# Computer Animation

## Algorithms and Techniques

Rick Parent

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ACADEMIC PRESS
A Harcourt Science and Technology Company
SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

| | |
|---|---|
| Executive Editor | Diane D. Cerra |
| Publishing Services Manager | Scott Norton |
| Assistant Developmental Editor | Belinda Breyer |
| Editorial Assistant | Ramona Buehler |
| Cover Design | Ross Carron Design |
| Text Design | Mark Ong, Side by Side Studios |
| Supplementary Technical Illustration | Tom Webster, LineWorks |
| Composition | Nancy Logan |
| Copyeditor | Sheila Berg |
| Proofreader | Karen Stough |
| Indexer | Ty Koontz |
| Printer | Courier Corporation |

Cover Images (from top): Detail of still from *Geri's Game,* courtesy Pixar Studios; simulation of volcano spitting fire, courtesy Katie Boner and Barb Olsafsky; simulation of waves hitting shore, courtesy Arun Somasundaram; "Bragger in Beantown," courtesy John Chadwick; robot, courtesy Jessica Hodgins; facial model, courtesy Scott King; driving simulation, courtesy Norm Badler

Interior Images: Figures 5.26 and 5.27 (plates 4 and 5), courtesy David S. Ebert; Figure 5.28 (plate 6) courtesy Ruchir Gatha and David S. Ebert; figure 6.26, courtesy David S. Ebert; figures 6.28 and 6.31, courtesy Frederic I. Parke; figure 6.39, "Face and 2d Mesh," reprinted from *Signal Processing: Image Communication* 15, nos. 4-5 (Tekalp, 2000): pp. 387-421, by permission of Elsevier Science; figure A.3 (plate 11), courtesy Disney Enterprises, Inc.; figure A.11 (plate 12), courtesy Pixar Studios, all rights reserved; figure A.12 (plate 13), courtesy Disney Enterprises, Inc..

Morgan Kaufmann Publishers
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA
*http://www.mkp.com*

ACADEMIC PRESS
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
*http://www.academicpress.com*

Academic Press
Harcourt Place, 32 Jamestown Road, London, NW1 7BY, United Kingdom
*http://www.academicpress.com*

06 05 04 03 02 5 4 3 2 1

**Library of Congress Control Number: 2001090620**
ISBN: 1-55860-579-7

This book is printed on acid-free paper.

To Mom and Dad

To John and Kim for teaching me to keep things in perspective

**Foreword** xv

**Preface** xvii

- **Overview** xvii
- **Organization of the Book** xx
- **Acknowledgments** xxi
- **References** xxii

**1 Introduction** 1

- **Perception** 2
- **The Heritage of Animation** 4
  - Early Devices 4
  - The Early Days of ˇConventionalÓAnimation 6
  - Disney 7
  - Contributions of Others 9
  - Other Media for Animation 9
  - Principles of Computer Animation 10
- **Animation Production** 12
  - Computer Animation Production Tasks 15
  - Digital Editing 17
  - Digital Video 20
- **A Brief History of Computer Animation** 21
  - Early Activity 21
  - The Middle Years 24
- **Chapter Summary** 29
- **References** 29
- **Spaces and Transformations** 31

**2 Technical Background** 31

- **The Display Pipeline** 32
- **Homogeneous Coordinates and the Transformation Matrix** 36

Compounding Transformations: Multiplying Transformation Matrices .......................................................... 39

Basic Transformations ............................................................ 40

Representing an Arbitrary Orientation ................................... 42

Extracting Transformations from a Matrix .............................. 45

Description of Transformations in the Display Pipeline ........ 46

Round-off Error Considerations .............................................. 48

Orientation Representation .................................................... 51

Fixed Angle Representation ................................................... 53

Euler Angle Representation ................................................... 56

Angle and Axis ......................................................................... 57

Quaternions .............................................................................. 58

Chapter Summary .................................................................... 61

References ................................................................................ 61

Interpolation ............................................................................ 63

3 Interpolation and Basic Techniques ........................... 63

The Appropriate Function ........................................................ 64

Controlling the Motion Along a Curve .................................... 68

Computing Arc Length ............................................................. 69

Speed Control ........................................................................... 84

Ease-in/Ease-out ...................................................................... 86

Constant Acceleration: Parabolic Ease-In/Ease-Out ............. 89

General Distance-Time Functions .......................................... 92

Curve Fitting to Position-Time Pairs ...................................... 94

Interpolation of Rotations Represented by Quaternions ....... 97

Path Following ........................................................................ 102

Orientation along a Path ........................................................ 102

Smoothing a Path .................................................................... 108

Determining a Path along a Surface ..................................... 114

Key-Frame Systems ............................................................... 116

Animation Languages ............................................................ 120

Artist-Oriented Animation Languages .................................... 121
Articulation Variables ................................................................ 122
Graphical Languages ................................................................. 123
Actor-Based Animation Languages ......................................... 123

**Deforming Objects** .................................................................. **124**
Warping an Object ...................................................................... 125
Coordinate Grid Deformation .................................................. 125

**Morphing (2D)** ........................................................................... **143**
Coordinate Grid Approach ........................................................ 143
Feature-Based Morphing .......................................................... 148

**3D Shape Interpolation** ............................................................ **153**
Matching Topology ..................................................................... 157
Star-Shaped Polyhedra ............................................................. 157
Axial Slices ................................................................................. 159
Map to Sphere ............................................................................ 160
Recursive Subdivision ............................................................... 166
Summary ..................................................................................... 169

**Chapter Summary** ..................................................................... **169**

**References** .................................................................................. **170**

**4 Advanced Algorithms** ............................................................ **173**

**Automatic Camera Control** ...................................................... **174**

**Hierarchical Kinematic Modeling** ........................................... **175**
Representing Hierarchical Models .......................................... 177
Forward Kinematics ................................................................... 184
Local Coordinate Frames .......................................................... 185
Inverse Kinematics ..................................................................... 192
Summary ..................................................................................... 203

**Rigid Body Simulation** ............................................................. **203**
Bodies in Free Fall ..................................................................... 204
Bodies in Contact ....................................................................... 216

**Enforcing Soft and Hard Constraints** .................................... **231**
Flexible Objects .......................................................................... 232
Virtual Springs ............................................................................ 235
Energy Minimization .................................................................. 236

Space-Time Constraints .......................................................... 239

**Controlling Groups of Objects ................................................ 241**

Particle Systems ...................................................................... 242

Flocking Behavior .................................................................... 246

Autonomous Behavior ............................................................. 257

**Implicit Surfaces .................................................................... 261**

Basic Implicit Surface Formulation ......................................... 261

Animation Using Implicitly Defined Objects ............................ 263

Collision Detection .................................................................. 264

Deforming the Implicit Surface as a Result of Collision .......... 265

Summary ................................................................................. 267

**Chapter Summary ................................................................. 267**

**References ............................................................................ 268**

**5 Natural Phenomena .................................................... 271**

**Plants ..................................................................................... 272**

A Little Bit of Botany ............................................................... 273

L-Systems ............................................................................... 275

Animating Plant Growth .......................................................... 280

Summary ................................................................................. 283

**Water ..................................................................................... 283**

Still Waters and Small-Amplitude Waves ............................... 284

The Anatomy of Waves ........................................................... 288

Modeling Ocean Waves .......................................................... 289

Finding Its Way Downhill ........................................................ 292

Summary ................................................................................. 296

**Gaseous Phenomena ............................................................ 296**

General Approaches to Modeling Gas ................................... 296

Computational Fluid Dynamics ............................................... 298

Clouds ..................................................................................... 299

Fire .......................................................................................... 310

Summary ................................................................................. 311

**Chapter Summary ................................................................. 312**

**References ............................................................................ 312**

**6 Modeling and Animating Articulated Figures ............ 317**

**Reaching and Grasping ............ 318**
Modeling the Arm ............ 318
The Shoulder Joint ............ 321
The Hand ............ 321
Coordinated Movement ............ 324
Reaching Around Obstacles ............ 325
Strength ............ 326

**Walking ............ 327**
The Mechanics of Locomotion ............ 328
The Kinematics of the Walk ............ 333
Using Dynamics to Help Produce Realistic Motion ............ 336
Forward Dynamic Control ............ 338
Summary ............ 339

**Facial Animation ............ 339**
Types of Facial Models ............ 341
Creating the Model ............ 342
Textures ............ 345
Approaches to Animating the Face ............ 347

**Overview of Virtual Human Representation ............ 353**
Representing Body Geometry ............ 354
Geometry Data Acquisition ............ 356
Geometry Deformation ............ 356
Clothing ............ 357
Hair ............ 357
Surface Detail ............ 358

**Layered Approach to Human Figure Modeling ............ 358**

**Cloth and Clothing ............ 361**
Simple Draping ............ 362
Getting into Clothes ............ 365

**Motion Capture ............ 369**
Processing the Images ............ 371
Camera Calibration ............ 372
3D Position Reconstruction ............ 373
Fitting to the Skeleton ............ 376

Modifying Motion Capture .................................................... 378
Summary ......................................................................... 378

**Chapter Summary ................................................................ 379**

**References ....................................................................... 379**

**A Rendering Issues ...................................................... 385**

**A.1 Double Buffering ............................................................... 385**

**A.2 Compositing ...................................................................... 386**
A.2.1 Compositing without Pixel Depth Information ............... 389
A.2.2 Compositing with Pixel Depth Information .................... 395

**A.3 Displaying Moving Objects: Motion Blur .......................... 398**

**A.4 Drop Shadows .................................................................. 401**

**A.5 Summary .......................................................................... 406**

**References ....................................................................... 406**

**B Background Information and Techniques ................ 409**

**B.1 Vectors and Matrices ....................................................... 409**
B.1.1 Inverse Matrix and Solving Linear Systems.................. 411
B.1.2 Singular Value Decomposition ...................................... 419

**B.2 Geometric Computations ................................................. 420**
B.2.1 Components of a Vector .............................................. 420
B.2.2 Length of a Vector......................................................... 420
B.2.3 Dot Product of Two Vectors .......................................... 420
B.2.4 Cross Product of Two Vectors ...................................... 422
B.2.5 Vector and Matrix Routines........................................... 424
B.2.6 Closest Point between Two Lines in Three-Space ....... 427
B.2.7 Area Calculations .......................................................... 428
B.2.8 The Cosine Rule ........................................................... 429
B.2.9 Barycentric Coordinates................................................ 430
B.2.10 Computing Bounding Shapes ..................................... 431

**B.3 Transformations ................................................................ 447**
B.3.1 Transforming a Point Using Vector-Matrix Multiplication......................................................................... 447

B.3.2 Transforming a Vector Using Vector-Matrix Multiplication........................................................................ 448
B.3.3 Axis-Angle Rotations.................................................... 449
B.3.4 Quaternions ................................................................. 450

**B.4 Interpolating and Approximating Curves ......................... 453**
B.4.1 Equations: Some Basic Terms...................................... 453
B.4.2 Simple Linear Interpolation: Geometric and Algebraic Forms ........................................................................ 454
B.4.3 Parameterization by Arc Length.................................... 456
B.4.4 Computing Derivatives .................................................. 456
B.4.5 Hermite Interpolation.................................................... 457
B.4.6 Catmull-Rom Spline ..................................................... 458
B.4.7 Four-Point Form............................................................ 461
B.4.8 Blended Parabolas........................................................ 462
B.4.9 Bezier Interpolation/Approximation ............................... 463
B.4.10 De Casteljau Construction of Bezier Curves............... 464
B.4.11 Tension, Continuity, and Bias Control......................... 465
B.4.12 B-Splines.................................................................... 467
B.4.13 Fitting Curves to a Given Set of Points ....................... 469

**B.5 Randomness.................................................................... 470**
B.5.1 Noise ............................................................................ 471
B.5.2 Turbulence ................................................................... 474
B.5.3 Random Number Generator ......................................... 474

**B.6 Physics Primer ................................................................. 476**
B.6.1 Position, Velocity, and Acceleration.............................. 476
B.6.2 Circular Motion ............................................................. 477
B.6.3 NewtonÌs Laws of Motion............................................. 478
B.6.4 Inertia and Inertial Reference Frames .......................... 479
B.6.5 Center of Mass............................................................. 479
B.6.6 Torque.......................................................................... 480
B.6.7 Equilibrium: Balancing Forces ..................................... 480
B.6.8 Gravity.......................................................................... 480
B.6.9 Centripetal force........................................................... 481
B.6.10 Contact Forces........................................................... 481
B.6.11 Centrifugal Force ....................................................... 483
B.6.12 Work and Potential Energy ......................................... 484
B.6.13 Kinetic Energy............................................................ 484

B.6.14 Conservation of Energy ............................................ 484
B.6.15 Conservation of Momentum ........................................ 484
B.6.16 Oscillatory Motion ...................................................... 485
B.6.17 Damping ...................................................................... 486
B.6.18 Angular Momentum .................................................... 486
B.6.19 Inertia Tensors ............................................................ 487

**B.7 Numerical Integration Techniques .................................. 488**
B.7.1 Function Integration ...................................................... 488
B.7.2 Integrating Ordinary Differential Equations .................. 489

**B.8 Standards for Moving Pictures ........................................ 493**
B.8.1 In the Beginning, There Was Analog ............................ 493
B.8.2 In the Digital World ....................................................... 497

**B.9 Camera Calibration ........................................................... 502**

**References .......................................................................... 507**

**Index ..................................................................................... 509**

**About the Author .................................................................. 1999**

# Foreword

From its novel and experimental beginnings over thirty years ago, computer animation has grown to become a mainstream artistic tool in the hugely active industries of motion films, television and advertising. The area has attracted many computer scientists, artists, animators, designers, and production people.

The past decade has been a particularly exciting period in computer animation both for those engaged in the field and for the audiences of the many films that have used the technology. It has been exciting to see animation that could not have been done any other way, as well as films with special effects that blend animation and live action so well as to make them indistinguishable.

Underlying this enormous activity is the set of algorithms that comprise the actual software engine of computer animation, defining the powers of expression that are available to the animator. With the industry's maturity has come the need for a textbook that captures the art and mathematics behind the technology and that can serve as both an archival record and a teaching manual. With *Computer Animation: Algorithms and Techniques,* Rick Parent has done a terrific job of meeting this need. He has worked in the area of computer animation since its early days in the seventies, first as a student and now as a professor on the faculty at Ohio State University as well as a consultant and entrepreneur.

Just as we at the National Research Council of Canada were fortunate to work with artists and animators in the early days of developing computer animation techniques, so was Rick Parent at Ohio State University. Whereas we worked under the leadership of Nestor Burtnyk primarily in 2D layered cel animation, Rick embarked on applying 3D graphics methods that have become the basis of most commercial computer animation today. Because of this experience, Rick's book is both academically rigorous and eminently practical, presenting a thorough and up-to-date treatment of the algorithms used in computer animation.

Marceli Wein
Kingston, Ontario

*Marceli Wein is Researcher Emeritus of the National Research Council of Canada, where he had worked for thirty years as Research Officer. In the late sixties and seventies he worked with Nestor Burtnyk, who wrote the original key frame interpolation software that became the basis of the first computer animation system. Their work in collaboration with the National Film Board of Canada resulted in* Hunger *(1974), by Peter Foldes, which was the first computer-animated film to be nominated for an Academy Award. In 1997, Burtnyk and Wein received a Technical Award from the Academy of Motion Picture Arts and Sciences.*

# Preface

## Overview

This book surveys algorithms and programming techniques for specifying and generating motion for graphical objects. It is primarily concerned with three-dimensional (3D) computer animation. The main audience is advanced undergraduate or beginning graduate computer science students. Computer graphics programmers who want to learn the basics of computer animation programming and computer animators (digital artists) who want to better understand the underlying computational issues of animation software will also benefit from this book.

