  cubic B-spline, 441
  higher-order B-spline, 445
  Lagrange, 378
  piecewise Hermite, 383, 384
  spline, 388, 389
Interpolatory constraints, 376
Interpolatory refinement scheme, 461
Intersection
  as zeros of function, 538
  block-polyhedron, 578
  comparison of methods, 579
  curve-curve, 545
  curve-surface, 552
  facet-plane, 554
  finding significant points for, 570
  line-cone, 578
  line-cylinder, 578
  line-plane, 241
  of convex hulls, 540
  of CSG objects, 579
  of intervals, 710
  of parametric cubics, 578
  of parallelopipeds, 562
  of rectangles, 710
  of segments, 240
  plane-surface (section/contour), 554
  projection approach to, 577
  ray-bounding box, 348
  ray-circle, 244
  ray-cone, 346
  ray-curve, 543, 544
  ray-cylinder, 345, 578
  ray-facet, 346
  ray-faceted surface, 348
  ray-polygon, 346
  ray-quadric surface, 578
  ray-quadrilateral, 347
  ray-slab, 348
  ray-sphere, 345
  ray-surface, 552
  representation of, 539
  substitution approach to, 574
  surface-surface, 553, 557
Intersection curves
  jumping components, 580
  misordered components, 580
  properties of, 580
Interval
  absolute value of, 729, 730
  of $\mathbf{R}^n$, 730
  midpoint of, 730
  width of, 730
Interval analysis, 177, 213, 545, 579, 615, 624, 726
  disadvantages of, 744
  Newton-Raphson method for, 744
Interval tree, 704
Invariant
  with respect to contractions, 813
Inverse Fourier transform
  discrete, 786
  of function of one variable, 781
  of function of two variables, 785
Inverse kinematics, 142

Inward layout, 666
Inward pointing normal, 239
Inward pointing normal test, 239
Isolated point
  of set in picture, 25
Isoline, 624
Isophote, 526
Isosurface, 362, 624
Isosurface generation algorithm, 593
Isovalue, 624
Iterated function system, 190, 810
  attractor for, 810
  code space for, 812
  contractivity factor of, 810
  deterministic, 807, 810
  just-touching, 812
  nondeterministic, 808, 810
  overlapping, 813
  totally disconnected, 812
  with condensation, 811
  with probabilities, 810
IVR, 686

J
Jittering, 344
Joint, 142
  prismatic, 142
  revolute, 142
Joint angle, 142, 143
Joint offset, 142, 143
Julia set, 220, 801
Junction
  in medial axis, 184

K
K (kilobyte, = 2^{10}), 9
k-adjacent, 24
Kd-tree, 702
k-fold composite, 798
k-fold iterate, 798
Kinematics, 142
  forward, 142
  inverse, 142
k-neighbor, 24
Knot, 387, 608
  multiplicity of, 387
Knot insertion, 428, 593
Knot spacing
  centripetal, 444
  chord-length/chordal, 444
  comparison of, 444
  uniform, 444
Knot vector, 387, 408, 411, 504
  clamped, 408
  length of, 387
  nonuniform, 408
  open uniform, 408
  periodic, 408
  standard clamped uniform, 409
  standard uniform, 409
  unclamped, 408
  uniform, 408
Koch curve, 188
  triadic, 805, 813
Kronecker delta, 815

L
L^∞
  distance, 775
  function, 775
  norm, 775
  space, 775
Lagrange
  basis functions, 379, 490
  interpolation, 378
  polynomial, 378, 379
Lambert's law, 311
Laplace equation, 768, 781
  for electric potential problem, 770
  for fluid flow, 770
  for steady temperature problem, 769
  in blending and fairing, 682
Lattice evaluation method
  for sections, 554
LCD, 7
Least squares approximation, 376
Least squares method, 619
  in curve fitting, 461, 533
Leaving list
  in Weiler polygon clipping, 85
Lebesgue integral, 767, 773
Left turn
  of vectors, 238
Length
  of computation, 219
  of discrete curve, 24
  of knot vector, 387
  of pwl curve, 657
Lens
  of eye, 295

Level curve, 624, 630
Level set, 801
  as shape descriptor, 188
Liang-Barsky line clipping, 70, 77, 80, 125, 126
  entry value, 78
  exit value, 78
Liang-Barsky polygon clipping, 70, 71, 86, 87, 98
Light, 297
  ambient, 310
  diffuse, 310
  monochromatic, 297
  specular, 310
Light intensity
  clipping, 334
  scaling, 334
Light map
  view-dependent, 309
  view-independent, 309, 323
Lightness
  perceived, 296
Line clipping, 69
  Cohen-Sutherland, 70, 71, 81, 89, 93, 97, 125
  Cyrus-Beck, 70, 73, 77, 125
  Liang-Barsky, 70, 77, 81, 125
  Nicholl-Lee-Nicholl, 70, 81
Line drawing algorithms
  Bresenham, 39, 41
  DDA approach to, 37
  midpoint, 40
  n-step, 40
  run-based, 43
Line of curvature, 520, 526, 651
Linearly separable sets, 711
  strictly, 712
Link, 142
Link curve, 598, 674
Link frame, 143
Link length, 143
Link parameters, 143
Link twist, 143
Lipschitz condition, 732, 735
Liquid crystal display, 7
List priority algorithm, 264, 265, 268, 269, 330, 332
Local control
  for B-spline curve, 411
  for B-spline surface, 504
  for Doo-Sabin surfaces, 528
  for NURBS curve, 436
  for NURBS surface, 514
Local convex hull property
  for B-splines, 417
  for NURBS curve, 436
  for NURBS surface, 514
Local deformation, 222
Local scaling, 135
Local shape function, 747
Lofted surface, 483, 487
Lofting, 387, 472, 473, 630
  conic, 533
Logical screen, 5
Lookup table
  for colors, 64
Loop, 456, 627, 628
Loop detection, 570, 579
Loop subdivision algorithm, 530
Lower halfplane, 815
Lower hemisphere, 816
L^p
  distance, 774
  function, 773
  metric, 774
  norm, 753, 773
  pseudometric, 774
  space, 773
LU-decomposition, 516
Luminous object, 295
Luminance, 301, 334, 335

M
M (megabyte, = 2^{20}), 9
MA, 183
Machine
  branch node of, 218
  computation node of, 218
  input node of, 218
  input space of, 218
  next node of, 218
  output node of, 218
  output space of, 218
  over ring, 218
  state space of, 218
Machine representation, 163, 192
Magnetic resonance imaging, 181
Maillot polygon clipping, 70, 71, 86, 87, 88, 89
Mandelbrot set, 801

Manipulator, 142
  degrees of freedom, 142
Manifold, 159, 166, 169, 188, 453, 454, 624, 633, 658, 689, 690, 692, 693, 739, 803
Marching cube algorithm, 362, 365, 594, 625
Marching method, 558, 561, 566, 567, 579
  problems with, 567
  step constraint approach in, 566
  step direction and size for, 568
Marching square algorithm, 369
MAT, 185
Material features, 194
Matrix
  in filament winding, 667
Max metric, 26
Maximal disk
  in set, 183
Maximum principle
  for harmonic functions, 768
Maxwell triangle chromaticity diagram, 297
Mean curvature, 525, 591, 644, 645, 650, 651
Mean value form, 735
Mean value theorem
  in interval analysis, 734
Mechanical spline, 445
Medial axis, 183
  arc, 184
  body sheet, 184
  junction, 184
  seam, 184
  sheet, 184
  wing sheet, 184
Medial axis representation, 185
  for CSG objects, 186
Medial axis transform, 185
Medial surface, 183
Median cut palette quantization, 333
Median cut scheme
  in ray tracing, 342
Medical imaging, 181
Meridian
  of surface, 473
  of surface of revolution, 475, 656
Mesh
  Bezier, 645
  Hermite, 645
Method of weighted residuals, 749
Metric
  Euclidean, 26
  Hansdorff, 809
  max, 26
  taxicab, 26
Meusnier theorem, 650
Microsoft Windows, 11, 14, 121
Midedge subdivision algorithm, 529
Midpoint
  of interval, 730
Midpoint line-drawing algorithm, 40, 362, 363, 365
Milling machine, 174, 456, 642, 643
  numerically controlled, 582
Minimax test
  for box intersections, 229
Mip-mapping, 327
Mirror direction, 310, 312, 313
Mirror operation, 223
Mode, 9
Model simplification, 592
Monohedral triangulations
  generated by reflections, 627
Monotone polygon, 716
Moore-Penrose inverse, 539
Morse function, 624
MRI, 181, 593, 594
Multiaffine map, 418
Multigrid method, 753
Multiplicative structure, 755, 756
Multiresolution analysis, 795
Multiresolution modeling, 598

N

Natural end condition, 388
Natural path
  for tape, 669
Natural spline, 388
Navier-Stokes equations, 182
NC cutter, 557
NC machine, 162, 582, 646
  two-axis, 639, 642
NC programming, 194
NDC, 5, 121, 130
Near clipping plane, 5, 138
Near plane, 229
Nef polyhedron, 225
Neighbors
  18-, 23
  26-, 23