This book addresses practical issues and provides accessible techniques and straightforward implementations. Appendixes cover basic material that the reader may find useful as a refresher as well as specific algorithms for use in implementations. In some cases, programming examples are complete working code segments, in C, which can be copied, compiled, and run to produce basic examples of the algorithms discussed; other programming examples are C-like pseudocode that can be translated into working code. C was chosen because it forms the common basis for languages such as C++ and Java, and it lends itself to illustrating the step-by-step nature of algorithms. Purely theoretical discussions have been avoided as much as possible except to point out avenues of current and future research. The emphasis is on three-dimensional techniques; the two-dimensional (2D) issues are not completely ignored, but they are a minor part of the text.

This text is not intended for animators using off-the-shelf animation software (except to the extent that it might help in understanding the underlying computations required for a particular technique). It does not attempt to cover the theory of computer animation, address the aesthetics of computer animation, or discuss the artistic issues involved in designing animations. It does not detail the production issues in the actual commercial enterprise of producing a finished piece of animation. And, finally, it does not address the issue of *computer-assisted animation,* which, for our purposes, is taken to mean the computerization of conventional hand-drawn techniques; for the most part, that area has its own set of separate issues [1] [2]. The book does concentrate on full 3D computer animation and identifies the useful algorithms and techniques that animators and programmers can use to move objects in interesting ways.

To *animate* literally means "to give life to"; *animating* is moving something (or making something appear to move) that cannot move itself—whether it is a puppet of King Kong, a hand-drawn image of Snow White, the hands of a watch, or a synthetic image of a wooden toy cowboy. Animation has been used to teach and entertain from the early days of puppetry and continues to be used today in film and video. It is a powerful tool to spark the imagination of the child in all of us. Animation adds the dimension of time to computer graphics. This opens up great potential for transmitting information and knowledge to the viewer while igniting the emotions. Animation also creates a whole new set of problems for those who produce the images that constitute the animated sequence. To animate something, the animator has to be able to control, either directly or indirectly, how the *thing* is to move through time and space as well as how it might change its own shape or appearance over time.

The fundamental objective of computer animation programming is to select techniques and design tools that are expressive enough for animators to specify what they intend, yet at the same time are powerful enough to relieve animators from specifying any details they are not interested in. Obviously, no one tool is going to be right for every animator, for every animation, or even for every scene in a single animation. The appropriateness of a particular animation tool depends on the effect desired and the control required by the animator. An artistic piece of animation will usually require tools different from those required by an animation that simulates reality or educates a patient.

There is a distinction made here between *computer-assisted animation* and *computer-generated animation* [1] [3]. Computer-assisted animation refers to systems consisting of one or more two-dimensional planes that computerize the traditional (hand-drawn) animation process. Interpolation between key shapes is typically the only use of the computer in producing this type of animation (in addition to the non–motion control uses of the computer in tasks such as inking, shuffling paper, and managing data).

This book is concerned with computer-generated animation in which the animator is typically working in a synthetic three-dimensional environment with the objective of specifying the motion of both the cameras and the 3D objects (e.g., as in *Toy Story*). For discussion purposes, motion specification for computer-generated animation is divided into two broad categories, *interpolation and basic techniques* and *advanced algorithms*. These somewhat arbitrary names have been chosen to accentuate the computational differences among approaches to motion control. The former group can be thought of as *low level* because the animator exercises fine control over the motion and the expectations of the animator are very precise. The latter group comprises *high-level* algorithms in which control is at a coarser level with less of a preconceived notion of exactly what the motion will look like. Use of the term *algorithms* is meant to reinforce the notion of the relative sophistication of these techniques.

The interpolation and basic technique category consists of ways in which the computer is used to fill in the details of the motion once the animator specifies the required information. Examples of techniques in this category are key framing and path following. When using these techniques, animators typically have a fairly specific idea of the exact motion they want; these techniques give animators a direct and precise way of specifying and controlling the motion, with the computer's function limited to filling in appropriate numeric values that are required to produce the desired motion.

High-level procedural algorithms and behavioral models generate motion using a set of rules or constraints that specify *what* is to be done instead of *how* it is to be done. The animator chooses an appropriate algorithm or sets up the rules of a model and then selects initial values or boundary conditions. The system is then set into motion, so to speak, and the objects' motions are automatically generated by the algorithm or model. These approaches, such as particle systems and rigid body dynamics, often rely on sophisticated computations. To aid the reader in understanding and implementing the computations required for high-level algorithms, the appendixes provide background information on various mathematical areas related to animation, such as vector algebra, numerical integration techniques, and optimization techniques. Algorithms that require exceptionally sophisticated mathematical formulas and advanced training in physics, mechanical engineering, math, and so on are not emphasized in this text, although pointers to references are often given for the interested reader.

The motion control methods can also be discussed in terms of the level of abstraction at which the animator is working. At one extreme, at a very low level of abstraction, the animator could individually color every pixel in every frame using a paint program. At the other extreme, at a very high level of abstraction, the animator could (in an ideal world) tell a computer to “make a movie about a dog.” Presumably, the computer would whirl away while it computes such a thing. A high level of abstraction frees the animator from dealing with the myriad details required to produce a piece of animation. A low level of abstraction allows the animator to be very precise in specifying exactly what is to be displayed and when. In practice, animators want to be able to switch back and forth and to work at various levels of abstraction depending on the desired effect. Developing effective animation tools that permit animators to work at both high and low levels of abstraction is a particular challenge.

This distinction between basic techniques and advanced algorithms is made primarily for pedagogical purposes. In practice the collection of techniques and algorithms used in computer animation forms a continuum from low to high levels of abstraction. Each technique requires a certain amount of effort from the animator and from the computer. This distribution of workload between the animator and the computer is a distinguishing characteristic among animation techniques. Intuitively, low-level techniques tend to require more user input and employ fairly straightforward computation. High-level algorithms, on the other hand, require less specific information from the animator and more computation. The dividing line between the categories is an artificial construct used solely to provide a convenient method of grouping techniques for discussion in the book.

In addition, model-specific applications are surveyed. These have been grouped into the two general areas *natural phenomena* and *figure modeling*. The former concentrates on naturally occurring complex phenomena and includes plants, fire, smoke, and water. The latter concentrates on animating the human form and includes walking, reaching, and facial expression. The topics in the two chapters dealing with these areas tend to be more recently developed than topics found in earlier chapters and are therefore less well established. These chapters survey various approaches rather than offer specific solutions to specific problems.

## Organization of the Book

The first chapter discusses general issues related to animation, including perception, the technological history of hand-drawn animation, a survey of animation production, and a brief snapshot of the ever-evolving history of computer animation. These provide a broad perspective of the art and craft that is animation.

Chapter 2 presents general techniques and technical issues. It reviews computational issues in computer graphics to ensure a solid background in the techniques that are important in understanding the remainder of the book. This includes a review of the rendering pipeline and a discussion of the ordering of transformations to reduce round-off errors that can creep into a series of calculations as one builds on another. A detailed section on quaternion representation of orientation is presented in this chapter as well. If the reader is well versed in computer graphics, this chapter may be skimmed to pick up relevant terminology or skipped altogether.

Chapters 3 and 4 present the techniques and algorithms that form the basis for computer animation. Chapter 3 discusses the low-level, basic techniques, concentrating on interpolation. It also presents the basics of key-frame systems and animation languages. The high-level advanced algorithms are presented in Chapter 4, which begins with forward and inverse kinematics, followed by physically based techniques. The algorithms that are typified by emergent behavior are included: particle systems, flocking, and autonomous behavior. Finally, energy minimization techniques are introduced.

Chapters 5 and 6 present model-specific applications of computer animation. Chapter 5 focuses on approaches to modeling natural phenomena and covers the representation and animation of plants, water, and gases. Chapter 6 concentrates on figure animation and includes sections on activities of figures and common modeling issues: reaching and grasping, walking, facial expression, representing virtual humans, and clothes.

Appendix A presents rendering issues often involved in producing images for computer animation: double buffering, compositing, computing motion blur, and drop shadows. It assumes a knowledge of the use of frame buffers, of how a *z*-buffer display algorithm works, and of aliasing issues. Appendix B is a collection of relevant material from a variety of disciplines. It contains a survey of interpolation and approximation techniques, vector algebra and matrices, quaternion conversion code, the first principles of physics, several useful numeric techniques, and attributes of film, video, and image formats.

The Web page associated with the book, containing images, code, figures, and animations, can be found at *http://www.mkp.com/caat/*.

## Acknowledgments

Many people contributed in various ways to this book. First and foremost, I'd like to thank the front-line editor, my wife, Arlene, who both tolerated my preoccupation with this project and greatly increased the readability of the book. Several people contributed directly to the text: Scott King (facial animation), David Ebert (clouds), Doug Roble (motion capture), Matt Lewis (virtual humans), and Meg Geroch (reaching). Several other people generated software and images: Kovalan

Muniandy, Manu Varghese, Arunachalam Somasundaram, Kathryn Boner, Barbara Olsafsky, and Pete Carswell. Additional cover images were provided by John Chadwick, Jessica Hodgins, Norm Badler, and Pixar. Others who helped by reviewing sections as they were initially being developed include Julie Barnes, Suba Varadarajan, and Lawson Wade. I would also like to thank Morgan Kaufmann's reviewers and editors for seeing this project through.

## References

1. E. Catmull, "The Problems of Computer-Assisted Animation," *Computer Graphics* (Proceedings of SIGGRAPH 78), 12 (3), pp. 348–353 (August 1978, Atlanta, Ga.).
2. M. Levoy, "A Color Animation System Based on the Multiplane Technique," *Computer Graphics* (Proceedings of SIGGRAPH 77), 11 (2), pp. 65–71 ( July 1977, San Jose, Calif.). Edited by James George.
3. N. Magnenat-Thalmann and D. Thalmann, *Computer Animation: Theory and Practice,* Springer-Verlag, New York, 1985.

CHAPTER 1

# Introduction

Computer animation, for many people, is synonymous with big-screen events such as *Star Wars, Toy Story,* and *Titanic*. But not all, or arguably even most, computer animation is done in Hollywood. It is not unusual for Saturday morning cartoons to be entirely computer generated. Computer games take advantage of state-of-the-art computer graphics techniques. Real-time performance-driven computer animation has appeared at SIGGRAPH[1] and on *Sesame Street.* Desktop computer animation is now possible at a reasonable cost. Computer animation on the Web is routine. Digital simulators for training pilots, SWAT teams, and nuclear reactor operators are commonplace. The distinguishing characteristics of these various venues are the cost, the image quality desired, and the amount and type of interaction allowed. This book does not address the issues concerned with a particular venue, but it does present the algorithms and techniques used to do animation in all of them.

In computer animation, any value that can be changed can be animated. An object's position and orientation are obvious candidates for animation, but all of

[1] SIGGRAPH is the Association for Computing Machinery's (ACM's) Special Interest Group on Computer Graphics. The ACM is the main professional group for computer scientists.

the following can be animated as well: the object's shape, its shading parameters, its texture coordinates, the light source parameters, and the camera parameters.

To set the context for computer animation, it is important to understand its heritage, its history, and certain relevant concepts. This chapter discusses motion perception, the technical evolution of animation, animation production, and notable works in computer animation. It provides a grounding in computer animation as a field of endeavor.

## 1.1 Perception

Images can quickly convey a large amount of information because the human visual system is a sophisticated information processor. It follows, then, that moving images have the potential to convey even more information in a short time. Indeed, the human visual system has evolved to provide for survival in an ever-changing world; it is designed to notice and interpret movement.

When animation is created for later viewing it is typically recorded on film or video as a series of still images that when displayed in rapid sequence are perceived by an observer as a single moving image. This is possible because the eye-brain complex assembles the sequence of still images and interprets it as continuous movement. A single image presented to a viewer for a short time will leave an imprint of itself—the *positive afterimage*—in the visual system for a short time after it is removed (dramatically demonstrated by looking into the flash produced by a flash camera). This phenomenon is attributed to what has come to be called *persistence of vision*. When a person is presented with a sequence of closely related still images at a fast enough rate, persistence of vision induces the sensation of continuous imagery.[2] The afterimages of the individual stills fill in the gaps between the images. In both film and video, a sequence of images is recorded that can be played back at rates fast enough to fool the eye into interpreting it as continuous imagery. When the perception of continuous imagery fails to be created, the image is said to *flicker*. In this case, the animation appears as a rapid sequence of still images to the eye-brain. Depending on conditions such as room lighting and viewing distance, the rate at which single images must be played back in order to maintain the persistence of vision varies. This rate is referred to as the *flicker rate*.

Persistence of vision is not the same as perception of motion. Rotating a white light source fast enough will create the impression of a stationary white ring.

[2] Recently in the literature, it has been argued that persistence of vision is not the mechanism responsible for successfully viewing film and video as continuous imagery. I avoid controversy here and, for purposes of this book and consistent with popular practice, assume that whatever psychophysical mechanism is responsible for persistence of vision is also responsible for the impression of continuous imagery in film and video.

Although this effect can be attributed to persistence of vision, the result is static. The sequential illumination of a group of lights typical of a movie theater marquee produces the illusion of a lighted object circling the signage. Motion is perceived, yet persistence of vision is not involved because no individual images are present.

While persistence of vision addresses the lower limits for establishing the perception of continuous imagery, there are also upper limits to what the eye can perceive. The receptors in the eye continually sample light in the environment. The limitations on motion perception are determined by the reaction time of those sensors and by other mechanical limitations such as blinking and tracking. If an object moves too quickly with respect to the viewer, then the receptors in the eye will not be able to respond fast enough for the brain to distinguish sharply defined, individual detail; *motion blur* results. In sequences of still images, motion blur results as a combination of the object's speed and the time interval over which the scene is sampled. In a still camera, a fast-moving object will not blur if the shutter speed is fast enough relative to the object's speed. In computer graphics, motion blur will never result if the scene is sampled at a precise instant in time; to compute motion blur, the scene needs to be sampled over an interval of time or manipulated to appear as though it were. (See Appendix A for a discussion of motion blur calculations.) If motion blur is not calculated, then images of a fast-moving object can appear disjointed. The motion becomes jerky, similar to live action viewed with a strobe light, and is often referred to as *strobing*. In hand-drawn animation, fast-moving objects are typically stretched in the direction of travel so that the object's images in adjacent frames overlap, reducing the strobing effect.

As reflected in the discussion above, there are actually two rates that are of concern. One is the *playback rate,* the number of images per second displayed in the viewing process. The other is the *sampling rate,* the number of different images that occur per second. The playback rate is the rate related to flicker. The sampling rate determines how jerky the motion appears. For example, a television signal conforming to the National Television Standards Committee (NTSC) format displays images at a rate of thirty per second, but in some programs (e.g., some Saturday morning cartoons) there may be only six different images per second, with each image repeatedly displayed five times. Often, lip-synch animation is drawn on twos (every other frame) because drawing it on ones (animating it in every frame) appears too hectic. Some films display each frame twice to reduce flicker effects. On the other hand, because an NTSC television signal is interlaced (which means the odd-numbered scanlines are played beginning with the first sixtieth of a second and the even-numbered scanlines are played beginning with the next sixtieth of a second), smoother motion can be produced by sampling the scene every sixtieth of a second even though the complete frames are only played back at thirty frames per second. See Appendix B for details concerning various film and video formats.

## 1.2 The Heritage of Animation

In the most general sense, *animation*[3] includes "live-action" puppetry such as that found on *Sesame Street* and the use of mechanical devices to articulate figures such as in *animatronics*. However, I concentrate here on devices that use a sequence of individual stills to create the effect of a single moving image, because these devices have a closer relationship to computer animation.

### 1.2.1 Early Devices

*Persistence of vision* and the ability to interpret a series of stills as a moving image were actively investigated in the 1800s [2], well before the film camera was invented. The recognition and subsequent investigation of this effect led to a variety of devices intended as parlor toys. Perhaps the simplest of these early devices is the *thaumatrope,* a flat disk with images drawn on both sides and having two strings connected opposite each other on the rim of the disk (see Figure 1.1). The disk could be quickly flipped back and forth by twirling the strings. When flipped rapidly enough, the two images appear to be superimposed. The classic example uses the image of a bird on one side and the image of a birdcage on the other; the rotating disk visually places the bird inside the birdcage. An equally primitive technique is the *flipbook,* a tablet of paper with an individual drawing on each page. When the pages are flipped rapidly, the viewer has the impression of motion.

One of the most well known early animation devices is the *zoetrope,* or wheel of life. The zoetrope has a short, fat cylinder that rotates on its axis of symmetry. Around the inside of the cylinder is a sequence of drawings, each one slightly different from the ones next to it. The cylinder has long vertical slits cut into its side between each adjacent pair of images so that when it is spun on its axis each slit allows the eye to see the image on the opposite wall of the cylinder (see Figure 1.2, Plate 1). The sequence of slits passing in front of the eye as the cylinder is spun on its axis presents a sequence of images to the eye, creating the illusion of motion.

Related gizmos that use a rotating mechanism to present a sequence of stills to the viewer are the *phenakistoscope* and the *praxinoscope*. The phenakistoscope also uses a series of rotating slots to present a sequence of images to the viewer by positioning two disks rotating in unison on an axis; one disk has slits, and the other contains images facing the slits. One sights along the axis of rotation so the slits

[3] A more restricted definition of *animation,* also found in the literature, requires the use of a sequence of stills to create the visual impression of motion. The restricted definition does not admit techniques such as animatronics or shadow puppets under the rubric *animation*.

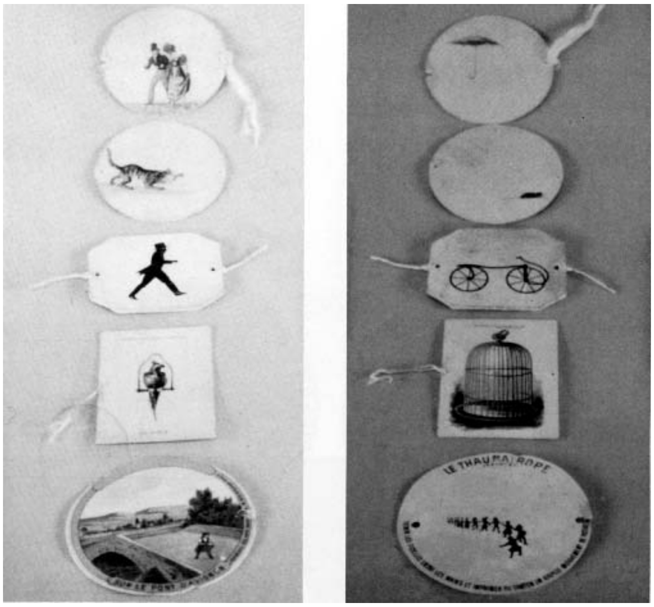


**Figure 1.1** A thaumatrope

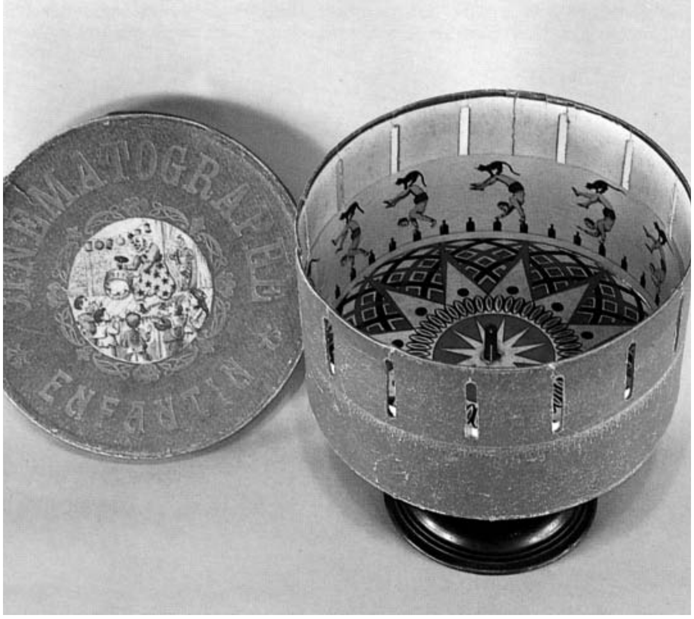


**Figure 1.2** A zoetrope

pass in front of the eye, which can thus view a sequence of images from the other disk. The praxinoscope uses a cylindrical arrangement of rotating mirrors inside a large cylinder of images facing the mirrors. The mirrors are angled so they reflect an observer's view of the images.

Just before the turn of the century, the moving image began making its way onstage. The magic lantern (an image projector powered by candle or lamp) and shadow puppets became popular theater entertainment [1]. On the educational front, Eadweard Muybridge investigated the motions of humans and animals. To show image sequences during his lectures, he invented the *zoopraxinoscope,* a projection device also based on rotating slotted disks. Then, in 1891, the seed of a revolution was planted: Thomas Edison invented the motion picture projector, giving birth to a new industry.

### 1.2.2 The Early Days of "Conventional" Animation

Animation in America exploded in the twentieth century in the form of filming hand-drawn, two-dimensional images (also referred to as *conventional animation*). Studying the early days of conventional animation is interesting in itself [12] [15] [18] [19], but the purpose of this overview is to provide an appreciation of the technological advances that drove the progress of animation during the early years. The earliest use of a camera to make lifeless things appear to move occurred in 1896. Georges Méliès used simple camera tricks such as multiple exposures and stop-motion techniques to make objects appear, disappear, and change shape [7] [21]. Some of the earliest pioneers in film animation were Emile Cohl, a Frenchman who produced several vignettes; J. Stuart Blackton, an American who actually animated "smoke" in a scene in 1900 (special effects) and is credited with creating the first animated cartoon, in 1906; and the American Winsor McCay, the first celebrated animator, best known for his works *Little Nemo* (1911) and *Gertie the Dinosaur* (1914). McCay is considered by many to have produced the first popular animations.

Like many of the early animators, McCay was an accomplished newspaper cartoonist. He redrew each complete image on rice paper mounted on cardboard and then filmed them individually. He was also the first to experiment with color in animation. Much of his early work was incorporated into vaudeville acts in which he "interacted" with an animated character on a screen. Similarly, early cartoons often incorporated live action with animated characters. To appreciate the impact of such a popular entertainment format, we should keep in mind the relative naïveté of audiences at the time; they had no idea how film worked, much less what hand-drawn animation was. It was, indeed, magic.

The first major technical developments in the animation process can be traced to the efforts of John Bray, one of the first to recognize that patenting aspects of the animation process would result in a competitive advantage. Starting in 1910, his work laid the foundation for conventional animation as it exists today. Earl Hurd, who joined forces with Bray in 1914, patented the use of translucent *cels*[4] in the compositing of multiple layers of drawings into a final image and also patented gray scale drawings as opposed to black-and-white. Later developments by Bray and others enhanced the overlay idea to include a peg system for registration and the drawing of the background on long sheets of paper so that *panning* (translating the camera parallel to the plane of the background) could be performed more easily. Out of Bray's studio came Max Fleischer (Betty Boop), Paul Terry (Terrytoons), George Stallings (Tom and Jerry), and Walter Lantz (Woody Woodpecker). In 1915 Fleischer patented *rotoscoping* (drawing images on cells by tracing over previously recorded live action). Several years later, in 1920, Bray experimented with color in the short *The Debut of Thomas Cat*.

While the technology was advancing, animation as an art form was still struggling. The first animated character with an identifiable personality was Felix the Cat, drawn by Otto Messmer of Pat Sullivan's studio. Felix was the most popular and most financially successful cartoon of the mid-1920s. In the late 1920s, however, new forces had to be reckoned with: sound and Walt Disney.

### 1.2.3 Disney

Walt Disney was, of course, the overpowering force in the history of conventional animation. Not only did his studio contribute several technical innovations, but Disney, more than anyone else, advanced animation as an art form. Disney's innovations in animation technology included the use of a storyboard to review the story and pencil sketches to review motion. In addition, he pioneered sound and color in animation (although he was not the first to use color). Disney also studied live-action sequences to create more realistic motion in his films. When he used sound for the first time in *Steamboat Willie* (1928), he gained an advantage over his competitors.

One of the most significant technical innovations of the Disney studio was development of the multiplane camera (see Figure 1.3). The multiplane camera consists of a camera mounted above multiple planes, each of which holds an animation cell. Each of the planes can move in six directions (right, left, up, down, in, out), and the camera can move closer and farther away (see Figure 1.4).

Multiplane camera animation is more powerful than one might think. By moving the camera closer to the planes while the planes are used to move foreground images out to the sides, a more effective zoom can be performed. Moving multiple

[4] *Cel* is short for *celluloid,* which was the original material used in making the translucent layers. Currently, cels are made from acetate.

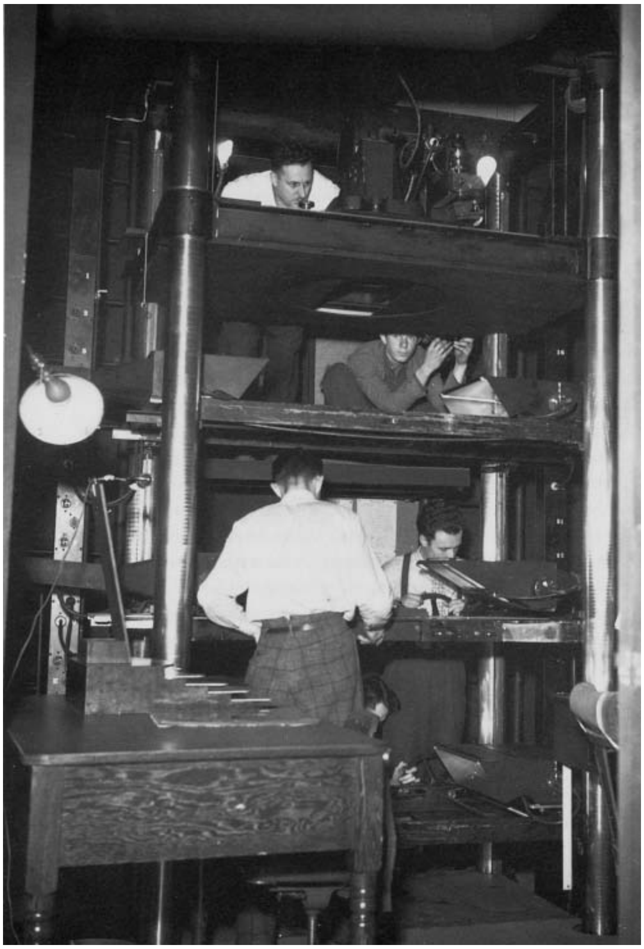


**Figure 1.3** Disney multiplane camera

planes at different rates can produce the *parallax effect,* which is the visual effect of closer objects apparently moving faster across the field of view than objects farther away as an observer's view pans across an environment. This is very effective in creating the illusion of depth and an enhanced sensation of three dimensions. Keeping the camera lens open during movement can produce several additional effects: figures can be extruded into shapes of higher dimension; depth cues can be incorporated into an image by blurring the figures on more distant cels; and motion blur can be produced**.**

With regard to the art of animation, Disney perfected the ability to impart unique, endearing personalities in his characters, such as those exemplified in

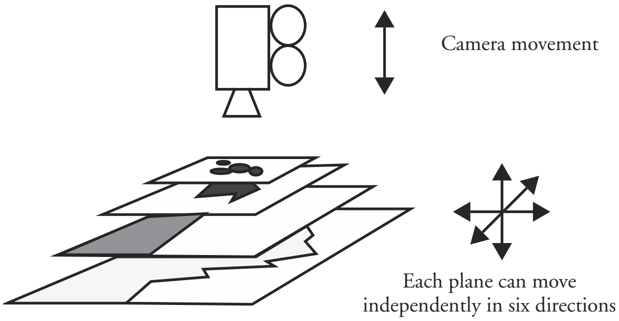


**Figure 1.4** Directional range of the multiplane camera, inside of which the image is optically composited.

Mickey Mouse, Pluto, Goofy, the Three Little Pigs, and the Seven Dwarfs. He promoted the idea that the mind of the character was the driving force of the action and that a key to believable animated motion was the analysis of real-life motion. He also developed mood pieces, for example, *Skeleton Dance* (1929) and *Fantasia* (1940).

## 1.2.4 Contributions of Others

The 1930s saw the proliferation of animation studios, among them Fleischer, Iwerks, Van Beuren, Universal Pictures, Paramount, MGM, and Warner Brothers. The technological advances that are of concern here were mostly complete by this period. The differences between and contributions of the various studios have to do more with the artistic aspects of animation than with the technology involved in producing animation. Many of the notable animators in these studios were graduates of Disney's or Bray's studio. Among the most recognizable names are Ub Iwerks, George Stallings, Max Fleischer, Bill Nolan, Chuck Jones, Paul Terry, and Walter Lantz.

## 1.2.5 Other Media for Animation

The rich heritage of hand-drawn animation in the United States makes it natural to consider it the precursor to computer animation, which also has strong roots in this country. However, computer animation has a close relationship to other animation techniques as well. A close comparison can be made between computer animation and some stop-motion techniques, such as clay and puppet animation. Typically in three-dimensional computer animation, the first step is the object modeling process. The models are then manipulated to create the three-dimensional scenes that are rendered to produce the images of the animation.

In much the same way, clay and puppet stop-motion animation use three-dimensional figures that are built and then animated in separate, well-defined stages. Once the physical, three-dimensional figures are created, they are used to lay out a three-dimensional environment. A camera is positioned to view the environment and record an image. One or more of the figures are manipulated, and the camera may be repositioned. The camera records another image of the scene. The figures are manipulated again, another image is taken of the scene, and the process is repeated to produce the animated sequence. Willis O'Brien of *King Kong* fame is generally considered the dean of this type of stop-motion animation. His understudy, who went on to create an impressive body of work in his own right, was Ray Harryhausen (*Mighty Joe Young, Jason and the Argonauts,* and many more). More recent impressive examples of 3D stop-motion animation are Nick Park's *Wallace and Gromit* series and the Tim Burton productions *Nightmare Before Christmas* and *James and the Giant Peach.*

Because of computer animation's close association with video technology, it has also been associated with video art, which depends largely on the analog manipulation of the video signal. Because creating video art is inherently a two-dimensional process, the relationship is viewed mainly in the context of computer animation postproduction techniques. Even this connection has faded since the popularity of recording computer animation by digital means has eliminated most analog processing.