4-, 23
6-, 23
8-, 23
k-, 24
Nerve cells
in eye, 295
Newell-Newell-Sancha visible surface algorithm, 265, 266, 269, 292
Newton-Raphson method, 286, 287, 544, 552, 553, 558, 559, 560, 562, 563, 566, 567, 568, 569, 570, 572, 579, 583, 594, 615, 617, 624, 637, 638, 672, 744
interval analysis version of, 744
overview of, 539
Nicholl-Lee-Nicholl line clipping, 70, 81
Node
in FEM, 746
Nondegenerate
planar cubic curve, 457
Nondestructive testing, 182
Nonluminous object, 295
Nonuniform rational B-spline surface, 513
Normal
induced, 239, 240
inward-pointing, 239
outward-pointing, 239
Normal curvature, 525, 651
Normal vector field, 462
relatively parallel, 462
Normalized device coordinates, 5, 121
Normals test, 232, 238
NTSC, 301
Numerically controlled machine, 162
Numerical stability, 467, 531, 539, 578
NURBS
history of term, 534
NURBS curve, 378, 433, 434, 435, 531, 599, 600
advantages of, 467
derivatives of, 438
disadvantages of, 467
evaluation of, 438
local control of, 436
local convex hull property, 436
order of, 433
projective invariance of, 436
properties of, 436
variation diminishing property, 436
NURBS surface, 221, 474, 513, 531, 599, 600, 645
evaluation of, 515
local control of, 514
local convex hull property, 514
partial derivatives of, 515
projective invariance of, 514
properties of, 513
Nyquist frequency, 790
Nyquist limit, 45

O

Object of revolution, 474
Object precision algorithm, 264
Object reconstruction, 139, 140
Oblique parallel projection, 133
Oblique view, 133
Octonion, 756
Octree, 185, 205, 362, 594
Octree visible surface algorithm, 266, 283
Offset, 638
geodesic, 638
Offset curve, 177, 638, 646
approximation of, 643
of space curve, 643
planar, 640, 641
rational, 644
Offset surface, 177, 186, 520, 638, 644, 646, 681
approximation of, 645
One-point perspective, 118
One-sided power function, 404
Op mode, 9
Opacity, 360
OpenGL, v, 11, 14, 15, 64, 138, 293, 340
Optic nerve, 295
Orbit, 798
Ordered edge list fill algorithm, 51
Ordering
induced, 240
Orientation
induced, 240
induced by normal, 239
of polygon, 239
Oriented bounding box, 573
Oriented polygon, 239
Origin
of view plane, 5

Origin registers, 64
Orthogonal projection, 133, 187
Orthogonal projection assumption, 266
Orthogonality constraints, 377
Orthographic projection, 133
Orthographic view, 133
Oslo algorithm, 429
Output node
  of machine, 218
Outward pointing normal, 239
Outward pointing normal test, 239
Overcut
  of NC cutter, 639

P
PADL-1, 157
Painter's algorithm, 63, 268, 320
Palette, 332
  choosing one, 332
Palette quantization
  fixed, 333
  median cut, 333
  octree, 333
  popularity, 333
  uniform, 333
Pan feature
  of hardware, 64
Parabolic cyclide, 519
Parabolic cylinder, 514
Parabolic point, 458
Paraboloid
  elliptic, 480, 514
  hyperbolic, 484, 514
  of revolution, 480, 655
Parallel curve, 639
Parallel graph grammar, 190
Parallel projection, 132
Parallel surface, 644
Parallel transport frames, 461, 463
Parametric blending
  polyhedral, 681
  rolling ball, 681
  spine-based, 681
  trimline-based, 681
  with n-sided patches, 681
Parametric curve, 373
Parametric model, 192
  versus variational model, 193
Parametric representation, 178, 221
  versus implicit definition, 259
Parameterization, 373
Parameterized surfaces
  implicitization of, 574
Parity check
  in fill algorithms, 28, 51, 52
Parity test, 102, 234, 238, 319, 610
Particle system, 190
Partition of unity, 387
Pattern analysis, 188
Patterning, 48
PC, 8, 9, 13, 59, 66, 337
Penumbra, 320
Perceived color
  characteristics of, 296
Periodic B-spline matrix, 414
Periodic end condition, 388
Periodic point, 220, 798
  attractive, 220, 800
  period of, 798
Perspective view, 113, 118
  one-point, 118
  two-point, 118
  three-point, 118
PHIGS, 11
Phong shading, 317, 331, 332, 369
Phong specular reflectance model, 313, 314, 333
  simplified, 313
Photorealism, 308, 310, 361
Physically based algorithm, 603
Physically based modeling, 191
Picture, 25
Piecewise linear curve, 657; *see* pwl curve
Piecewise polynomial, 387
Pinhole camera, 113
Pitteway-Watkinson algorithm, 46
Pixel, 8
  choosing coordinates of, 49
Pixel ray, 339
Place
  of curve, 615, 616
Planar unfolding, 661
PMC, 206
Point sampling, 265
Point set topology, 167
Poisson distribution
  minimum-distance, 344
Poisson kernel, 778

Polar form, 419, 420
of quaternion, 760
tensor product, 505
triangular, 506
Polar form theorem, 419
Pole
of polynomial, 422
Polygon, 714
as B-spline, 405
diagonal of, 715
end vertex of, 717
interior angle at vertex, 716
merge vertex of, 717
monotone with respect to line, 716
regular vertex of, 717
shadow, 318
simple, 715
split vertex of, 717
start vertex of, 717
strictly y-monotone, 718
triangulation of, 714, 718, 719
turn vertex of, 716
y-monotone, 716
Polygon clipping, 69
Greiner-Hormann, 70, 71, 98, 106
Liang-Barsky, 70, 71, 86, 87, 98
Maillot, 70, 71, 86, 87, 88, 89
Sutherland-Hodgman, 70, 71, 84, 86, 87, 98, 105
turning point based, 71
Vatti, 70, 71, 98, 106, 108
Weiler, 70, 71, 85, 98, 106
Polygonization, algorithms
see also tiling algorithms
adaptive, 589, 591
by refinement, 598
comparison of, 593
for Bézier and B-spline objects, 593
for Bézier triangles, 591
for curves, 588, 593
for surfaces, 591, 593
properties of, 588
step size in, 593
Polygonize, 588
Polyhedral blending, 673
Popularity palette quantization, 333
Portable code, 10
Postfiltering, 46, 47
Potential method
in blending, 676
Preferred polarity approach
in surface tiling, 595
Prefiltering, 46
Primary color, 298
imaginary, 298
real, 298
transmission, 301
Primary colors
additive, 297
subtractive, 299
Primitive function, 375
Primitive instancing, 165
Principal curvatures, 591, 650, 651
Principal normal curvatures, 525, 645
Principal patches, 651
Prismatic joint, 142
Procedural model, 190
Procedural modeling, 215
Production Automation Project
University of Rochester, 157
Profile curve
for blending surface, 680, 681
Progressive refinement, 353
Progressive transmission, 597
Projecting cone
of space curve, 579
Projecting curves
to surface, 671
Projection
axonometric, 133
oblique parallel, 133
of one set on another, 664
of point along vector, 712
of set along vector, 712
orthogonal, 133
orthographic, 133
parallel, 132
Projection approach
for surface intersections, 577
Projection operator, 490
Projections
taxonomy of, 134
Projective invariance
of NURBS curves, 436
of NURBS surfaces, 514
Proximate interval, 740
Pupil, 295
Pwl curve, 657
angle at vertex, 662
closed, 657

connecting, 661
elementary subdivision of, 658
induced by interval, 658
length of, 657
part of path of, 659
path of, 657
simple, 657
standard parameterization of, 658
subdivision of, 658
underlying space of, 657

Q
Quadratic form, 579
Quadric
as projection of paraboloid, 514
as rational surface, 514
Quadric surface, 480, 514, 622, 678
Quadtree, 185, 204, 561, 565, 566, 592, 594, 604
Quantitative invisibility of point, 266
Quasi-disjoint, 159
Quaternion
absolute value of, 758
as rotation, 763
conjugate of, 758
exponential notation of, 761
norm of, 758
polar form of, 760
pure, 757
pure part of, 757
real part of, 757
unit, 759
Quaternion algebra, 757
Quaternion product, 756
Quaternions
for curve frames, 465
in sweeping, 175
vector product of, 759