### 1.2.6 Principles of Computer Animation

To study various techniques and algorithms used in computer animation, it is useful to first understand their relationship to the animation principles used in hand-drawn animation. In an article by Lasseter [8], the principles of animation as articulated by some of the original Disney animators [19] are related to techniques commonly used in computer animation. These principles are *squash & stretch*, *timing, secondary actions*, *slow in & slow out, arcs, follow through/overlapping action*, *exaggeration*, *appeal, anticipation, staging,* and *straight ahead* versus *pose to pose.* Lasseter is a conventionally trained animator who worked at Disney before going to Pixar. At Pixar he was responsible for many celebrated computer animations, including the first to win an Academy Award, *Tin Toy*. Whereas Lasseter discusses each principle in terms of how it might be implemented using computer animation techniques, the principles are organized here according to the type of issue they contribute to in a significant way. Because several principles relate to multiple issues, some principles appear under more than one heading.

### Simulating Physics

Squash & stretch, timing, secondary actions, slow in & slow out, and arcs establish the physical basis of objects in the scene. A given object possesses some degree of rigidity and should appear to have some amount of mass. This is reflected in the distortion (*squash & stretch*) of its shape during an action, especially a collision. The animation must support these notions consistently for a given object throughout the animation. *Timing* has to do with how actions are spaced according to the weight, size, and personality of an object or character and, in part, with the physics of movement as well as the artistic aspects of the animation. *Secondary actions* support the main action, possibly supplying physically based reactions to an action that just occurred. *Slow in & slow out* and *arcs* are concerned with how things move through space. Objects *slow in* and *slow out* of poses. When speaking of the actions involved, objects are said to ease in and ease out. Such speed variations model inertia, friction, and viscosity. Objects, because of the physical laws of nature such as gravity, usually move not in straight lines but rather in arcs.

### Designing Aesthetic Actions

*Exaggeration, appeal*, and *follow through/overlapping action* are principles that address the aesthetic design of an action or action sequence. Often the animator needs to exaggerate a motion so it cannot be missed or so it makes a point (Tex Avery is well known for this type of conventional animation). To keep the audience's attention, the animator needs to make it enjoyable to watch (*appeal*). In addition, actions should flow into one another (*follow through/overlapping action*) to make the entire shot appear to continually evolve instead of looking like disjointed movements. Squash & stretch can be used to exaggerate motion and to create flowing action. Secondary actions and timing considerations also play a role in designing motion.

### Effective Presentation of Actions

*Anticipation* and *staging* concern how an action is presented to the audience. *Anticipation* dictates that an upcoming action is set up so that the audience knows it (or something) is coming. *Staging* expands on this notion of presenting an action so that it is not missed by the audience. Timing is also involved in effective presentation to the extent that an action has to be given the appropriate duration for the intended effect to reach the audience. *S*econdary actions and exaggeration can also be used to create an effective presentation of an action.

### Production Technique

*Straight ahead* versus *pose to pose* concerns how a motion is created. *Straight ahead* refers to progressing from a starting point and developing the motion continually along the way. Physically based animation could be considered a form of this. *Pose* *to pose,* the typical approach in conventional animation, refers to identifying key frames and then interpolating intermediate frames.

## 1.3 Animation Production

Although animation production is not the subject of this book, it merits some discussion. It is useful to have some familiarity with how a piece of animation is broken into parts and how animators go about producing a finished piece. Much of this is taken directly from conventional animation and is applicable to computer animation.

A piece of animation is usually discussed using a four-level hierarchy, although the specific naming convention for the levels may vary.[5] Here the overall animation—the entire project—is referred to as the *production.* Typically, productions are broken into major parts referred to as sequences. A sequence is a major episode and is usually identified by an associated staging area; a production usually consists of one to a dozen sequences. A sequence is broken down into one or more shots; each shot is a continuous camera recording. A shot is broken down into the individual frames of film. A *frame* is a single recorded image. This results in the hierarchy shown in Figure 1.5.

Several steps are required to successfully plan and carry out the production of a piece of animation [9] [18]. Animation is a trial-and-error process that involves feedback from one step to previous steps and usually demands several iterations through multiple steps at various times. Even so, the production of animation typically follows a standard pattern. First, a *preliminary story* is decided on, including a *script.* A *storyboard* is developed that lays out the action scenes by sketching representative frames. The frames are often accompanied by text that sketches out the

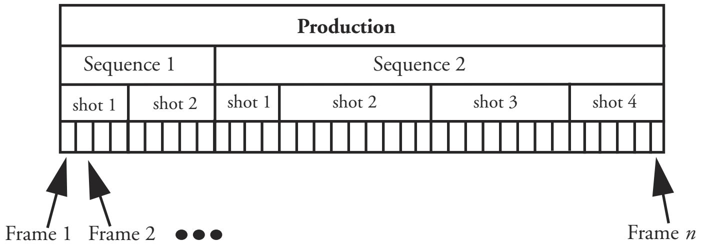


**Figure 1.5** Sample hierarchy of a simple animation production

[5] Live-action film tends to use a five-level hierarchy: film, sequence, scene, shot, frame [3]. Here terminology used in feature-length computer animation is presented.

action taking place. This is used to present, review, and critique the action as well as to examine character development. A *model sheet* is developed that consists of a number of drawings for each figure in various poses and is used to ensure that each figure's appearance is consistent as it is repeatedly drawn during the animation process. The *exposure sheet* records information for each frame such as sound track cues, camera moves, and compositing elements. The *route sheet* records the statistics and responsibility for each scene. An *animatic,* or *story reel,* may be produced in which the storyboard frames are recorded, each for as long as the sequence it represents, thus creating a rough review of the timing. Once the storyboard has been decided on, the *detailed story* is worked out to identify the actions in more detail. *Key frames* (also known as *extremes*) are then identified and produced by master animators to aid in confirmation of character development and image quality. Associate and assistant animators are responsible for producing the frames between the keys; this is called *in-betweening. Test shots,* short sequences rendered in full color, are used to test the rendering and motions. To completely check the motion, a *pencil test* may be shot, which is a full-motion rendering of an extended sequence using low-quality images such as pencil sketches. Problems identified in the test shots and pencil tests may require reworking of the key frames, detailed story, or even the storyboard. *Inking* refers to the process of transferring the penciled frames to cels. *Opaquing,* also called *painting,* is the application of color to these cels.

Sound is very important in animation because of the higher level of precise timing that is possible when compared to live action [9]. Sound takes three forms in an animation: music, special effects, and voice. Whether the sound or the animation should be created first depends on the production and the role that sound plays. For lip-synched animation, the sound track should be created first and the animation made to fit. In most other cases, the animation can take place first, with the sound created to fit the action. In such cases, a *scratch track,* or rough sound track, is built at the same time the storyboard is being developed and is included in the animatic.

Computer animation production has borrowed most of the ideas from conventional animation production, including the use of a storyboard, test shots, and pencil testing. The storyboard has translated directly to computer animation production, although it may be done on-line. It still holds the same functional place in the animation process and is an important component in planning animation. The use of key frames and in-betweens has also been adopted in certain computer animation systems.

While computer animation has borrowed the production approaches of conventional animation, there are significant differences between how computer animation and conventional animation create an individual frame of the animation. In computer animation, there is usually a strict distinction among creating the models; creating a layout of the models including camera positioning and lighting; specifying the motion of the models, lights, and camera; and the rendering process applied to those models. This allows for reusing models and lighting setups. In conventional animation, all of these processes happen simultaneously as each drawing is created, the only exception being the possible reuse of backgrounds, for example, with the multilayer approach.

The two main evaluation tools of conventional animation, test shots and pencil tests, have counterparts in computer animation. A speed/quality trade-off can be made in each of the three stages of creating a frame of computer animation: model building, motion control, and rendering. By using high-quality techniques in only one or two of these stages, that aspect of the presentation can be quickly checked in a cost-effective manner. A test shot in computer animation is produced by a high-quality rendering of a highly detailed model to see a single frame, a short sequence of frames of the final product, or every *n*th frame of a longer sequence from the final animation. The equivalent of a pencil test can be performed by simplifying the sophistication of the models used, by using low-quality and/or low-resolution renderings, or by using simplified motion.

Often, it is useful to have several representations of each model available at varying levels of detail. For example, placeholder cubes can be rendered to present the gross motion of rigid bodies in space and to see spatial and temporal relationships among objects. “Solids of revolution” objects (objects created by rotating a silhouette edge around an axis) lend themselves quite well to multiple levels of detail for a given model based on the number of slices used. Texture maps and displacement maps can be disabled until the final renderings.

To simplify motion, articulated figures[6] can be kept in key poses as they navigate through an environment in order to avoid interpolation or inverse kinematics. Collision detection and response can be selectively “turned off” when not central to the effect created by the sequence. Complex effects such as smoke and water can be removed or represented by simple geometric shapes during testing.

Many aspects of the rendering can be selectively turned on or off to provide great flexibility in giving the animator clues to the finished product’s quality without committing to the full computations required in the final presentation. Real-time rendering can be used to preview the animation. Wire frame rendering of objects can sometimes provide sufficient visual cues to be used in testing. Often, the resulting animation can be computed in real time for very effective motion testing before committing to a full anti-aliased, transparent, texture-mapped rendering. Shadows, smooth shading, texture maps, environmental maps, specular reflection, and solid texturing are options the animator can use for a given run of

[6] *Articulated figures* are models consisting of rigid segments usually connected in a treelike structure; the connections are either revolute or prismatic joints, allowing a segment to rotate or translate relative to the connected segment.

the rendering program. Even in finished pieces of commercial animation it is common practice to take computational shortcuts when they do not affect the quality of the final product. For example, the animator can select which objects can shadow which other objects in the scene. In addition to being a compositional issue, selective shadowing saves time over a more robust approach in which every object can shadow every other object. In animation, environmental mapping is commonly used instead of ray tracing. Radiosity is typically avoided.

Computer animation is well suited for producing the equivalent of test shots and pencil tests. In fact, because the quality of the separate stages of computer animation can be independently controlled, it may be even better suited for these evaluation techniques than conventional animation.

### 1.3.1 Computer Animation Production Tasks

While motion control is the primary subject of this book, it is worth noting that motion control is only one aspect of the effort required to produce computer animation. The other tasks (and the other talents) that are integral to the final product should not be overlooked. As previously mentioned, producing quality animation is a trial-and-error, iterative process wherein performing one task may require rethinking one or more previously completed tasks. Even so, these tasks can be laid out in an approximate chronological order according to the way they are typically encountered. Any proposed ordering is subject to debate. The one presented here summarizes an article that describes the system used to produce Pixar's *Toy Story* [5]. See Figure 1.6.

- The Story Department translates the verbal into the visual. The screenplay enters the Story Department, the storyboard is developed, and the story reel leaves. It goes to the Art Department.
- The Art Department, working from the storyboard, creates the designs and color studies for the film, including detailed model descriptions and lighting scenarios.The Art Department develops a consistent look to be used in the imagery. This look guides the Modeling, Layout, and Shading Departments.
- The Modeling Department creates the characters and the world in which they live. Every brick and stick to appear in the film must be handcrafted. Often, articulated figures or other models with inherent movements are created as parameterized models. Parameters are defined that control possible articulations or other movements for the figure. This facilitates the ability of animators to *stay on the model,* ensuring that the animation remains consistent with the concept of the model. The models are given to Layout and Shading.

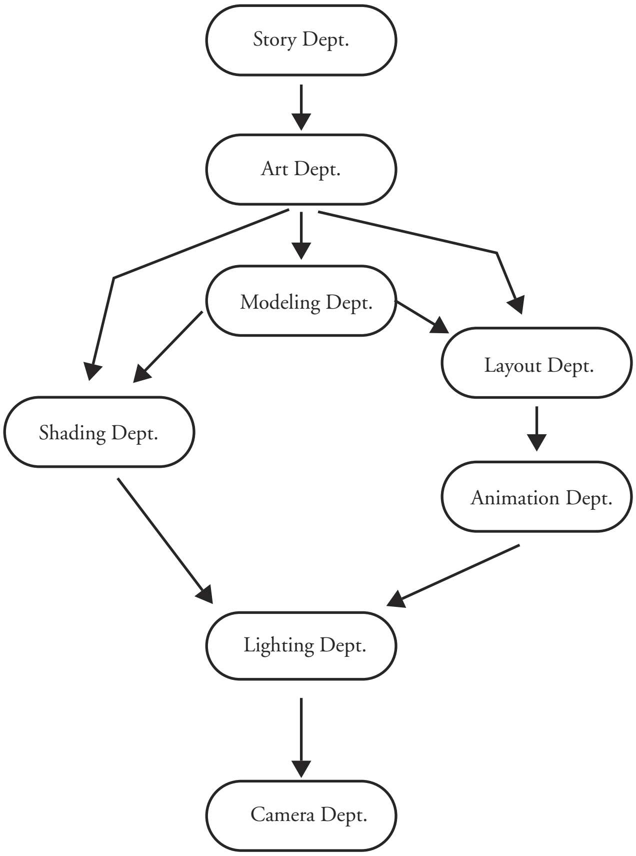


**Figure 1.6** Computer animation production pipeline

- The Layout Department is responsible for taking the film from two dimensions to three dimensions. To ensure good flow, Layout implements proper staging and blocking. This guides the Animation Department.
- On one path between the Modeling Department and Lighting Department lies Shading. The Shading Department must translate the attributes of the object that relate to its visual appearance into texture maps, displacement shaders, and lighting models. Relevant attributes include the material the object is made of, its age, and its condition. Much of the effective appearance of an object comes not from its shape but from the visual qualities of its surface.
- On another path between Modeling and Lighting lies Layout, followed by Animation. Working from audio, the story, and the blocking and staging produced by Layout, the Animation Department is responsible for bringing

the characters to life. In the case of parameterized models, basic movements have already been defined. The Animation Department creates the subtler gestures and movements necessary for the "actor" to effectively carry out the scene.

- The Lighting Department assigns to each sequence teams that have responsibility for translating the Art Department's vision into digital reality. At this point the animation and camera placement have been done. Key lights are set to establish the basic lighting environment. Subtler lighting particular to an individual shot refines this in order to establish the correct mood and bring focus to the action.
- The Camera Department is responsible for actually rendering the frames. During *Toy Story,* Pixar used a dedicated array of hundreds of processors called the *RenderFarm*.

## 1.3.2 Digital Editing

A revolution has swept the film and video industries in recent years: the digital representation of images. Even if computer graphics and digital effects are not a consideration in the production process, it is becoming commonplace to store program elements in digital form instead of using the analog film and videotape formats. Digital representations have the advantage of being able to be copied with no image degradation. So even if the material was originally recorded using analog means, it has recently become cost-effective to transcribe the images to digital image store. And, of course, once the material is in digital form, digital manipulation of the images is a natural capability to incorporate in any system.

### In the Old Days . . .

The most useful and fundamental digital image manipulation capability is that of editing sequences of images together to create a new presentation. Originally, film sequences were edited together by physically cutting and splicing tape. This is an example of *nonlinear editing,* in which sequences can be inserted in any order at any time to assemble the final presentation. However, splicing is a time-consuming process, and making changes in the presentation or trying different alternatives can place a heavy burden on the stock material as well.