R
Radial transformation, 809
Radiosity, 318, 320, 323, 351
antialiasing in, 357
Radiosity equation, 351
Radiosity method, 350
Radius function
for medial axis, 183
Range parameter
for blend, 679
Range tree, 700, 703
Raster, 8
Raster scan CRT, 7
Rational basis functions, 433, 512
Rational Bézier curve, 432
control point of, 433
weight of, 433
Rational Bézier surface, 512
control point of, 512
weight of, 512
Rational B-spline curve, 432
control point of, 433
rational basis functions for, 512
weight of, 433
Rational B-spline surface, 512
control point of, 512
rational basis functions for, 512
u-knot of, 512
v-knot of, 512
weight of, 512
Rational tensor product surface, 512
Ray casting, 358, 369
Ray theory, 310
Ray tracing, 265, 266, 309, 318, 319, 322
adaptive supersampling, 343
beam tracing, 343
cone tracing, 343
distributed ray tracing, 344
minimum-distance Poisson distribution, 344
jittering, 344
stochastic sampling, 344
supersampling, 343
Ray tracing program, 338
Realism in graphics, 337
Realization
of CSG-tree, 168
Receptor cells
in eye, 295
Reconstruction
of camera data, 139, 141
of objects, 139, 140
Rectangle
how specified on screen, 12
Recursive subdivision curve, 465
Recursive subdivision surface, 526, 530, 681
affinely invariant, 528
Recursively enumerable set, 219
Reeb graph, 188
Ref, 821
Refinement, 597
Reflectance, 310

Reflectance factor
  diffuse, 311
Reflectance model, 308
  distance factor in, 315
Reflection coefficient
  ambient, 311
  diffuse, 311
  specular, 312
Reflection coefficients
  suggestions for, 334
Reflection line, 525, 526
Reflection mapping, 328
Reflection ray, 339
Reflection rule
  in Coxeter triangulation, 627
Refleshing object
  from medial axis, 186
Refresh buffer, 8
Region
  feasable, 735
  indeterminate, 735
  infeasible, 735
  subtended by curve from point, 251
Regular set, 158
Regular vertex
  of polygon, 717
Regularization
  of set, 158
Regularization operator, 158
Regularized set operator, 160
Relatively parallel
  adapted frame field, 463
  normal vector field, 462
  tangential vector field, 463
  vector field, 463
Relaxation method, 352, 603, 604
Relaxing points, 539
  in Barnhill-Kersey algorithm, 562
  to surface,
Removing singularity
  with quadratic transformation, 617
Render, 10
Rendering
  by-polygon, 330
  by-scan-line, 330
  smooth surfaces, 330
Rendering equation, 323
Rendering pipeline, 330
Reparameterization
  of curves, 396, 453
  of surfaces, 522
Representation, 161
  medial axis, 185
  symantically correct, 161
  syntactically correct, 161
  valid, 161
Representation scheme, 157, 161, 223
  complete, 161
  domain of, 161, 162
  informal properties of, 163
  object, 163
  unambiguous, 161
  unique, 161
  validity problem of, 162
Residual, 352, 749
Resolution, 8
Resultant, 575, 577, 578
Retina, 295
Revolute joint, 142
RGB color model, 299
RGB color values, 64
Riemann integral, 767, 773
Right turn
  of vectors, 238
Ritz method, 746
R-move, 55
Robot, 639, 646, 670
Robot path planning, 188, 639
Robustness
  issues with respect to, 213, 726, 738, 798
Rod receptor cell
  in eye, 295
Romberg integration, 637
Rotate operation, 223
Rotation
  via quaternion, 763
Rotational sweep, 174
Rotations
  composition of, 764
Rounding, 187, 222, 672
R-set, 157, 158
Rubber banding, 62
Ruled surface, 482, 483
  directrix-generator representation of, 473
Run-length encoding, 360

S

Sampling problem, 44, 767, 788
Sampling frequency, 45
Saturation
  perceived, 296
Scaling function, 793

Scaling operation, 223
Scan conversion, 10, 49, 50
Scan line, 7, 8
Scan line algorithm, 276, 289, 290, 291, 292, 309, 317, 318, 330, 598, 622
Scene analysis, 598
Schumacker visible surface algorithm, 265, 266, 268, 270, 292
Screw sweep surface, 485
Sculptured surface, 220, 472
Seam
  in medial axis, 184
Section
  of set, 553
  of ship, 553
Section algorithm, 554
Sederberg-Nishita Bézier clipping, 551
Seed fill algorithm, 28
Segment, 228, 276
  of B-spline curve,
Segment tree, 707
Segmentation
  in volume rendering, 361
Segre characteristic, 579
Self-relation
  of cells, 200
Self-similar, 813
Semialgebraic set, 159, 579
Semianalytic set, 159
Sensitive dependence
  on initial conditions, 798
Separable function, 785
Separating hyperplane, 711
  strictly, 712
Separating plane, 208, 270
Separation of variable method, 772
Set constraint function, 735
  solution acceptance, 735
SetColor procedure, 15
SetMode procedure, 15
Shading
  constant, 316
  Gouraud, 316
  Phong, 317
Shading model, 308
Shadow
  of point, 664
  of point on edge, 664
Shadow algorithm
  hard, 320
  soft, 320
Shadow algorithms, 318
Shadow polygon, 318
Shadow ray, 339
Shadow volume, 318
Shadow z-buffer, 318, 319
Shadowed face, 664
Shape, 112
Shape analysis, 393
Shape constraints, 377
Shape coordinate system, 112
Shape function
  global, 749, 751
  local, 751, 752, 753
Shape optimization, 188
Shape recognition, 188
Shape parameter, 454
Shape preserving surface, 651
Shear, 136
Sheet
  in medial axis, 184
Shellable n-cell, 174
Shelling
  closed, 645
  open, 645
Shininess, 313
Shooting
  of light, 354
Shortest edge sequence, 663
Shrink wrapping, 327
Sign function, 817
Signal processing, 44, 767, 788
  definition-to-display, 792
Significant point
  of intersection curve, 570
Silhouette, 285
Simple
  edge sequence, 660
  edge-adjacent sequence, 660
Simplicial complex, 167, 224, 723, 724
Simplification problem, 597
  decimation, 597
  refinement, 598
Simply connected set in picture, 25
Sinc function, 783, 785
Singularity
  of curve, 456, 615
  removal of, 617
Site, 720
Size criterion
  for surface triangulation, 600
Skeletal line, 651

Skeleton, 183
  exterior, 186
  interior, 186
Skinning, 223, 361, 557, 630, 633
  branching problem for, 631
  correpondence problem for, 631
  for B-spline curves, 632
  surface-fitting problem for, 631
  tiling problem for, 631
Skinning curves, 630
Skinning surface, 517, 630, 633, 651
  properties of, 630
Slab, 229, 348, 573, 708
  induced from unit vector, 229
Smith seed fill algorithm, 29, 30
Smoothing, 672
Smoothness conditions, 377
Snell's law, 320
SO(2), 755, 817
SO(3), 764, 817
Soft area flooding, 32
Solid, 158, 159, 176, 178
  tricubic parametric, 178, 224
Solid modeling, 157
Solid sweep, 174
Sorting phase
  in surface intersection computations, 558, 561, 566
Span, 276, 387
  sample, 278
Sparse polynomial, 572
Spatial coherence, 342
Spatial domain, 784
Spatial occupancy enumeration, 179, 180, 183
Sphere
  n-dimensional, 816
  parameterization of, 478
Spine curve, 679, 681
Splatting, 359
Spline, 387
  see also B-spline curve
  cubic, 387
  geometric, 468
  in tension, 468
  knot of, 387
  linear, 387, 445
  mechanical, 445
  natural, 388
  nonlinear, 445
  physical, 387, 445
  quadratic, 387
  shape preserving, 468
  wooden, 445
Spline curve,
  interpolating, 390
Spline interpolation problem, 388
Splitting node, 699
Sprites, 66
Square integrable functions, 774
  inner product of, 775
Standard trace direction, 619
STEP, 198, 211, 467
Stereo views, 131
Stereolithography, 601, 603
Stochastic sampling, 344
Subdistributive, 728
Subdivision
  adaptive, 177, 178, 573, 589
  cyclic, 737
  of cubic curve, 449, 451
  of Bèzier curve, 451
  of B-spline curve, 452
  of pwl curve, 658
  of triangle, 592
  proper, 658
  recursive, 465, 526
  triangular surface, 521
  uniform, 177, 178
Subdivision algorithm
  Catmull-Clark, 529
  Chaikin, 465
  corner cutting, 465, 526, 527
  cracks in, 599, 600, 601, 603, 604, 608
  Doo-Sabin, 527
  Loop, 530
  midedge, 529
  trapezoidal, 606
  vertex insertion, 526, 530
Subdivision problem, 449, 521
Subject polygon, 69
Subtractive primary colors, 299
Sup, 816
Super VGA, 8, 9
Supercyclide, 521, 678
Superellipse, 448, 482, 676
  corner point, 448
  superness, 448
Superficial blending, 672