Electronic editing[7] allows one to manipulate images as electronic signals rather than use a physical process. The standard configuration uses two source videotape players, a switching box, and an output videotape recorder (see Figure 1.7). The

[7] To simplify the discussion and make it more relevant to the capabilities of the personal computer, the discussion here focuses on video editing, although much of it is directly applicable to digital film editing, except that film standards require much higher resolution and therefore more expensive equipment.

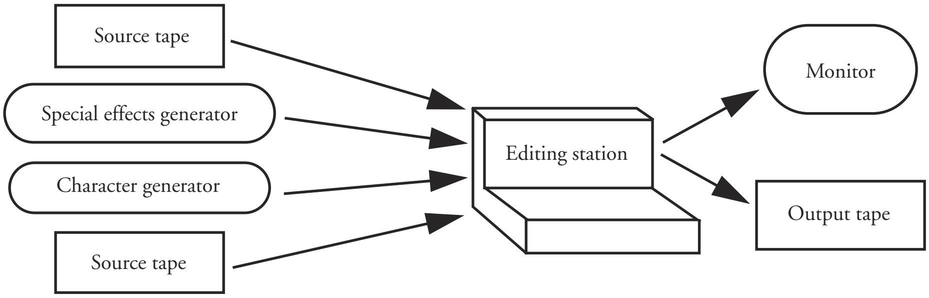


**Figure 1.7** Linear editing system

two source tapes are searched to locate the initial desired sequence; the tape deck on which it is found is selected for recording on the output deck, and the sequence is recorded. The tapes are then searched to locate the next segment, the deck is selected for output, and it is recorded on the output tape. This continues until the new composite sequence has been created on the output tape. The use of two source tapes allows multiple sources to be more easily integrated into the final program. Other analog effects can be incorporated into the output sequence at the switcher by using a *character generator* (text overlays) and/or a *special effects generator* (wipes, fades, etc.). Because the output is assembled in a sequence order, this is referred to as *linear editing*. One of the drawbacks of this approach is that the output material has to be assembled in a linear fashion. Electronic editing also has the drawback that the material is copied in the editing process, introducing some image degradation. Because the output tape is commonly used to master the tapes that are sent out to be viewed, these tapes are already third generation. Another drawback is the amount of wear on the source material as the source tapes are repeatedly played and rewound as the next desired sequence is searched for. If different output versions are required (called *versioning*), the source material will be subject to even more wear and tear because the source material is being used for multiple purposes.

Often, to facilitate the final assemblage of the output sequence and avoid excessive wear of the original source material, copies of the source material are used in a preprocessing stage in which the final edits are determined. This is called *off-line editing*. The result of this stage is an *EDL* (edit decision list), which is a final list of the edits that need to be made to assemble the final piece. The EDL is then passed to the *on-line editing* stage, which uses the original source material to make the edits and create the finished piece. This process is referred to as *conforming*.

To keep track of edit locations, control track pulses can be incorporated onto the tape used to assemble the thirty-frame-per-second NTSC video signal. Simple editing systems count the pulses; this is called *control track editing*. However, the continual shuffling of the tape back and forth during the play and rewind of the editing process can result in the editing unit losing count of the pulses. This is something the operator must be aware of and take into account. In addition, because the edit counts are relative to the current tape location, the edit locations are lost when the editing station is turned off.

The Society of Motion Picture and Television Engineers (SMPTE) time code is an absolute eight-digit tag on each frame in the form of WWXXYYZZ, where WW is the hour, XX is the minute, YY is the second, and ZZ is the frame number. This tag is calculated from the beginning of the sequence. This allows an editing station to record the absolute frame number for an edit and then store the edit location in a file that can be retrieved for later use.

The process described so far is *assemble editing*. *Insert editing* is possible if a control signal is first laid down on the output tape. Then sequences can be inserted anywhere on the tape in forming the final sequence. This provides some nonlinear editing capability, but it is still not possible to easily lengthen or shorten a sequence without repositioning other sequences on the tape to compensate for the change.

### Digital On-line Nonlinear Editing

To incorporate a more flexible nonlinear approach, fully digital editing systems have become more accessible [6] [13] [22]. These can be systems dedicated to editing, or they can be software systems that run on standard computers. Analog tape may still be used as the source material and for the final product, but everything in between is digitally represented and controlled (see Figure 1.8).

After a sequence has been digitized, an icon representing it can be dragged onto a time line provided by the editing system. Sequences can be placed relative to one another; they can be repeated, cut short, overlapped with other sequences, combined with transition effects, and mixed with other effects. A simplified example of such a time line is shown in Figure 1.9.

The positioning of the elements in the time line is conceptual only; typically the data in the digital image store is not actually copied or moved. The output sequence can be played back in real time if the disk random access and graphics display are fast enough to fetch and compile the separate tracks on the fly. In the case of overlapping sequences with transitions, either the digital store must support the access of multiple tracks simultaneously so a transition can be constructed on the fly or the transition sequence needs to be precomputed and explicitly stored for access during playback. When the sequence is finalized it can be assembled and stored digitally or recorded on video. Whatever the case, the flexibility of this approach and the ability to change edits and try alternatives make nonlinear digital editing systems very powerful.

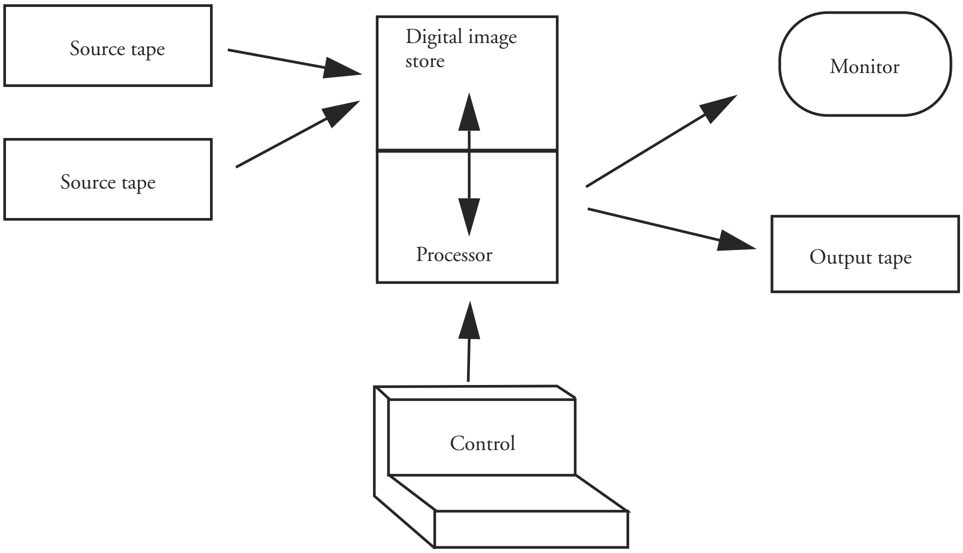


**Figure 1.8** On-line nonlinear editing system

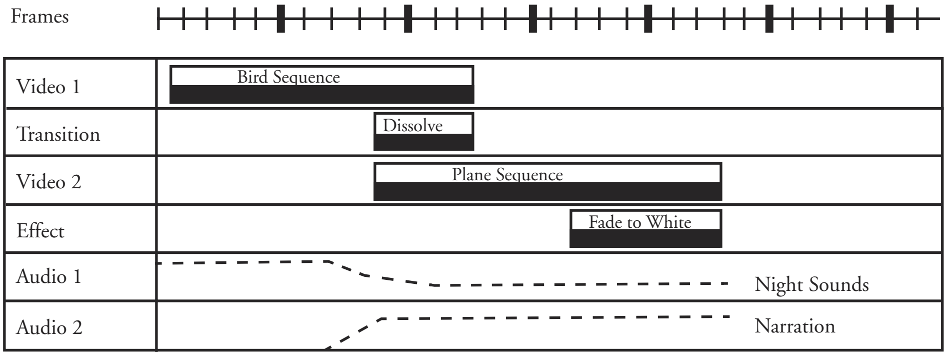


**Figure 1.9** Simplified example of a time line used for nonlinear digital editing

## 1.3.3 Digital Video

As the cost of computer memory decreases and processor speeds increase, the capture, compression, storage, and playback of digital video have become more prevalent [17] [20]. This has several important ramifications. First, desktop animation has become inexpensive enough to be within the reach of the consumer. Second, in the film industry it has meant that compositing is no longer optical. Optically compositing each element in a film meant another pass of the negative through an optical film printer, which meant degraded quality. With the advent of digital compositing (see Appendix A), the limit on the number of composited elements is removed. Third, once films are routinely stored digitally, digital techniques can be used for wire removal and to apply special effects. These digital techniques have become the bread and butter of computer graphics in the film industry.

When one works with digital video, there are several issues that need to be addressed to determine the cost, speed, storage requirements, and overall quality of the resulting system. Compression techniques can be used to conserve space, but some compression compromises the quality of the image and the speed of compression/decompression may restrict a particular technique's suitability for a given application. During video capture, any image compression must operate in real time. Formats used for storage and playback can be encoded off-line, but the decoding must support real-time playback. Video resolution, video frame rates, and full-color imagery require that 21 Mb/sec be supported for video playback. An hour of uncompressed video requires 76 Gb of storage. There are several digital video formats used by different manufacturers of video equipment for various applications; these formats include D1, D2, D3, D5, miniDV, DVC, Digital8, MPEG, and digital Betacam. Better signal quality can be attained with the use of component instead of composite signals. Discussion of these and other issues related to digital video is beyond the scope of this book. Information on some of the more popular formats can be found in Appendix B.

## 1.4 A Brief History of Computer Animation

### 1.4.1 Early Activity

The earliest computer animation of the late 1960s and early 1970s was produced by a mix of researchers in university labs and individual visionary artists [10] [11] [14]. At the time, raster displays driven by frame buffers were just being developed and digital output to television was still in the experimental stage. The displays in use were primarily storage tubes and refresh vector displays. Storage tubes retain an image indefinitely because of internal circuitry that continuously refreshes the display. However, because the image cannot be easily modified, storage tubes were used mainly to draw complex static models. Vector (calligraphic) displays use a display list of line- and arc-drawing instructions that an internal processor uses to repeatedly draw an image that would otherwise quickly fade on the screen. Vector displays can draw moving images by carefully changing the display list between refreshes. These were popular for interactive design. Static images were often recorded onto film by placing a camera in front of the display and taking a picture of the screen. Shaded images could be produced by opening the shutter of the film camera and essentially scan converting the elements (e.g., polygons) by drawing closely spaced horizontal vectors to fill the figure; after scan conversion was completed, the shutter was closed to terminate the image recording. The intensity of the image could be regulated by using the intensity control of the vector display or by controlling other aspects of the image recording such as by varying the density of the scanlines. An image of a single color was generated by placing a colored filter in front of the camera lens. A full-color image could be produced by breaking the image into its red, green, and blue components and triple exposing the film with each exposure using the corresponding colored filter. This same approach could be used to produce animation as long as the motion camera was capable of single-frame recording. This required precise frame registration, usually available only in expensive equipment. Animated sequences could be colored by triple exposing the entire film. A programmer or animator was fortunate if both the camera and the filters could be controlled by computer.

The earliest research in computer graphics and animation occurred at MIT in 1963 when Ivan Sutherland developed an interactive constraint satisfaction system on a vector refresh display. The user could construct an assembly of lines by specifying constraints between the various graphical elements. If one of the graphical elements moved, the system calculated the reaction of other elements to this manipulation based on satisfying the specified constraints. By interactively manipulating one of the graphical elements, the user could produce complex motion in the rest of the assembly. Later, at the University of Utah, Sutherland helped David Evans establish the first significant research program in computer graphics and animation.

In the early 1970s, computer animation in university research labs became more widespread. Computer graphics, as well as computer animation, received an important impetus through government funding at the University of Utah. As a result, Utah produced several groundbreaking works in animation: an animated hand and face by Ed Catmull (*Hand/Face,* 1972); a walking and talking human figure by Barry Wessler (*Not Just Reality,* 1973); and a talking face by Fred Parke (*Talking Face,* 1974). Although the imagery was extremely primitive by today's standards, the presentations of lip-synched facial animation and linked figure animation were impressive demonstrations well ahead of their time.

In the mid-1970s, Norm Badler at the University of Pennsylvania conducted investigations into posing a human figure. He developed a constraint system to move the figure from one pose to another. He has continued this research and established the Center for Human Modeling and Simulation at Penn. *Jack* is a software package developed at the center that supports the positioning and animation of anthropometrically valid human figures in a virtual world.

At Ohio State University in the 1970s, the Computer Graphics Research Group (CGRG), founded by the artist Chuck Csuri, produced animations using a real-time video playback system developed at North Carolina State University under the direction of John Staudhammer. Software developed at CGRG compressed frames of animation and stored them to disk. During playback, the compressed digital frames were retrieved from the disk and piped to the special-purpose hardware, which took the digital information, decompressed it on the fly, and converted it into a video signal for display on a standard television. The animation was driven by the ANIMA II language [4].

In the late 1970s, the New York Institute of Technology (NYIT) produced several computer animation systems thanks to individuals such as Ed Catmull and Alvy Ray Smith. At the end of the 1970s, NYIT embarked on an ambitious project to produce a wholly computer generated feature film using three-dimensional computer animation, titled *The Works*. While the project was never completed, excerpts were shown at several SIGGRAPH conferences as progress was made. The excerpts demonstrated high-quality rendering, articulated figures, and interacting objects. The system used at NYIT was BBOP, a three-dimensional key-frame articulated figure animation system [16].

Early artistic animators in this period included Ken Knowlton, Lillian Schwartz, S. Van Der Beek, John Whitney, Sr., and A. M. Noll. Typical artistic animations consisted of animated abstract line drawings displayed on vector refresh displays. Using the early technology, Chuck Csuri produced pieces such as *Hummingbird* (1967) that were more representational.

In 1974, the first computer animation nominated for an Academy Award, *Hunger*, was produced by Rene Jodoin; it was directed and animated by Peter Foldes. This piece used a 2½D system that depended heavily on object shape modification and line interpolation techniques. The system was developed by Nestor Burtnyk and Marceli Wein at the National Research Council of Canada in conjunction with the National Film Board of Canada. *Hunger* was the first animated story using computer animation.

In the early 1980s Daniel Thalmann and Nadia Magnenat-Thalmann started work in computer animation at the University of Montreal. Over the years, their labs have produced several impressive animations, including *Dream Flight* (N. Magnenat-Thalmann and D. Thalmann, 1982), *Tony de Peltrie* (P. Bergeron, 1985), and *Rendez-vous à Montréal* (N. Magnenat-Thalmann and D. Thalmann, 1988).

Others who advanced computer animation during the period were Ed Emshwiller at NYIT, who demonstrated moving texture maps in *Sunstone* (1979); Jim Blinn, who produced the *Voyager* flyby animations at the Jet Propulsion Laboratory (1979); Don Greenberg, who used architectural walk-throughs of the Cornell University campus (1971); and Nelson Max at Lawrence Livermore Laboratory, who animated space-filling curves (1978).

The mid- to late 1970s saw the first serious hints of commercial computer animation. Tom DeFanti developed the Graphical Symbiosis System (GRASS) at Ohio State University (1976), a derivative of which was used in the computer graphics sequences of the first *Star Wars* film (1977). In addition to *Star Wars,* films such as *Future World* (1976) and *Looker* (1981) began to incorporate simple computer animation as examples of advanced technology. This was an exciting time for those in the research labs wondering if computer animation would ever see the light of day. One of the earliest companies to use computer animation was the Mathematical Application Group Inc. (MAGI), which used a ray-casting algorithm to provide scientific visualizations. MAGI also adapted its technique to produce early commercials for television.