Superness
of superellipse, 448
Superposition, 472
Superquadric surface, 482
Supersampling, 47, 291, 343
scaling factor, 47
Support, 406
compact, 387
Surface, 472
bilinear, 486
developable, 483
doubly curved, 484
doubly ruled, 483
dull, 313
four-point interpolating, 486
free-form, 472
glossy, 313
implicit, 177, 481
interrogation of, 526
lofted, 483
NURBS, 513
offset, 177, 186
rational Bézier, 512
rational B-spline, 512
rational tensor product, 512
recursive subdivision, 526
sculptured, 472
singly curved, 484
superquadric, 482
sweep, 484, 485
trimmed, 177
Surface blending, 672
Surface-fitting problem, 625
for skinning, 631
Surface of revolution, 474, 475, 667, 668
circle of latitude of, 475
directrix-generator representation of, 473
full, 475
meridian of, 475
problems for, 475, 476
standard parameterization of, 475
Surface patch, 472
Surface reconstruction, 633
Surface rendering, 358
Surface simplification, 597
Surface tiler
continuation approach, 594
enumeration approach, 594
subdivision approach, 594
Surface-surface intersection, 557
comparison of methods, 579
hunting phase for, 558
sorting phase for, 558
tangential, 579
tracing phase for, 558
via algebraic methods, 574
via continuation method, 570
via divide-and-conquer, 572
via embedding method, 570
via homotopy method, 570
via lattice evaluation, 558
via marching method, 558
via recursive subdivision, 572
via Timmer algorithm, 558
via tracing method, 558
Surrounding test
evaluation of, 238
for quadrilateral, 347
using angle counting, 235
using barycentric coordinates, 233
using crossings, 234
using equations, 233
using normals, 232
using parity, 234
using wedges, 233
Sutherland-Hodgman polygon clipping, 70, 71, 84, 86, 87, 98, 105
Sweep operation, 174, 222, 678, 679, 681
Sweep representation, 175, 177
Sweep surface, 484, 485
screw, 485
Sweep vector, 483
Symmetric curve, 399
Symmetric axis, 183
Symmetric difference, 816
Synthetic camera, 113

T

Tangent plane continuity, 522
Tangential vector field, 462
relatively parallel, 463
Tape laying, 667, 668
Tape path, 669
Taxicab metric, 26
Templated discrete ray
in volume rendering, 358
Tension, 454
Tensor product blossom, 505

Tensor product polar form, 505
Tensor product surface, 472, 495, 505
  Bézier, 501
  bicubic, 496
  B-spline, 504
  rational, 512
  subdivision of, 521
Terminating point, 564, 565
Tesselate, 588
Text clipping, 110
Texture
  aliasing problems for, 326, 327
  distortion problems for, 326
  physical, 324
  shrink wrapping approach to, 327
  statistical measures for, 324
  structural measures for, 325
  visual, 324
Texture coordinates, 325
Texture mapping, 325
  2-dimensional, 325
  3-dimensional, 327
  intermediate surface for, 326, 327
  using chordlength, 326
Three-point perspective, 119
Thresholding, 48
Thumbweight
  of profile curve, 680
Tile, 588
Tiling algorithm; *see also* polygonization algorithms
  adaptive, 594, 596
  approaches to, 594
  continuation approach to, 594
  continuous, 593
  discrete, 593
  enumeration approach to, 594
  implicit, 593
  subdivision approach to, 594
Tiling problem
  for skinning, 631
Timmer algorithm, 558, 559, 563, 569, 615
  hunting phase for, 559
  ordering phase for, 561
  tracing phase for, 559
Tint fill, 32
Tolerance features, 194
Tolerancing, 223
Tomography, 180
Tool frame, 142
Topologically adequate
  data structure, 200
Topologically transitive, 798
Topology inference approach
  in surface tiling, 595
Toroidal graph, 631
Torrance-Sparrow specular reflectance model, 314
Torus
  parameterization of, 478
Total angle
  for surrounding test, 236
Total internal reflection, 322
  critical angle of, 322
Total variation, 779
Trace direction, 619, 620, 621
  standard, 619, 620
Tracing method, 558
Tracing phase
  in surface intersection computations, 558, 559, 563
Tracing seams and sheets, 185
Trajectory generation, 142
Transfer function
  in volume rendering, 360
Transfinite interpolant, 488
Transforming
  implicit representation, 260
  parametric representation, 260
Translational sweep, 174
Transmission primary, 301
Transparency, 310, 320
Transparency ray, 339
Trapezoidation problem
  for planar regions, 605
Trapezoidation algorithm, 606
Tree structure encoding,
Triangular Bézier surface, 508, 509
  evaluation, 508
  partial derivatives of, 511
Triangular blossom, 506
Triangular control net, 508
Triangular domain
  advantage, 531
  disadvantage, 532
Triangular polar form, 506
Triangulating polygons, 714, 718, 719

Triangulation, 600, 601, 602, 603, 604, 626, 627
  angle-optimal, 724
  angle vector for, 724
  Coxeter, 627
  deviation criterion, 600
  local, 627
  monohedral, 627
  of planar point set, 724
  size criterion, 600
Triangulation algorithm
  for y-monotone polygons, 718, 719
  via locally isometric approximations, 605
Triangulation problem
  for planar regions, 605
Tricubic parametric solid, 178
Trim curve, 674
Trimmed surface, 596, 598
Trimline
  for blending surface, 679, 681
Trimming, 638, 642
Trimming algorithm, 599, 600, 601, 603, 604, 606, 611
Trimming curve, 598, 599, 600, 614, 674
Tripod 6-connected line algorithm, 365
Tri-stimulus theory of light, 297
Trunc function, 817
Truncated cone, 476
Truncated power function, 404
Tunnel
  in discrete 3d set, 362
Turning point, 71, 86, 87, 641
  of intersection curve, 570
Twist vectors, 496, 499, 533
  Adini, 499
Two-point perspective, 118, 136,
Typeface, 109

U
Umbra, 320
Undecidable set, 219
Undercut
  of NC cutter, 639
Undersampled, 45
Undoing
  an editing operation, 222
Unfolding map, 661
Unfolding of set, 661
Uniform B-spline matrix, 414
Uniform subdivision, 178
Unit cube, 815
Unit disk
  closed, 816
  open, 816
Unit quaternion integral curve, 650
Unit sphere, 816
Unstable system, 798
Up direction, 113
Upper halfplane, 815
Upper hemisphere, 816
User representation, 163, 192
  informal issues, 164
Uv-monotone region, 599, 600

V
Valid representation, 161
Validity problem
  for representation schemes, 162
  of boundary representation, 166
  of CSG representation, 169
  of Euler operation representation, 171, 173
  of feature models, 195, 196
Vanishing point
  of line for view, 118
Variation diminishing property, 430, 502, 505, 514, 544, 547
Variational constraints, 377
Variational method
  for FEM, 748
Variational model, 192
  versus parametric model, 193
Vatti polygon clpping, 70, 71, 98, 106, 108, 606
VE, 685
Vector
  transforming, 259
Vector field, 36, 177, 756
  along curve, 462
  normal to curve, 462
  relatively parallel, 463
  tangential to curve, 462
Vector product
  of quaternions, 759
Vernier acuity, 45
Vertex, 641
  of convex set, 714

Vertex insertion
  subdivision algorithm, 526, 530
Vertex ordering
  induced, 240
Vertical retrace, 7, 63
VGA, 8, 9
View
  axonometric, 133
  oblique, 133
  orthographic, 133
  perspective, 113
View direction, 5, 113
View plane, 4
View plane coordinate system, 4, 117
View pyramid, 5
  truncated, 5
View volume, 5
  truncated, 5
Viewpoint, 4
Viewport, 5, 119
Viewport-to-pixel-space transformation, 121
Virtual environments, 685
  interaction in, 686
Virtual reality, 685
  immersive, 686
  passive, 686
Visible human project, 182
Visible line determination, 266
Visible radiant energy, 297
Visible surface algorithm
  Blinn, 285
  BSP, 270
  Carpenter-Lane, 290
  Catmull, 284
  Clark, 290
  curved surface 284
  list priority, 268, 269, 283
  Newell-Newell-Sancha, 269
  Octree, 283
  Schumacker, 268
  Warnock, 273
  Watkins, 278
  Weiler-Atherton, 274
  Whitted, 290
  Z-buffer, 275
Visible surface algorithms
  comparison, 292
  image precision, 264
  list priority, 264
  object precision, 264
Visible surface determination, 264
  overview of algorithms, 264
Vitreous humor, 295
Volume
  of parallelopiped, 249
  shadow, 318
  of tetrahedron, 251
Volume data structures, 203
Volume graphics, 180
Volume modeling, 180
Volume rendering, 180, 358, 625, 685
  Artzy's algorithm, 362
  data classification for, 358, 360
  image precision, 358
  marching cube algorithm, 362, 365
  object precision, 359
  shear warp, 360
Volume thinning, 185
Volume visualization, 180
Volumetric blending, 673
Volumetric data, 180
Voronoi cell, 720, 723, 724
  edge of, 721
  vertex of, 721
Voronoi diagram, 185, 232, 603, 666, 667, 720, 721, 722, 723, 725
  Fortune's algorithm for, 722
Voronoi graph, 722
Vorticity
  of fluid, 182
VoxBlt, 181
Voxel, 179
Voxel block transfer, 181
VR, 658

W

Warnock visible surface algorithm, 265, 266, 273, 291, 292, 614
Waterline
  of ship, 533
Watkins visible surface algorithm, 265, 266, 278, 292
Wavelet, 792
  approximation, 795
  compression, 795
Wavelet transform
  discrete, 795, 796
Wedge test, 233, 238
Weierstrass approximation theorem, 398

Weight
of rational Bézier curve, 433
of rational Bézier surface, 512
of rational B-spline curve, 433
of rational B-spline surface, 512
Weights
geometric interpretation of, 435
Weiler polygon clipping algorithm, 70, 71, 85, 98, 106
entering list, 85
leaving list, 85
Weiler-Atherton shadow algorithm, 318
Weiler-Atherton visible surface algorithm, 266, 274
Whittaker-Shannon sampling theorem, 44, 790
Whitted visible surface algorithm, 290
Width
of interval, 730
Winding number, 106, 235
Window, 5, 6, 119, 273
Window-to-device pipeline, 6
Window-to-viewport transformation, 12
WinG, 11
Wing sheet
in medial axis, 184
Winged edge data structure, 202
Winged edge representation, 173, 201, 203
Wireframe display, 165
Wireframe representation, 164
Wooden spline, 445
Workspace, 142
World coordinate system, 4, 6, 112
World-to-camera transformation, 115, 622
Write procedure, 16

X
XOR mode, 9
moving objects with, 9
X-ray, 181

Y
YIQ color model, 300
Yon clipping plane, 5

Z
Z-buffer, 9, 275
Shadow, 319
Z-buffer algorithm, 265, 266, 275, 291, 292, 319, 330, 598, 606
scan line, 276
Zoom feature
of hardware, 64

Bibliographic Index

[AbdY96], 571, 850
[AbdY97], 568, 579, 850
[AbeD81], 18, 19, 854
[AbhB87a], 835
[AbhB87b], 835
[AbhB87c], 835
[AbWW85], 291, 836
[ACDL00], 633, 862
[AckW81], 53, 857
[AgHK00], 667, 843
[AgoG87], 294, 295, 301, 837
[AgoM05], v, vi, vii, viii, 22, 111, 141, 156, 171, 149, 179, 188, 224, 235, 246, 247, 249, 259, 264, 367, 373, 376, 379, 386, 399, 403, 418, 431, 446, 456, 457, 464, 472, 480, 482, 484, 517, 537, 539, 563, 574, 575, 582, 615, 617, 622, 624, 625, 639, 641, 644, 649, 650, 651, 652, 668, 678, 684, 688, 714, 721, 740, 755, 756, 763, 765, 767, 768, 774, 797, 802, 852
[AlbD97], 521, 841
[AleH83], 726, 728, 852
[AllD97a], 678, 836
[AllD97b], 678, 836
[AllD97c], 678, 836
[AllG87], 595, 624, 849
[AllG90], 571, 595, 843
[AllG91], 595, 624, 849
[AllG93], 571, 845
[AllS85], 595, 624, 849
[AlSY97], 804, 842
[Aman84], 343, 858
[Amat96], 688, 845
[Ande93], 651, 840
[AnGC99], 604, 863
[Ange00], 843
[ANSI85], 11, 848
[ANSI88], 11, 848
[Apos58], 767, 779, 780, 784, 786, 788, 835
[Appe67], 266, 864
[ArcM75], 35, 857
[Arms94], 754, 842
[Arno83], 836
[Arvo91], 854, 857
[Aste88], 569, 850
[AtWG78], 318, 859
[Aure91], 720, 721, 724, 725, 838
[AzBB90], 538, 579, 850

[BaBB87], 388, 404, 407, 416, 468, 860
[BaDD87], 591, 592, 860
[Bado90], 233, 347, 854
[Baez02], 756, 856
[BagW95], 165, 845
[Banc95], 693, 865
[Baja92a], 835
[Baja92b], 835
[BajX92], 850
[BalB82], 843
[BaraW98], 855
[BarB83], 843, 845, 846, 847
[BarD89], 455, 860
[BarD90], 860
[BarD98], 11, 848
[BarF95], 645, 855
[BarG89], 860
[BarK90], 538, 558, 561, 562, 565, 566, 579, 850

[Barn87], 21, 842
[Barn88], 190, 804, 806, 807, 808, 809, 810, 811, 812, 813, 842
[Barn92], 843, 849, 850, 851, 852, 855, 861
[BarR74], 844, 860, 861
[Barr81], 482, 856
[Barr92], 482, 856
[BarrA87], 191, 855
[BarrA89], 856
[Bars88], 860
[Barze92], 856
[Baum72], 201, 845
[Baum75], 172, 201, 845
[BBGDS92], 799, 842, 852
[BBGS99], 184, 185, 186, 187
[BCGH92], 149, 856
[BDDH95], 11, 848
[BDST92], 725, 838
[BeaB82], 843, 859, 865
[Beac91], 389, 844
[BeBF78], 848
[BeFH86], 651, 862
[BeMR94], 719, 838
[Bézi71], 533, 845
[Bézi72], 459, 844
[Bézi74], 860
[BFJP87], 558, 566, 850
[BHLH88], 568, 569, 615, 850
[Bier95], 226, 845
[BieS86], 326, 863
[Binf71], 175, 845
[Bish75], 461, 462, 463, 464, 841
[BisW86], 317, 859
[BKOS97], 699, 701, 702, 704, 706, 708, 709, 715, 716, 718, 720, 721, 722, 724, 725, 838
[Blin00a], 459, 860
[Blin00b], 459, 860
[BliN76], 325, 326, 328, 863
[Blin77], 313, 314, 315, 859
[Blin78], 328, 863
[Blin81], 284, 864
[Blin82], 673, 845
[Blin87], 57, 839
[Blin88a], 59, 839
[Blin88b], 111, 848
[Blin89a], 459, 860
[Blin89b], 459, 860
[Blin91a], 111, 125, 848
[Blin91b], 111, 848
[Blin91c], 111, 848
[Blin92], 111,121, 848
[Blin93], 301, 837
[Blin99], 459, 860
[BloK02], 534, 861
[Bloo88], 624, 849
[Bloo90], 461, 841
[Bloo97], 187, 337, 594, 596, 622, 623, 849
[BloW89a], 682, 836
[BloW89b], 682, 836
[BlSS89], 217, 219, 220, 845
[Blum67], 183, 853
[Blum73], 183, 853
[BluN78], 853
[Boeh80], 429, 861
[Boeh87], 456, 845
[Boeh89], 841
[Boeh90], 517, 519, 521, 841
[Boen91], 538, 850
[BoeP94], 164, 844
[BoFK84], 845
[BoiC00], 633, 862
[BoiT93], 725, 838
[BoLZ75], 35, 857
[Boot79], 843, 864
[BotM58], 756, 836
[Bouk70], 309, 311, 859
[BowW83], 258, 852
[Bowy94], 841, 844, 846, 853
[BoyB00], 40, 858
[Brac86], 767, 783, 842
[Brad82], 856
[Bran92], 185, 853
[Brec92], 638, 646, 855
[Bres65], 39, 858
[Bres77], 39, 858
[BrHS80], 172, 845
[BroA99], 838, 844, 845, 852, 853, 854
[Brod80], 839, 844, 845
[Broo99], 685, 686, 864
[Brou84], 856
[Brun95], 863
[Buch95], 745, 842
[BuiT75], 309, 310, 313, 317, 859
[BurM71], 174, 863
[BurS93], 558, 850

[CaBU92], 863
[CalH68], 853
[Carl82], 547, 573, 850
[Carp84], 291, 864

[Cars98], 11, 848
[Casa87], 863
[CatC78], 529, 862
[Catm74], 326, 572, 864
[Catm75], 284, 290, 864
[Catm78], 276, 291, 864
[CatR74], 861
[CavM89], 465, 526, 845
[CCWG88], 351, 353, 355, 357, 857
[Chai74], 465, 862
[ChaK87], 557, 850
[Chan88], 59, 614, 849
[Chas78], 258, 854
[Chaz91], 718, 838
[ChBA94], 538, 850
[ChCM97], 853
[ChDH89], 517, 519, 841
[CheH90], 660, 666, 667, 843
[Chen89], 567, 850
[CheO88], 569, 850
[Chew93], 863
[Chia92], 853
[ChiK83], 681, 836
[Chin95], 273, 854
[Chiy87], 681, 836
[Chiy88], 173, 494, 502, 504, 681, 844
[ChoJ89], 681, 836
[ChPP98], 604, 614, 863
[CiMS98], 725, 838
[Clar79], 290, 573, 864
[ClaS89], 725, 838
[CMBZ00], 687, 864
[CMPP99], 863
[CohG85], 351, 857
[CohK97], 363, 365, 865
[CohW93], 351, 857
[CoLR80], 430, 861
[ConD72], 37, 390, 745, 854
[Cook86], 344, 858
[Cook89], 344, 858
[Coon67], 488, 533, 845
[CooT65], 786, 842
[CooT82], 859
[CoPL84], 344, 858
[CouH53], 856
[Crai88], 670, 841
[Crai89], 142, 146, 859
[Cran95], 799, 842
[Crip98], 841, 844
[Crow77a], 291, 836
[Crow77b], 318, 859
[CrSD93], 686, 864
[Cuil98], 592, 856
[CuKM99], 185, 213, 853
[CycW92], 578, 850
[CycW94], 578, 850
[CyrB78], 73, 837