### 1.4.2 The Middle Years

The 1980s saw a more serious move by entrepreneurs into commercial animation. Computer hardware advanced significantly with the introduction of the VAX computer in the 1970s and the IBM PC at the beginning of the 1980s. Hardware z-buffers were produced by companies such as Raster Tech and Ikonas; Silicon Graphics was formed; and flight simulators based on digital technology were taking off because of efforts by the Evans and Sutherland Corporation. These hardware developments were making the promise of cost-effective computer animation to venture capitalists. At the same time, graphics software was getting more sophisticated: Turner Whitted introduced anti-aliased ray tracing (*The Compleat Angler,* 1980); Nelson Max produced several films about molecules as well as one of the first films animating waves (*Carla's Island,* 1981); and Loren Carpenter produced a flyby of fractal terrain (*Vol Libre*, 1982). Companies such as Alias, Wavefront, and TDI were starting to produce sophisticated software tools making advanced rendering and animation available off-the-shelf for the first time.

Animation houses specializing in 3D computer animation started to appear. Television commercials, initially in the form of flying logos, provided a profitable area where companies could hone their skills. Demo reels appeared at SIGGRAPH produced by the first wave of computer graphics companies such as Information International Inc. (Triple-I), Digital Effects, MAGI, Robert Abel and Associates, and Real Time Design (ZGRASS). The first four of these companies combined to produce the digital imagery in Disney's *TRON* (1982), which was a landmark movie in its (relatively) extensive use of a computer-generated environment in which graphical objects were animated. Previously, the predominant use of computer graphics in movies had been to show a monitor (or simulated projection) of something that was supposed to be a computer graphics display (*Futureworld,* 1976; *Star Wars,* 1977; *Looker,* 1981). Still, in *TRON,* the computer-generated imagery was not meant to simulate reality; the action takes place inside a computer, so a computer-generated look was consistent with the story line. At the same time that computer graphics was starting to find its way into the movies it was becoming a more popular tool for generating television commercials. As a result, more computer graphics companies surfaced, including Digital Pictures, Image West, Cranston-Csuri, Pacific Data Images, Lucasfilm, Marks and Marks, Digital Productions, and Omnibus Computer Graphics.

Most early use of synthetic imagery in movies was incorporated with the intent that it would appear as if computer generated. The other use of computer animation during this period was to do animation. That is, the animated pieces were meant not to fool the eye into thinking that what was being seen was real but rather to replace the look and feel of 2D conventional animation with that of 3D computer animation. Of special note are the award-winning animations produced by Lucasfilm and Pixar:

*The Adventures of Andre and Wally B.* (1984)—first computer animation demonstrating motion blur

*Luxo Jr.* (1986)—nominated for an Academy Award

*Red's Dream* (1987)

*Tin Toy* (1988)—first computer animation to win an Academy Award

*Knick Knack* (1989)

*Geri's Game* (1999)—Academy Award winner

These early animations paved the way for 3D computer animation to be accepted as an art form. They were among the first fully computer generated three-dimensional animations to be taken seriously as animations, irrespective of the technique involved. Another early piece of 3D animation, which integrated computer graphics with conventional animation, was *Technological Threat* (1988). This was one of three films nominated for an Academy Award as an animated short in 1988; *Tin Toy* came out the victor.

One of the early uses of computer graphics in film was to model and animate spacecraft. Working in (virtual) outer space with spacecraft has the advantages of simple illumination models, a relatively bare environment, and relatively simple animation of rigid bodies. In addition, spacecraft are usually modeled by relatively simple geometry—as is the surrounding environment (planets)—when in flight. *The Last Starfighter* (1984, Digital Productions) used computer animation instead of building models for special effects; the computer used, the Cray X-MP, even appeared in the movie credits. The action takes place in space as well as on planets; computer graphics was used for the scenes in space, and physical models were used for the scenes on a planet. Approximately twenty minutes of computer graphics are used in the movie. While it is not hard to tell when the movie switches between graphical and physical models, this was the first time graphics was used as an extensive part of a live-action film in which the graphics were supposed to look realistic.

### Animation Comes of Age

As modeling, rendering, and animation became more sophisticated and the hardware became faster and inexpensive, quality computer graphics began to spread to the Internet, television commercials, computer games, and stand-alone game units. In film, computer graphics helps to bring alien creatures to life. Synthetic alien creatures, while they should appear to be real, do not have to match specific audience expectations. *Young Sherlock Holmes* (1986, ILM) was the first to place a synthetic character in a live-action feature film. An articulated stained glass window comes to life and is made part of the live action. The light sources and movements of the camera in the live action had to be mimicked in the synthetic environment, and images from the live action were made to refract through the synthetic stained glass. In *The Abyss* (1989, ILM), computer graphics is used to create an alien creature that appears to be made from water. Other notable films in which synthetic alien creatures are used are *Terminator II* (1991, ILM), *Casper* (1995, ILM), *Species* (1995, Boss Film Studios), and *Men in Black* (1997, ILM).

A significant advance in the use of computer graphics for the movies came about because of the revolution in cheap digital technology, which allowed film sequences to be stored digitally. Once the film is stored digitally, it is in a form suitable for digital special effects processing, digital compositing, and the addition of synthetic elements. For example, computer graphics can be used to remove the mechanical supports of a prop or to introduce digital explosions or laser blasts. For the most part, this resides in the 2D realm and thus is not the focus of this book. However, with the advent of digital techniques for 2D compositing, sequences are more routinely available in digital representations, making them amenable to a variety of digital postprocessing techniques. The first digital blue screen matte extraction was in *Willow* (1988, ILM). The first digital wire removal was in *Howard the Duck* (1986, ILM). In *True Lies* (1994, Digital Domain), digital techniques erased inserted atmospheric distortion to show engine heat. In *Forrest Gump* (1994, ILM), computer graphics inserted a Ping-Pong ball in a sequence showing an extremely fast action game, inserted a new character into old film footage, and enabled the illusion of a double amputee as played by a completely able actor. In *Babe* (1995, Rhythm & Hues), computer graphics was used to move the mouths of animals and fill in the background uncovered by the movement. In *Interview with a Vampire* (1994, Digital Domain), computer graphics were used to curl the hair of a woman during her transformation into a vampire. In this case, some of the effect was created using 3D graphics, which were then integrated into the scene by 2D techniques.

A popular graphical technique for special effects is the use of particle systems. One of the earliest examples is in *Star Trek II: The Wrath of Khan* (1982, Lucasfilm), in which a wall of fire sweeps over the surface of a planet. Although by today's standards the wall of fire is not very convincing, it was an important step in the use of computer graphics in movies. Particle systems are also used in *Lawnmower Man* (1992, Angel Studios, Xaos), in which a character disintegrates into a swirl of small spheres. The modeling of a comet's tail in the opening sequence of the television series *Star Trek: Deep Space Nine* (1993– ) is a more recent example of a particle system. In a much more ambitious and effective application, *Twister* (1996, ILM) uses particle systems to simulate a tornado.

More challenging is the use of computer graphics to create realistic models of creatures with which the audience is intimately familiar. *Jurassic Park* (1993, ILM) is the first example of a movie that completely integrates computer graphics characters (dinosaurs) of which the audience has fairly specific expectations. Of course, there is still some leeway here, because the audience does not have precise knowledge of how dinosaurs look. *Jumanji* (1995, ILM) takes on the ultimate task of modeling creatures for which the audience has precise expectations: various jungle animals. Most of the action is fast and blurry, so the audience does not have time to dwell on the synthetic creatures visually, but the result is very effective. To a lesser extent, *Batman Returns* (1995, PDI) does the same thing by providing "stunt doubles" of Batman in a few scenes. The scenes are quick and the stunt double is viewed from a distance, but it was the first example of a full computer graphics stunt double in a movie. Computer graphics shows much potential for managing the complexity in crowd scenes. PDI used computer graphics to create large crowds in the *Bud Bowl* commercials of the mid-1980s. In feature films, crowd scenes include the wildebeest scene in *Lion King* (1994, Disney), the alien charge in *Starship Troopers* (1997, Tippet Studio*)*, and (for distant shots) synthetic figures populating the deck of the ship in *Titanic* (1998, ILM).

A holy grail of computer animation is to produce a synthetic human character indistinguishable from a real person. Early examples of animations using "synthetic actors" are *Tony de Peltrie* (1985, P. Bergeron), *Rendez-vous à Montréal* (1988, D. Thalmann), *Sextone for President* (1989, Kleiser-Walziac Construction Company), and *Don't Touch Me* (1989, Kleiser-Walziac Construction Company). However, it is obvious to viewers that these animations are computer generated. Recent advances in illumination models and texturing have produced human figures that are much more realistic and have been incorporated into otherwise live-action films. Synthetic actors have progressed from being distantly viewed stunt doubles and passengers on a boat to assuming central roles in various movies: the dragon in *Dragonheart* (1996, Tippett Studio, ILM); the Jellolike main character in *Flubber* (1997, ILM); the aliens in *Mars Attacks* (1996, ILM); and the ghosts in

*Casper* (1995, ILM). The first fully articulated humanoid synthetic actor integral to a movie was the character Jar-Jar in *Star Wars: Episode I* (1999, ILM).

Of course, one use of computer animation is simply to "do animation"; computer graphics is used to produce animated pieces that are essentially 3D cartoons that would otherwise be done by more traditional means. The animation does not attempt to fool the viewer into thinking anything is real; it is meant simply to entertain. The film *Hunger* falls into this category, as do the Lucasfilm/Pixar animations. *Toy Story* is the first full-length, fully computer generated 3D animated feature film. Recently, other feature-length 3D cartoons have emerged, such as *Ants* (1998, PDI), *A Bug's Life* (1998, Pixar), and *Toy Story 2* (1999, Pixar). Many animations of this type have been made for television. In an episode of *The Simpsons* (1995, PDI), Homer steps into a synthetic world and turns into a 3D computer-generated character. There have been popular television commercials involving computer animation, such as some Listerine commercials, the Shell dancing cars, and a few LifeSavers commercials. Many Saturday morning cartoons are now produced using 3D computer animation. Because many images are generated to produce an animation, the rendering used in computer animation tends to be computationally efficient. An example of rendering at the other extreme is *Bunny* (1999, Blue Skies), which received an Academy Award for animated short. *Bunny* uses high-quality rendering in its imagery, including ray tracing and radiosity.

Three-dimensional computer graphics is playing an increasing role in the production of conventional hand-drawn animation. Computer animation has been used to model 3D elements in hand-drawn environments. *Technological Threat* (1988), previously mentioned, is an early animation that combined computer-animated characters with hand-drawn characters to produce an entertaining commentary on the use of technology. Three-dimensional environments were constructed for conventionally animated figures in *Beauty and the Beast* (1991, Disney) and *Tarzan* (1999, Disney); three-dimensional synthetic objects, such as the chariots, were animated in conventionally drawn environments in *Prince of Egypt* (1998, Dreamworks). Because photorealism is not the objective, the rendering in such animation is done to blend with the relatively simple rendering of hand-drawn animation.

Last, morphing, even though it is a two-dimensional animation technique, should be mentioned because of its use in some films and its high impact in television commercials. This is essentially a 2D procedure that warps control points (or feature lines) of one image into the control points (feature lines) of another image while the images themselves are blended. In *Star Trek IV: The Voyage Home* (1986, ILM), one of the first commercial morphs occurred in the back-in-time dream sequence. In *Willow* (1988, ILM), a series of morphs changes one animal into another. This technique is also used very effectively in *Terminator II* (1991, ILM).

PDI is known for its use of morphing in various commercials. Michael Jackson's music video *Black and White,* in which people's faces morph into other faces, did much to popularize the technique. In a Plymouth Voyager commercial the previous year's car bodies and interiors morph into the new models, and in an Exxon commercial a car changes into a tiger. Morphing remains a useful and popular technique.

## 1.5 Chapter Summary

Computer graphics and animation have created a revolution in visual effects. Advances are still being made, and new effects are finding a receptive audience. Yet there is more potential to be realized as players in the entertainment industry demand their own special look and desire a competitive edge. Computer animation has come a long way since the days of Ivan Sutherland and the University of Utah. Viewed as another step in the development of animation, the use of digital technology is indeed both a big and an important step in the history of animation. With the advent of low-cost computing and desktop video, animation is now within reach of more people than ever. It remains to be seen how the limits of the technology will be pushed as new and interesting ways to create moving images are explored.

## References

1. R. Balzer, *Optical Amusements: Magic Lanterns and Other Transforming Images; A Catalog of Popular Entertainments,* Richard Balzer, 1987.
2. P. Burns, "The Complete History of the Discovery of Cinematography," http://www.precinemahistory.net/introduction.htm, 2000.
3. D. Coynik, *Film: Real to Reel,* McDougal, Littell, Evanston, Ill., 1976.
4. R. Hackathorn, "Anima II: A 3-D Color Animation System," *Computer Graphics* (Proceedings of SIGGRAPH 77), 11 (2), pp. 54–64 ( July 1977, San Jose, Calif.). Edited by James George.
5. M. Henne, H. Hickel, E. Johnson, and S. Konishi, "The Making of *Toy Story*" (Proceedings of Comp Con '96), pp. 463–468 (February 25–28, 1996, San Jose, Calif.).
6. M. Horton, C. Mumby, S. Owen, B. Pank, and D. Peters, "Quantel On-Line, Non-linear Editing," http://www.quantel.com/editingbook/index.html, 2000.
7. M. Hunt, "Cinema: 100 Years of the Moving Image," http://www.angelfire.com/vt/mhunt/cinema.html, 2000.
8. J. Lasseter, "Principles of Traditional Animation Applied to 3D Computer Animation," *Computer Graphics* (Proceedings of SIGGRAPH 87), 21 (4), pp. 35–44 (July 1987, Anaheim, Calif.). Edited by Maureen C. Stone.

9. K. Laybourne, *The Animation Book: A Complete Guide to Animated Filmmaking—from Flip-Books to Sound Cartoons to 3-D Animation,* Three Rivers Press, New York, 1998.
10. N. Magnenat-Thalmann and D. Thalmann, *Computer Animation: Theory and Practice,* Springer-Verlag, New York, 1985.
11. N. Magnenat-Thalmann and D. Thalmann, *New Trends in Animation and Visualization,* John Wiley & Sons, New York, 1991.
12. L. Maltin, *Of Mice and Magic: A History of American Animated Cartoons,* Penguin Books, New York, 1987.
13. B. Pryor, Opus Communications, "VIDEOFAQ#1: What the Heck Is 'Non-linear' Editing Anyway?" http://www.opuskc.com/vf1.html, 2000.
14. H. Siegel, "An Overview of Computer Animation & Modelling," *Computer Animation: Proceedings of the Conference Held at Computer Graphics '87,* pp. 27–37, (October 1987, London).
15. C. Solomon, *The History of Animation: Enchanted Drawings,* Wings Books, New York, 1994.
16. G. Stern, "Bbop: A Program for 3-Dimensional Animation," *Nicograph* 83, December 1983, pp. 403–404.
17. Synthetic Aperture, "Synthetic Aperture," http://www.synthetic-ap.com/tips/index.html, 2000.
18. B. Thomas, *Disney's Art of Animation from Mickey Mouse to "Beauty and the Beast,"* Hyperion, New York, 1991.
19. F. Thomas and O. Johnson, *The Illusion of Life,* Hyperion, New York, 1981.
20. Tribeca Technologies, LLC, "White Paper: The History of Video," http://tribecatech.com/histvide.htm, 2000.
21. Wabash College Theater, "Georges Méliès," http://www.wabash.edu/depart/theater/THAR4/Melies.htm, 2000.
22. R. Whittaker, "Linear and Non-linear Editing," http://www.internetcampus.com/tvp056.htm, 2000.