[DahB74], 37, 854
[DanD89], 467, 861
[Debe99], 338, 859
[deBo78], 388, 404, 861
[DeFL87], 59, 858
[Dege94], 521, 841
[Dege98], 521, 841
[DehZ91], 591, 845
[DelE95], 836
[DenH55], 146, 859
[DeSB92], 215, 855
[Deva86], 799, 800, 842
[Devi98], 725, 838
[DevK89], 190, 842
[DLTW90], 626, 627, 628, 629, 839
[DocT81], 860
[DooS78], 527, 862
[DoSY89], 538, 579, 850
[Dowd85], 625, 839
[DrCH88], 865
[Duff79], 317, 860
[DuMP93], 521, 841
[Dwye87], 725, 838

[Earn85], 839, 843
[Earn88], 844, 861
[EdaL99], 215, 845
[Edel87], 540, 720, 838
[EdeM90], 215, 845
[Egga98], 164, 856
[ElbC96], 330, 859
[ElbC97], 681, 836
[ElbK99], 186, 853
[ElLK97], 643, 644, 855
[Elsa83], 532, 845
[Elvi92], 358, 865
[EnKP84], 11, 848
[EtzR99], 722, 838

[Falc85], 804, 813, 842
[FaNO89], 557, 579, 850
[FanP93], 725, 838

[FanP95], 725, 838
[Fari83], 532, 845
[Fari87], 836, 837, 844, 845, 846, 847, 850, 851, 852
[Fari89], 436, 514, 861
[Fari92a], 460, 861
[Fari92b], 557, 850
[Fari95], 433, 512, 861
[Fari97], 389, 399, 411, 417, 424, 441, 444, 452, 455, 467, 494, 505, 514, 517, 522, 535, 844
[FarJ94], 853
[FarN90a], 641, 642, 643, 855
[FarN90b], 643, 855
[Faro85], 855
[Faro86], 855
[Faro87], 557, 850
[Faro91], 467, 861
[Faro92], 459, 644, 849
[FarR87], 467, 861
[FarR88], 467, 861
[FarR98], 188, 853
[FarS95], 644, 855
[FauL01], 139, 848
[FauP79], 522, 844
[Feat87], 146, 859
[Fede69], 804, 842
[Ferg64], 533, 845
[Ferr94], 8, 855
[Ferw01], 295, 854
[Figu95], 590, 840
[FilB89], 633, 845
[Fili86], 521, 591, 592, 856
[FiMM86], 558, 573, 591, 592, 845
[FisB85], 53, 858
[Fish90a], 303, 837
[Fish90b], 29, 32, 858
[Fish94], 844, 853
[Fium89], 4, 854
[FoFC82], 842
[FolR93], 526, 845
[Forr72], 459, 861
[Fors12], 519, 841
[Fors95], 645, 855
[Fort87], 722, 725, 838
[Fort95], 213, 845
[Four95], 795, 842
[Free69], 858
[Free80], 843, 859, 860, 864
[Frie63], 783, 842
[FTAT00], 188, 865
[FuAG83], 270, 864
[FuKN80], 270, 864
[FuKU77], 631, 839
[Full73], 836
[FuTI86], 342, 858
[FVFH90], 46, 47, 59, 88, 264, 284, 290, 308, 351, 843

[Gall00], 505, 506, 508, 521, 530, 844
[Galt89], 839
[GanD82], 632, 839
[Garl85], 726, 852
[Garl87], 726, 852
[GarW89], 558, 850
[GarZ79], 571, 849
[Gass83], 844
[GeCG99], 845
[GelD95], 186, 188, 853
[Geor92], 248, 854
[GerP90], 333, 837
[GHSV93], 206, 209, 211, 215, 216, 217, 221, 223, 224, 844
[GibB85], 853
[GJPT78], 716, 838
[Glas84], 342, 858
[Glas86], 858
[Glas89], 858
[Glas90], 837, 841, 854, 858, 859, 861
[Glas92], 806, 842
[Glas95], 44, 310, 767, 790, 792, 795, 841
[Glas99], 786, 842
[Glid97], 11, 848
[Gold83], 480, 579, 856
[Gold90], 150, 854, 855
[GolM87], 579, 850
[GolS87], 850
[GomV98], 795, 842
[GonN02], 621, 849
[GonW87], 44, 767, 841
[Gord69], 846
[Gord71], 846
[GorR74a], 534, 846
[GorR74b], 861
[GoTG84], 323, 860
[Gour71], 316, 860
[GraK97], 557, 558, 565, 850
[Grav95], 637, 840
[Gray98], 519, 841
[Gree86], 328, 863

[Gree94], 578, 850
[Gree99], 337, 859
[Greg74], 861
[Greg86], 841, 844, 851
[Greg89], 456, 522, 846
[GreH98], 106, 837
[GreS77], 725, 838
[Grif75], 557, 864
[Grif78], 557
[Grif78a], 864
[Grif78b], 864
[GSPC77], 11, 848
[GSPC79], 11, 848
[GueP90], 635, 636, 638, 840
[GuiS85], 725, 838
[Gunn93], 693, 865
[GupS81], 46, 858

[HaAG83], 557, 850
[HaFN95], 844, 845, 847, 855, 862, 863
[Hage92], 526, 862
[HagH95], 525, 862
[Hain89], 579, 858
[Hain94], 238, 854
[Hall89], 294, 308, 309, 334, 837
[HalM63], 685, 864
[HaMa95], 461, 465, 857
[HaMF94], 693, 865
[Hami69], 857
[Hand89], 837, 844, 846
[HanH92], 693, 865
[Hanr89], 579, 858
[HarA02], 175, 857
[Hara79], 863
[HecG97], 597, 846
[HecH84], 343, 858
[Heck82], 333, 837
[Heck86], 325, 863
[Heck90a], 49, 858
[Heck90b], 32, 858
[Heck90c], 40, 858
[Heck93], 842
[Heck94], 838, 850, 851, 852, 854, 863
[Heck97], 846
[HEFS85], 566, 573, 591, 851
[HeKE99], 579, 850
[HerB87], 592, 855
[Herm98], 4, 23, 24, 26, 362, 854
[Hers75], 857
[Hill01], 340, 843
[Hill90], 340, 843
[Hiro74], 160, 856
[HKBZ97], 342, 858
[HMPY97], 537, 557, 579, 851
[HMSP96], 537, 546, 851
[HMSV99], 855
[Hoch83], 532, 846
[Hodg92], 132, 854
[HofB97], 836, 844, 854
[Hoff89], 213, 215, 538, 568, 569, 577, 615, 619, 620, 621, 710, 844
[Hoff91], 853
[Hoff93], 849
[Hoff94], 186, 853
[HofH85], 676, 836
[HofH87], 676, 836
[HofR95], 844, 845, 847, 851, 853, 855, 856
[Hogg92], 146, 149, 852
[HoHK89], 213, 215, 846
[Hohm91], 570, 580, 850
[HoLe88], 842
[Hopp96], 597, 846
[Horn75], 745, 854
[HosL93], 178, 444, 445, 446, 448, 460, 461, 468, 494, 502, 522, 526, 538, 546, 579, 625, 643, 672, 844
[HsuT98], 681, 836
[HuaM02], 633, 846
[HuPY96a], 213, 852
[HuPY96b], 213, 852
[HurW48], 802, 803, 804, 863
[HutH96], 526, 846
[HwaA92], 859

[IGES88], 822, 848
[ISO88], 11, 849

[JGMH88], 843, 859, 860, 863, 864, 865
[JiMa97], 196, 846
[John87], 745, 754, 842
[John93], 521, 579, 841
[JoLH73], 839
[JulB81], 863
[Jule62], 863
[Just92], 223, 846

[KaCY93], 180, 358, 865
[Kaji86], 309, 323, 859
[KakG96], 651, 840

[KaLL83], 857
[Kalv92], 594, 865
[Kap85], 839
[Kapl85], 273, 858
[Kapo99], 660, 843
[KassB93], 856
[Kauf98], 182, 865
[KayK86], 230, 340, 859
[KemF97], 11, 849
[KimD93], 459, 849
[KimK03], 587, 846
[Kirk92], 850, 854, 856, 859
[KlaK92], 678, 681, 851
[Klas80], 525, 846
[Klas94], 578, 851
[Kobb96], 461, 846
[Kopa91], 558, 851
[KopM83], 545, 547, 551, 840
[Kost91], 593, 846
[KraM00], 517, 521, 841
[KriM97], 579, 580, 851
[KrLM98], 650, 840
[KrPP90], 570, 851
[KrPW92], 567, 851
[KSHS03], 655, 657, 843
[Kuip99], 765, 857
[KumM94], 600, 863
[KumM95], 600, 863
[KuSP02], 645, 855
[KuSP03], 645, 855
[Kypr80], 194, 846

[LaCJ94], 853
[LacL94], 360, 865
[LamM95], 579, 851
[LamM96], 579, 851
[LanR80], 544, 545, 547, 551, 572, 573, 592, 602, 846
[LanR83], 430, 861
[LanS86], 844
[LasB95], 614, 863
[LaTH86], 206, 839
[LazV99], 188, 853
[LBDW92], 186, 838
[LCWB80], 290, 572, 573, 864
[LeeE82], 438, 439, 861
[LeeP77], 716, 838
[Levo88], 865
[Levo90], 362, 865
[LiaB83], 71, 86, 837
[LiaB84], 77, 837
[LiCN98], 182, 358, 865
[LiCr97], 458, 840
[LicS87], 591, 846
[LiHS02], 193, 846
[Limi44], 674, 836
[Lind68], 190, 842
[Lind92], 333, 334, 859
[LinM96], 215, 838
[LiON02], 198, 846
[Lisc94], 725, 838
[LiSH92], 591, 856
[LiuW02], 531, 861
[LoDW97], 795, 842
[LooD89], 532, 861
[Loop87], 530, 862
[LorC87], 365, 368, 865
[LorW86], 844
[Lü95], 644, 855
[Lueb01], 597, 846
[Luka89], 538, 566, 851
[LukC96], 438, 516, 861
[Luke96], 600, 863
[LuMM95], 579, 851
[LWZL02], 461, 846
[LycS89], 844, 845, 846, 850, 860

[MacB79], 857
[Maek99], 638, 855
[MaeP93], 639, 643, 855
[MagT87], 290, 291, 843
[Mail92], 89, 93, 837
[Malc77], 445, 446, 861
[MaLe98], 570, 580, 851
[ManC90], 840
[ManC92a], 458, 840
[ManC92b], 835
[Mand83], 190, 800, 801, 806, 843
[ManD94], 551, 578, 851
[ManK97], 551, 578, 851
[MäNS96], 688, 846
[Mant88], 173, 844
[MaPS86], 521, 579, 841
[MarM89], 567, 579, 851
[MarM91], 558, 851
[MarS89], 846
[Mart82], 517, 521, 841
[Mart87], 837, 844
[Mart94], 846
[Matv03], 693, 836

[McCa98], 211, 854
[Mcil92], 58, 858
[Meag82a], 860
[Meag82b], 860
[MeeW00], 650, 840
[MeeW90], 643, 855
[Mehl74], 445, 446, 861
[MeSS92], 630, 632, 839
[Micr94], 15, 848
[Miel91], 59, 843
[Mill86], 846
[Mill87], 579, 851
[Mill99], 150, 854
[MiMP87], 660, 662, 663, 667, 843
[MitR68], 857
[MiTW73], 857
[MitW78], 748, 842
[Miur00], 650, 840
[MölH99], 859
[Moon99], 460, 849
[Moor66], 726, 744, 852
[Moor79], 726, 852
[Morg83], 571, 849
[Morr90], 53, 858
[Morr91], 246, 854
[MorS92], 526, 862
[Mort85], 160, 178, 522, 559, 844
[Mort89], 852
[Mull96], 841, 844
[Mulm94], 838

[Nack82], 188, 853
[NarM95], 718, 838
[Nasr00], 530, 862
[Nasr87], 529, 573, 846
[Nata61], 773, 780, 781, 859
[NeNS72], 269, 865
[NeuN95], 358, 857
[Neva82], 324, 863
[NFMD90], 866
[NiBl94], 201, 202, 203, 846
[Niel74], 468, 861
[NiLN87], 81, 84, 837
[NinB93], 594, 595, 849
[NowR83], 532, 847

[OckS84], 579, 847
[Orou94], 718, 838
[OttP92], 842
[OweR87], 579, 851

[Paet90], 333, 837
[Paet95], 838, 840, 852, 854
[PalB98], 841
[PaPV95], 193, 847
[ParK96], 630, 632, 839
[Patr92], 538, 851
[Patr93], 538, 851
[Paul82], 146, 859
[Pavl82], 26, 32, 854
[Pedd92], 15, 848
[PenP86], 139, 164, 856
[PepH92], 752, 842
[PePR99], 483, 847
[Pete95], 526, 862
[Peter94], 602, 863
[PetR97], 529, 862
[PFTV86], 654, 855
[Pham92], 638, 855
[PhiO84], 567, 851
[PicS83], 857
[Pieg91], 433, 512, 861
[Pieg92], 557, 851
[PieR95], 601, 864
[PieT00], 517, 861
[PieT95], 409, 411, 430, 433, 436, 438, 445, 502, 512, 514, 516, 517, 531, 632, 861
[PieT98], 601, 864
[Pitt67], 40, 858
[Pitt85], 839
[PitW80], 46, 858
[Podg02], 193, 847
[PokG89], 85, 266, 415, 843
[Port81], 857
[Pösc84],526, 847
[Pott95], 644, 855
[Powe72], 567, 851
[PraG86], 538, 554, 557, 567, 570, 851
[PraG92], 502, 505, 861
[Pras91], 241, 854
[Prat87a], 198, 847
[Prat87b], 198, 847
[Prat89], 837
[Prat90], 517, 520, 521, 679, 841
[Prat95], 521, 841
[Prat96], 521, 841
[Prat97], 521, 841
[PraW85], 194, 847
[PreS85], 232, 233, 720, 838
[Preu86], 839

[RamG03], 186, 853
[Rams88], 419, 420, 861
[Rams89], 469, 861
[RanR91], 526, 847
[Rasa90], 444, 861
[ReBl85], 190, 843
[RedT95], 853
[Reev83], 190, 843
[Reif95], 527, 862
[RenE03], 199, 847
[Requ80], 847
[Requ96], 688, 847
[ReqV82], 160, 847
[ReqV83], 847
[ReqV85], 205, 208, 847
[RhyT01], 361, 866
[Ries75], 465, 862
[Ritt90], 345, 859
[RoaM92], 682, 837
[Robe63], 266, 865
[Rock70], 711, 840
[Rock90], 551, 630, 851
[Rock93], 418, 861
[RocO87], 676, 837
[RogA90], 134, 164, 388, 389, 433, 512, 852
[Roge01], 433, 512, 534, 861
[RogE90], 844, 847
[Roge98], 32, 46, 48, 58, 235, 266, 276, 284, 290, 291, 294, 307, 365, 843
[RoHD89], 599, 600, 601, 606, 864
[Roll95], 196, 847
[Rose79], 854
[RosG64], 668, 841
[RosK76], 26, 28, 44, 184, 185, 767, 841
[RosR84], 678, 837
[RosR86], 557, 855
[RosT91], 844, 850, 854
[RosV89], 206, 839
[Roth82], 348, 349, 859
[RouB96a], 638, 840
[RouB96b], 638, 840
[RoWW90], 40, 858

[Sabi85], 625, 839
[Sabi90], 473, 526, 532, 847
[Salo99], 360, 494, 502, 845
[Same84], 860
[Same90a], 860
[Same90b], 860
[SamW88], 860
[SaPD88], 855
[Sapi92], 650, 840
[SaRE76], 169, 847
[Sarr98], 525, 862
[SaWS95], 460, 849
[SBGS69], 268, 865
[Scho46], 861
[Scho67], 408, 534, 861
[SchS95], 347, 854
[Schu93], 856
[Schw97], 668, 841
[ScML98], 181, 182, 367, 369, 866
[SeCK89], 570, 580, 851
[SéCM95], 461, 525, 847
[Sede87], 847
[Sede89], 551, 851
[SedM88], 570, 580, 851
[SedN90], 538, 547, 548, 551, 852
[SedP86], 538, 545, 551, 852
[SedW87], 459, 849
[Seel66], 767, 773, 778, 780, 781, 786, 788, 842
[Seid89], 411, 418, 422, 425, 426, 429, 430, 862
[Seid93], 418, 862
[SeWZ89], 547, 847
[SeZZ89], 615, 849
[ShaB84], 461, 862
[Shaf94], 756, 836
[ShaM95], 192, 193, 194, 195, 197, 198, 211, 687, 845
[Shap91], 224, 847
[ShAR95], 185, 853
[ShAR96], 184, 185, 853
[ShaS86], 664, 666, 843
[ShaT82], 637, 638, 840
[ShaV91a], 209, 211, 839
[ShaV91b], 211, 839
[ShaV93], 209, 211, 839
[ShaV95], 221, 847
[SheH92], 601, 864
[SheJ87], 579, 852
[Shen00], 679, 837
[Shen94], 578, 852
[Shen95], 578, 852
[Shen98], 679, 837
[Shew96], 725, 838
[ShiG95], 603, 604, 856
[ShiK91], 188, 847
[ShiS98], 139, 847