CHAPTER 2

# Technical Background

This chapter serves as a prelude to the animation techniques presented in the remaining chapters. It is divided into two sections. The first serves as a quick review of the basics of the computer graphics display pipeline and discusses the control of round-off error when repeatedly transforming data. It is assumed that the reader has already been exposed to transformation matrices, homogeneous coordinates, and the display pipeline, including the perspective transformation; this section concisely reviews these. The second section covers various orientation representations and quaternion arithmetic, which is important for the discussion of orientation interpolation in Chapter 3.

## 2.1 Spaces and Transformations

Much of computer graphics and computer animation involves transforming data. Object data are transformed from a defining space into a world space in order to build a synthetic environment. Object data are transformed as a function of time in order to produce animation. And, finally, object data are transformed in order to view the object on a screen. The workhorse transformational representation of graphics is the 4x4 transformation matrix, which can be used to represent combinations of rotations, translations, and scales.

A coordinate space can be defined by using either a left- or a right-handed coordinate system. Left-handed coordinate systems have the *x*-, *y*-, and *z*-coordinate axes aligned as the thumb, index finger, and middle finger of the left hand are arranged when held at right angles to each other in a natural pose: thumb extending out to the side of the hand, the middle finger extending perpendicular to the palm, and the index finger held colinear with the forearm with no bend at the wrist. The right-handed coordinate system is organized similarly with respect to the right hand. These configurations are inherently different; there is no series of pure rotations that transforms a left-handed configuration of axes into a right-handed configuration. Which configuration to use is a matter of convention. It makes no difference as long as everyone knows and understands the implications. Below, the handedness of each space is given after all the spaces have been introduced. Some application areas make the assumption that the *y*-axis is "up." Other applications assume that the *z*-axis is up. As with handedness, it makes no difference as long as everyone is aware of the assumption being made. In this book, the *y*-axis is considered up.

This section first reviews the transformational spaces through which object data pass as they are massaged into a form suitable for display and then the use of homogeneous representations of points and the 4x4 transformation matrix representation of rotation, translation, and scale. Next come discussions of representing arbitrary position and orientation by a series of matrices, representing compound transformations in a matrix, and extracting a series of basic transformations from a compound matrix. The display pipeline is then described in terms of the transformation matrices used to effect it; the discussion is focused on transforming a point in space. In the case of transforming vectors, the computation is slightly different; see Appendix B. This section concludes with a discussion of round-off error considerations, including orthonormalization of a rigid transformation matrix.

### 2.1.1 The Display Pipeline

The *display pipeline* refers to the transformation of object data from its original defined space through a series of spaces until its final mapping onto the screen. The object data are transformed into different spaces in order to efficiently compute illumination, clip the data to the view volume, and perform the perspective transformation. After reviewing these spaces and their properties and the transformations that map data from one space to the next, this section defines useful terminology. The names used for these spaces vary from text to text, so they will be reviewed here to establish a consistent naming convention for the rest of the book. Clipping, while an important process, is not relevant to motion control and therefore is not covered.

The space in which an object is originally defined is referred to as *object space*. The data in object space are usually centered around the origin and often are created to lie within some limited standard range such as minus one to plus one. The object, as defined by its data points (which are also referred to as its *vertices*), is transformed, usually by a series of rotations, translations, and scales, into *world space*, the space in which objects are assembled to create the environment to be viewed.

In addition, world space is the space in which light sources and the observer are placed. For purposes of this discussion, *observer position* is used synonymously and interchangeably with *camera position* and *eye position*. The observer parameters include its *position* and its orientation, consisting of the *view direction* and the *up vector*. There are various ways to specify these orientation vectors. Sometimes the view direction is specified by giving a *center of interest* (COI), in which case the view direction is the vector from the observer or eye position (EYE), also known as the *look-from point,* to its center of interest, also known as the *look-to point*. The default orientation "straight up" defines the object's up vector as the vector that is perpendicular to the view direction and in the plane defined by the view direction and the global $y$-axis. This vector can be computed by first taking the cross product of the view direction vector and the $y$-axis and then taking the cross product of this vector with the view direction vector (Equation 2.1). Head tilt information can be provided in one of two ways. It can be given by specifying an angle deviation from the straight up direction, in which case a head tilt rotation matrix can be incorporated in the world- to eye-space transformation. Alternatively, head tilt information can be given by specifying an up direction vector. The user-supplied up direction vector is typically not required to be perpendicular to the view direction as that would require too much work on the part of the user. Instead, the vector supplied by the user, together with the view direction vector, defines the plane in which the up vector lies. The difference between the user-supplied up direction vector and the up vector is that the up vector by definition is perpendicular to the view direction vector. The computation of the up vector is the same as that outlined in Equation 2.1, with the user-supplied up direction vector replacing the $y$-axis.

$$
\begin{aligned}
w &= COI - EYE && \text{view direction vector}\\
u &= w \times (0, 1, 0) && \text{cross product with } y\text{-axis}\\
v &= u \times w && \text{up vector}
\end{aligned}
\tag{Eq. 2.1}
$$

Care must be taken when using a default up vector. Defined as perpendicular to the view vector and in the plane of the view vector and global $y$-axis, it is undefined for straight up and straight down views. These situations must be dealt with as special cases or simply avoided. In addition to the undefined cases, some observer motions can result in unanticipated effects. For example, the default head-up orientation means that if the observer has a fixed center of interest and the observer's position arcs directly over the center of interest, then just before and just after being directly overhead, the observer's up vector will instantaneously rotate by up to 180 degrees (see Figure 2.1).

In addition to the observer position and orientation, the field of view has to be specified, as is standard in the display pipeline. This includes an *angle of view* (or the equally useful half angle of view), *hither clipping distance,* and *yon clipping distance.* Sometimes the terms *near* and *far* are used instead of *hither* and *yon.* The field of view information is used to set up the *perspective projection.*

The view specification discussed above is somewhat simplified. Other view specifications use an additional vector to indicate the orientation of the projection plane, allow an arbitrary viewport to be specified on the plane of projection that is not symmetrical about the view direction to allow for off-center projections, and allow for a parallel projection. The reader should refer to standard graphics texts such as the one by Foley et al. [2] for in-depth discussion of such view specifications.

The visible area of world space is formed by the observer position, view direction, angle of view, hither clipping distance, and yon clipping distance (Figure 2.2). These define the *view frustum,* the six-sided volume of world space containing data that need to be considered for display.

In preparation for the perspective transformation, the data points defining the objects are usually transformed from world space to *eye space.* In *eye space,* the observer is positioned along the $z$-axis with the line of sight made to coincide with

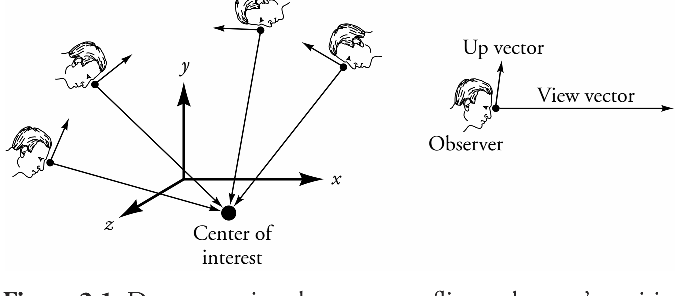


**Figure 2.1** Demonstrating the up vector flip as observer's position passes straight over center of interest

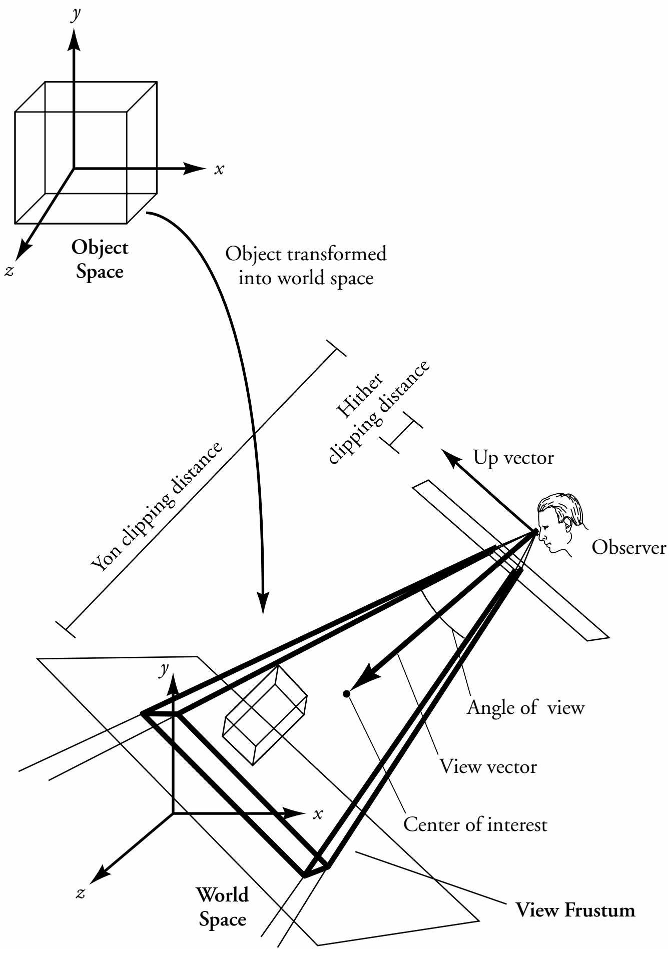


**Figure 2.2** Object- to world-space transformation and the view frustum in world space

the $z$-axis. This allows the depth of a point, and therefore perspective scaling, to be dependent only on the point's $z$-coordinate. The exact position of the observer along the $z$-axis and whether the eye space coordinate system is left-handed or right-handed vary from text to text. For this discussion, the observer is positioned at the origin looking down the positive $z$-axis in left-handed space. In eye space as in world space, lines of sight emanate from the observer position and diverge as they expand into the visible view frustum, whose shape is often referred to as a *truncated pyramid*.

The *perspective transformation* transforms the objects' data points from eye space to *image space*. The perspective transformation can be considered as taking the observer back to negative infinity in $z$ and, in doing so, makes the lines of sight parallel to each other and to the $z$-axis. The pyramid-shaped view frustum becomes a rectangular solid, or cuboid, whose opposite sides are parallel. Thus, points that are farther away from the observer in eye space have their $x$- and $y$-coordinates scaled down more than points closer to the observer. This is sometimes referred to as *perspective foreshortening*. Visible extents in image space are usually standardized into the minus one to plus one range in $x$ and $y$ and from zero to one in $z$ (although in some texts visible $z$ is mapped into the minus one to positive one range). Image space points are then scaled and translated (and possibly rotated) into *screen space* by mapping the visible ranges in $x$ and $y$ (minus one to plus one) into ranges that coincide with the viewing area defined in the coordinate system of the window or screen; the $z$-coordinates can be left alone. The resulting series of spaces is shown in Figure 2.3.

Ray casting (ray tracing without generating secondary rays) differs from the above sequence of transformations in that the act of tracing rays through world space implicitly accomplishes the perspective transformation. If the rays are constructed in world space based on pixel coordinates of a virtual frame buffer positioned in front of the observer, then the progression through spaces for ray casting reduces to the transformations shown in Figure 2.4. Alternatively, data can be transformed to eye space and, through a virtual frame buffer, the rays can be formed in eye space.

In any case, animation is typically produced by the following: modifying the position and orientation of objects in world space over time; modifying the shape of objects over time; modifying display attributes of objects over time; transforming the observer position and orientation in world space over time; or some combination of these transformations.

### 2.1.2 Homogeneous Coordinates and the Transformation Matrix

Computer graphics often uses homogeneous representations of points. This means that a three-dimensional point is represented by a four-element vector.[1] The coordinates of the represented point are determined by dividing the fourth component into the first three (Equation 2.2).

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) = [x, y, z, w] \qquad \textbf{(Eq. 2.2)}$$

[1] Note the potential source of confusion in the use of the term *vector* to mean (1) a direction in space or (2) a $1xn$ or $nx1$ matrix. The context in which *vector* is used should make its meaning clear.

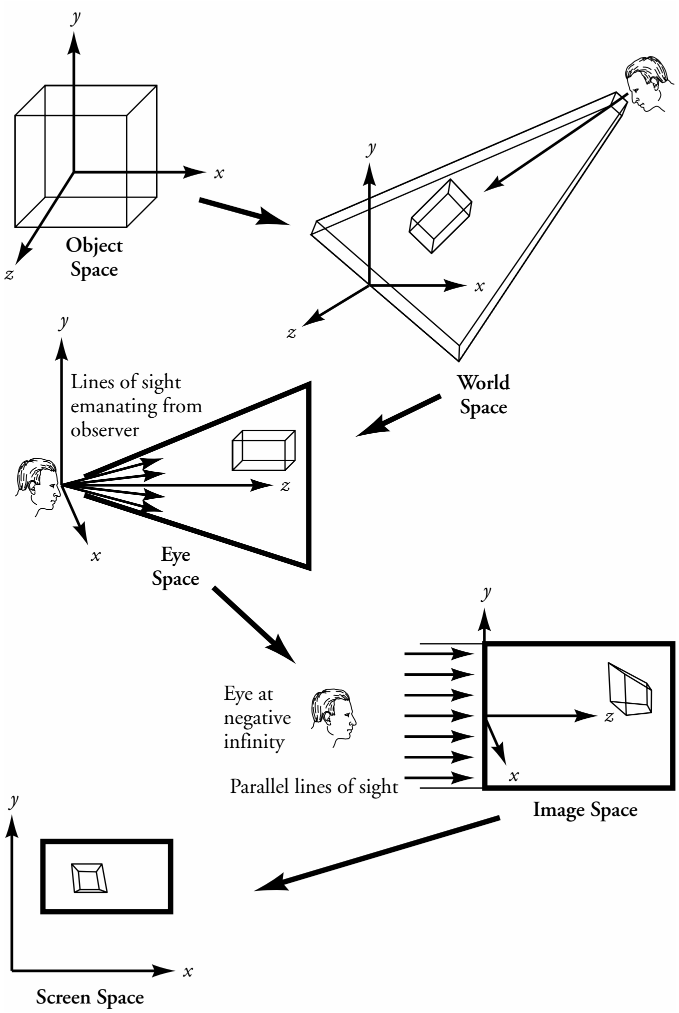


**Figure 2.3** Display pipeline showing transformation between spaces

Typically, when transforming a point in world space, the fourth component will be one. This means a point in space has a very simple homogeneous representation (Equation 2.3).

$$(x, y, z) = [x, y, z, 1] \tag{Eq. 2.3}$$

The basic transformations rotate, translate, and scale can be kept in 4x4 transformation matrices. The 4x4 matrix is the smallest matrix that can represent all of the basic transformations, and, because it is a square matrix, it has the potential for

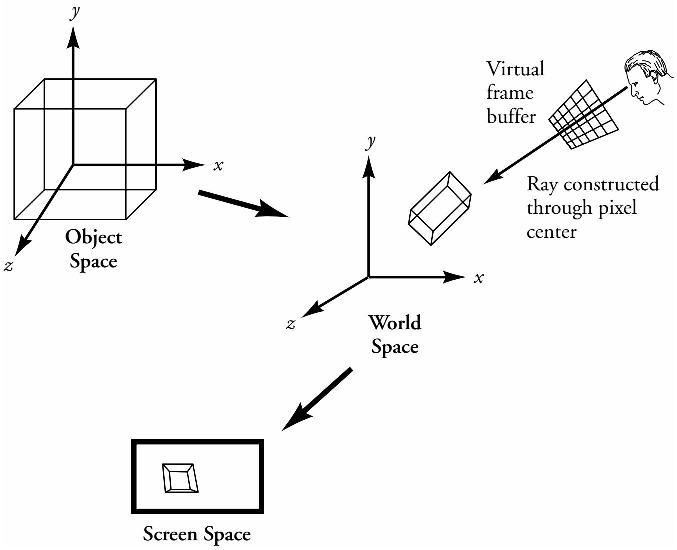


**Figure 2.4** Transformation through spaces using ray casting

having a computable inverse, which is important for texture mapping and illumination calculations. In the case of the transformations rotation, translation, and nonzero scale, the matrix always has a computable inverse. It can be multiplied with other transformation matrices to produce compound transformations while still maintaining 4x4-ness. The 4x4 identity matrix has zeros everywhere except along its diagonal; the diagonal elements all equal one (Equation 2.4).