[ShKK91], 188, 847
[Shoe85], 147, 857
[Shoe91], 766, 857
[Shoe93], 146, 149, 857
[ShPB95], 185, 853
[ShPB96], 185, 853
[Smit79], 29, 32, 858
[Smit84], 190, 843
[Snyd92], 175, 177, 615, 624, 726, 732, 735, 736, 738, 739, 741, 742, 845
[Snyd92a], 726, 852
[SodT94], 192, 847
[Spie69], 776, 777, 859
[Spiv65], 249, 835
[StaH97], 596, 624, 856
[Stam98], 530, 862
[Stee51], 756, 836
[SteL00], 43, 858
[StFF91], 181, 866
[STGLS97], 186, 187, 188, 854
[Stil98], 756, 857
[StoD89], 456, 458, 459, 862
[Stoy92], 568, 852
[StrS89], 836, 837, 841, 845, 849, 850, 851
[SuDr95], 725, 839
[SuHH99], 228, 847
[SuLi83], 456, 457, 862
[SuLi89], 456, 460, 525, 526, 845
[SunS92], 342, 859
[SuSS74], 264, 293, 865
[Sutc80], 554, 839
[Suth65], 685, 864
[SutH74], 84, 837
[Szil91], 681, 837

[TaJS99], 854
[Taub94], 614, 836
[Tayl79], 764, 765, 857
[THBP90], 369, 866
[TheF97], 526, 840
[Thom90], 46, 858
[TilH84], 644, 855
[Till83], 435, 862
[Tilo80], 160, 208, 847
[Timm77], 559, 852
[Timm96], 11, 849
[TorS67], 314, 860
[TSGCV97], 854
[Turn88], 545, 557, 852
[Vale64], 711, 840
[VaMV89], 672, 837
[VanA84], 839
[VanD88], 11, 849
[VanG95], 249, 854
[VanN85], 40, 57, 58, 858
[VanW96], 459, 848
[Vatt92], 98, 837
[VaVM89], 672, 837
[VeDG99], 596, 597, 856
[VelB94], 799, 843
[Verm94], 185, 854
[VeVC94], 572, 849
[VFLL00], 685, 686, 864
[VigB95], 603, 604, 864
[ViMV94], 672, 679, 681, 683, 837
[VSBJ98], 592, 848

[WaCF92], 859
[WaCG87], 357, 857
[Wall90], 423, 855
[Wang81], 456, 457, 862
[Wang92], 580, 852
[Warn69], 273, 865
[Watk70], 278, 865
[WatP98], 185, 327, 854
[Wats86], 571, 849
[Wats89], 848
[Watt90], 843
[WatW92], 146, 149, 290, 291, 308, 318, 323, 327, 342, 351, 365, 859
[Week85], 693, 865
[WeiA77], 85, 274, 865
[WeiD97], 325, 863
[Weil80], 85, 837
[Weil85], 200, 201, 202, 848
[Weil94], 235, 237, 854
[Wern79], 445, 862
[West90], 359, 866
[Whit80], 322, 337, 343, 860
[Whit85], 343, 859
[Widd71], 784, 788, 842
[Will78], 318, 860
[Will83], 327, 836
[WilM93], 579, 852
[Wils87], 215, 839
[Wilt87], 8, 855
[WilV90a], 362, 866
[WilV90b], 367, 866
[WNDS99], 11, 849

[WolF97], 650, 840
[Woll00], 650, 840
[WolT92], 651, 840
[Wolt95], 854
[WooA98], 326, 863
[Wood87], 672, 673, 674, 837
[WooT85], 202, 848
[WoPF90], 318, 860
[Wrig85], 571, 849
[WriS00], 11, 849
[WuAn99], 569, 852
[Wüth98], 23, 858
[WuXi92], 837
[WyMW86], 365, 595, 866
[Wyvi90], 40, 858
[YaCK92], 365, 866
[YanF64], 764, 857
[Ye96], 651, 840
[YeMa99], 580, 852
[YKFT84], 860
[YooT98], 866
[Yu92], 213, 848
[YuGD91], 854
[YuMS01], 633, 862
[YuPM00], 484, 841

[ŽalC99], 606, 719, 839
[ZheS00], 593, 856
[ZhoS99], 228, 852
[ZorS99], 526, 530, 862

Index of Algorithms

Algorithm 2.3.1
Border following, 27
Algorithm 2.4.1
Basic fill, 29
Algorithm 2.4.2
Smith seed fill, 30
Algorithm 2.4.3
Fishkin seed fill, 33
Algorithm 2.5.2.1
Basic line drawing, 40
Algorithm 2.5.2.2
Bresenham line drawing, 41
Algorithm 2.5.3.1
Midpoint line drawing, 43
Algorithm 2.9.1.1
Ordered edge list fill, 51
Algorithm 2.9.2.1
Bresenham circle drawing, 56
Algorithm 2.9.2.2
Improved circle drawing, 58
Algorithm 3.2.1.1
Cohen-Sutherland line clipping, 74
Algorithm 3.2.3.1
Liang-Barsky line clipping, 78
Algorithm 3.2.3.2
Liang-Barsky line clipping, 79
Algorithm 3.2.4.1
Nicholl-Lee-Nicholl clipping, 82
Algorithm 3.3.3.1
Liang-Barsky polygon clipping, 89
Algorithm 3.3.4.1
Maillot polygon clipping, 90
Algorithm 3.3.4.2
Maillot polygon clipping, 93
Algorithm 3.3.4.3
extended clipping code, 97
Algorithm 3.3.5.1
Vatti polygon clipping, 101
Algorithm 3.3.6.1
Greiner-Hormann's Phase 3, 108
Algorithm 4.6.1
homogeneous coordinate clipping, 127
Algorithm 4.6.2
homogeneous coordinate clipping, 128
Algorithm 4.6.3
nontrivial part of clipping, 129
Algorithm 5.3.3.1
divide-and-conquer in CSG, 170
Algorithm 5.8.1.1
Computing the inverse adjacency relation, 201
Algorithm 7.3.1
painter's algorithm, 268
Algorithm 7.5.1
BSP tree algorithm, 272
Algorithm 7.6.1
Warnock algorithm, 273
Algorithm 7.7.1
Z-buffer algorithm, 276
Algorithm 7.7.2
scan line Z-buffer algorithm, 277
Algorithm 7.8.1
Watkins visible surface algorithm, 279
Algorithm 8.6.1
Converting RBG to HSV, 303
Algorithm 8.6.2
Converting HSV to RGB, 304
Algorithm 8.6.3

Converting RBG to HSL, 305
Algorithm 8.6.4
Converting HSL to RGB, 306
Algorithm 10.2.1
ray tracing program, 340
Algorithm 10.2.2
shade function, 341
Algorithm 10.2.3.1
CSG ray intersection, 350
Algorithm 10.3.1
Gauss-Seidel radiosity algorithm, 353
Algorithm 10.3.2
progressive refinement for radiosity, 354
Algorithm 10.4.1.1
26-connected line drawing, 364
Algorithm 10.4.1.2
6-connected tripod line drawing, 366
Algorithm 11.4.1
de Casteljau algorithm, 401
Algorithm 11.5.2.1
de Casteljau algorithm, 424
Algorithm 11.5.2.2
de Boor algorithm, 426
Algorithm 11.5.4.1
B-spline span finding algorithm, 437
Algorithm 11.5.4.2
B-spline evaluation algorithm, 439
Algorithm 11.5.4.3
NURBS curve evaluation algorithm, 441
Algorithm 11.5.4.4
NURBS curve derivatives algorithm, 442
Algorithm 12.12.2.1
de Casteljau algorithm, 508
Algorithm 13.2.1
are convex sets disjoint, 541
Algorithm 13.4.3.1
faceted surface sectioning, 555
Algorithm 14.3.1
adaptive curve subdivision algorithm, 590
Algorithm 14.3.2
Outline for implicit tilers, 594
Algorithm 14.4.1
trapezoid creation algorithm, 607
Algorithm 14.5.1.1
Incremental curve tiling algorithm, 616
Algorithm 14.7.1
B-spline skinning algorithm, 632
Algorithm 14.7.2
Procedural skinning algorithm, 634
Algorithm 14.8.1
arc length algorithm, 635
Algorithm 14.8.2
arc length table building algorithm, 636
Algorithm 15.3.2.1
generating edge sequences, 665
Algorithm 17.2.1
Finding the splitting node, 699
Algorithm 17.2.2
A 1d range query algorithm, 700
Algorithm 17.2.3
Building a range tree, 701
Algorithm 17.2.4
A 2d range query algorithm, 702
Algorithm 17.3.1
The interval tree query algorithm, 705
Algorithm 17.3.2
The segment tree query algorithm, 707
Algorithm 17.3.3
Segment tree insertion algorithm, 708
Algorithm 17.6.1
Triangulation algorithm for monotone polygons, 719
Algorithm 18.4.1
constraint solution, 736
Algorithm 18.4.2
Merging intervals into components, 737
Algorithm 18.5.1
Implicit Curve Approximation, 740
Algorithm 18.6.1
constrained minimization, 743
Algorithm 22.4.1
Deterministic IFS, 807
Algorithm 22.4.2
Nondeterministic IFS, 808