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{Eq. 2.4}$$

Typically in the literature, a point is represented as a 4x1 column matrix (also known as a *column vector*) and is transformed by multiplying by a 4x4 matrix on the left (also known as *premultiplying* the column vector by the matrix), as shown in Equation 2.4. However, some texts use a 1x4 matrix (also known as a *row vector*) to represent a point and transform it by multiplying it by a matrix on its right (*postmultiplying*). For example, postmultiplying a point by the identity transformation would appear as in Equation 2.5.

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{Eq. 2.5}$$

Because the conventions are equivalent, it is immaterial which is used as long as consistency is maintained. The 4x4 transformation matrix used in one of the notations is the transpose of the 4x4 transformation matrix used in the other notation.

## 2.1.3 Compounding Transformations: Multiplying Transformation Matrices

One of the main advantages of representing transformations as square matrices is that they can be multiplied together to produce a compound transformation. This enables a series of transformations, $M_i$, to be premultiplied so that a single compound transformation matrix, $M$, can be applied to a point $P$ (see Equation 2.6). This is especially useful (i.e., computationally efficient) when applying the same series of transformations to a multitude of points. Note that matrix multiplication is associative $((AB)C = A(BC))$ but not commutative $(AB \neq BA)$.

$$\begin{aligned} P' &= M_1 M_2 M_3 M_4 M_5 M_6 P \\ M &= M_1 M_2 M_3 M_4 M_5 M_6 \\ P' &= MP \end{aligned} \tag{Eq. 2.6}$$

When using the convention of postmultiplying a point represented by a row vector by the same series of transformations used when premultiplying a column vector, the matrices will appear in reverse order as well as being the transpose of the matrices used in the premultiplication. Equation 2.7 shows the same computation as Equation 2.6, except in Equation 2.7 a row vector is postmultiplied by the transformation matrices. The matrices in Equation 2.7 are the same as those in Equation 2.6 but are now transposed and in reverse order. The transformed point is the same in both equations, with the exception that it appears as a column vector in Equation 2.6 and as a row vector in Equation 2.7. In the remainder of this book, such equations will be in the form shown in Equation 2.6.

$$\begin{aligned} P'^T &= P^T M_6^T M_5^T M_4^T M_3^T M_2^T M_1^T \\ M^T &= M_6^T M_5^T M_4^T M_3^T M_2^T M_1^T \\ P'^T &= P^T M^T \end{aligned} \tag{Eq. 2.7}$$

### 2.1.4 Basic Transformations

For now, only the basic transformations rotate, translate, and scale (uniform as well as nonuniform) will be considered. These transformations, and any combination of them, are referred to as *affine transformations* [3]. The perspective transformation is discussed later. Restricting discussion to the basic transformations allows the fourth element of each point vector to be assigned the value one and the last row of the transformation matrix to be assigned the value [0001] (Equation 2.8).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.8)}$$

The *x, y,* and *z* translation values of the transformation are the first three values of the fourth column (*d, h, m* in Equation 2.8). The upper left 3x3 submatrix represents rotation and scaling. Setting the upper left 3x3 submatrix to an identity transformation and specifying only translation produces Equation 2.9.

$$\begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.9)}$$

A transformation consisting of only uniform scale is represented by the identity matrix with a scale factor, *S,* replacing the first three elements along the diagonal (*a, f, k* in Equation 2.8). Nonuniform scale allows for independent scale factors to be applied to the *x*-, *y*-, and *z*-coordinates of a point and is formed by placing $S_x$, $S_y$, and $S_z$ along the diagonal as shown in Equation 2.10.

$$\begin{bmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.10)}$$

Uniform scale can also be represented by setting the lowest rightmost value to $1/S$, as in Equation 2.11. In the homogeneous representation, the coordinates of the point represented are determined by dividing the first three elements of the vector by the fourth, thus scaling up the values by the scale factor *S.* This technique invalidates the assumption that the only time the lowest rightmost element is not one is during perspective, and, therefore, this should be used with care or avoided altogether.

$$\begin{bmatrix} S\cdot x \\ S\cdot y \\ S\cdot z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{1}{S} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{S} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.11)}$$

Values to represent rotation are set in the upper left 3x3 submatrix (*a, b, c, e, f, g, i, j, k* of Equation 2.8). Rotation matrices around the *x*-axis, *y*-axis, and *z*-axis in a right-handed coordinate system are shown in Equation 2.12, Equation 2.13, and Equation 2.14, respectively. In a right-handed coordinate system, a positive angle of rotation produces a counterclockwise rotation as viewed from the positive end of the axis looking toward the origin (the right-hand rule).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.12)}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.13)}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.14)}$$

Combinations of rotations and translations are usually referred to as *rigid transformations* because the spatial extent of the object does not change; only its position and orientation in space are changed. Sometimes uniform scale is included in the family of rigid transformations because the object's intrinsic properties[2] (e.g.,

[2] An object's intrinsic properties are those that are measured irrespective of an external coordinate system.

dihedral angles)[3] do not change. Nonuniform scale, however, is usually not considered a rigid transformation because object properties such as dihedral angles are changed. A *shear* transformation is a combination of rotation and nonuniform scale and it creates columns (rows) that are nonorthogonal to each other but still retains the last row of three zeros followed by a one. Notice that any affine transformation can be represented by a multiplicative 3x3 matrix (representing rotations, scales, and shears) followed by an additive three-element vector (translation).

## 2.1.5 Representing an Arbitrary Orientation

### Fixed Angle Representation

One way to represent an orientation is as a series of rotations around the principal axes (the *fixed angle representation*). For example, consider that an object, say, an aircraft, is originally defined at the origin of a right-handed coordinate system with its nose pointed down the $z$-axis and its up vector in the positive $y$-axis direction. Now imagine that the desire is to position the aircraft in world space so that its center is at (20, –10, 35), its nose is oriented toward the point (23, –14, 40), and its up vector is pointed in the general direction of the $y$-axis (or, mathematically, so that its up vector lies in the plane defined by the aircraft's center, the point the plane is oriented toward, and the global $y$-axis). See Figure 2.5.

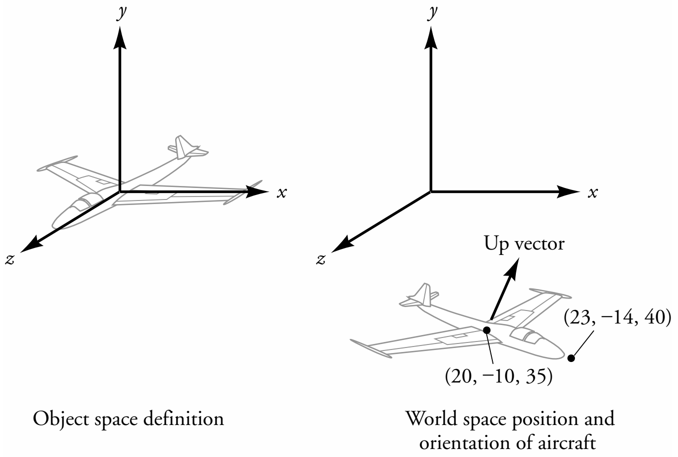


**Figure 2.5** Desired position and orientation

[3] The dihedral angle is the interior angle between adjacent polygons measured at the common edge

The task is to determine the representation of the transformation from the aircraft's object space definition to its desired position and orientation in world space. The transformation can be decomposed into a (possibly compound) rotation followed by a translation of (20, –10, 35). The rotation will transform the aircraft to an orientation so that, with its center at the origin, its nose is oriented toward (23 – 20, –14 + 10, 40 – 35) = (3, –4, 5); this will be referred to as the aircraft's orientation vector. The transformation that takes the aircraft into the desired orientation (i.e., transforms the positive *z*-axis vector into the plane's desired orientation vector while keeping its up vector generally aligned with the positive *y*-axis) can be determined by noting that an *x*-axis rotation (pitch) followed by a *y*-axis rotation (yaw) will produce the desired result. The sines and cosines necessary for the rotation matrices can be determined by considering the desired orientation vector's relation to the principal axes. In first considering the *x*-axis rotation, the orientation vector is projected onto the *y*-*z* plane to see how the nose must be rotated up or down (pitch). The sines and cosines can be read from the triangle formed by the projected vector and the line segment that drops from the end of the vector to the *z*-axis (Figure 2.6). To rotate the object to line up with the projected vector, a positive *x*-axis rotation with $\sin\psi = -4/\sqrt{50}$ and $\cos\psi = 34/\sqrt{50}$ is required.

Note that if the desired orientation vector projects onto the *y*-axis, then the orientation vector lies in the *x*-*y* plane. In this case, the sines and cosines of the appropriate pitch can be read from the orientation vector's *x*- and *y*-coordinate values and used in a *z*-axis rotation matrix.

After the pitch rotation has been applied to spin the aircraft around (yaw) to its desired orientation, a *y*-axis rotation can be determined by looking at the (rotated) direction vector in the *x*-*z* plane. To rotate the aircraft, a positive *y*-axis rotation with $\sin\phi = 3/\sqrt{34}$ and $\cos\phi = 5/\sqrt{34}$ is required (Figure 2.7).

An alternative way to represent a transformation to a desired orientation is to construct what is known as the *matrix of direction cosines.* Consider transforming a copy of the global coordinate system so that it coincides with a desired

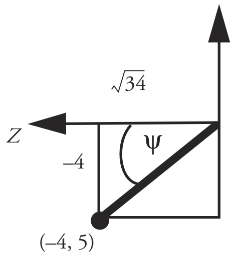


**Figure 2.6** Projection of desired orientation vector onto *y*-*z* plane

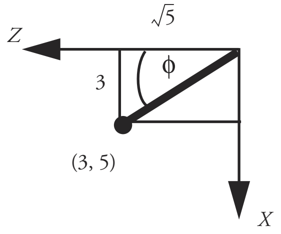
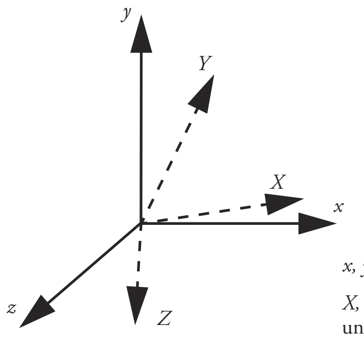


**Figure 2.7** Projection of desired orientation vector onto *x*-*z* plane



*x, y, z*—global coordinate system

*X, Y, Z*—desired orientation defined by unit coordinate system

**Figure 2.8** Global coordinate system and unit coordinate system to be transformed

orientation defined by a unit coordinate system (see Figure 2.8). To construct this matrix, note that the transformation matrix, *M,* should do the following: map the unit *x*-axis vector into the *X*-axis of the desired orientation, map a unit *y*-axis vector into the *Y*-axis of the desired orientation, and map a unit *z*-axis vector into the *Z*-axis of the desired orientation. See Equation 2.15. These three mappings can be assembled into one matrix expression that defines the matrix *M* (Equation 2.16).

$$X = M \cdot x \qquad Y = M \cdot y \qquad Z = M \cdot z$$

$$\begin{bmatrix} X_x \\ X_y \\ X_z \end{bmatrix} = M \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} Y_x \\ Y_y \\ Y_z \end{bmatrix} = M \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} Z_x \\ Z_y \\ Z_z \end{bmatrix} = M \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{Eq. 2.15}$$

$$\begin{bmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{bmatrix} = M \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{bmatrix} = M \tag{Eq. 2.16}$$

Since a unit *x*-vector (*y*-vector, *z*-vector) multiplied by a transformation matrix will replicate the values in the first (second, third) column of the transformation matrix, the columns of the transformation matrix can be filled with the coordinates of the desired transformed coordinate system. Thus, the first column of the transformation matrix becomes the desired *X*-axis as described by its *x*-, *y*-, and *z*-coordinates in the global space, the second column becomes the desired *Y*-axis, and the third column becomes the desired *Z*-axis (Equation 2.16). The name *matrix of direction cosines* is derived from the fact that the coordinates of a desired axis in terms of the global coordinate system are the cosines of the angles made by the desired axis with each of the global axes.

In the example of transforming the aircraft, the desired *Z*-axis is the desired orientation vector. With the assumption that there is no longitudinal rotation (roll), the desired *X*-axis can be formed by taking the cross product of the original *y*-axis and the desired *Z*-axis. The desired *Y*-axis can then be formed by taking the cross product of the desired *Z*-axis and the desired *X*-axis. Each of these is divided by its length to form unit vectors.

### 2.1.6 Extracting Transformations from a Matrix

For a compound transformation matrix that represents a series of rotations and translations, a set of individual transformations can be extracted from the matrix, which, when multiplied together, produce the original compound transformation matrix. Notice that the series of transformations to produce a compound transformation is not unique, so there is no guarantee that the series of transformations so extracted will be exactly the ones that produced the compound transformation (unless something is known about the process that produced the compound matrix).

The compound transformation can be formed by up to three rotations about the principal axes (or one compound rotation represented by the direction cosine matrix) followed by a translation.

The last row of a 4x4 transformation matrix, if the matrix does not include a perspective transformation, will have zero in the first three entries and one as the fourth entry (ignoring the use of that element to represent uniform scale). As shown in Equation 2.17, the first three elements of the last column of the matrix, $A_{14}$, $A_{24}$, $A_{34}$, represent a translation. The upper left 3x3 submatrix of the original 4x4 matrix can be viewed as the definition of the transformed unit coordinate system. It can be decomposed into three rotations around principal axes by arbitrarily choosing an ordered sequence of three axes (such as *x* followed by *y* followed by *z*). By using the projection of the transformed unit coordinate system to determine the sines and cosines, the appropriate rotation matrices can be formed in much the same way that transformations were determined in Section 2.1.5

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \text{(Eq. 2.17)}$$

If the compound transformation matrix includes a uniform scale factor, the rows of the 3x3 submatrix will form orthogonal vectors of uniform length. The length will be the scale factor, which, when followed by the rotations and translations, forms the decomposition of the compound transformation. If the rows of the 3x3 submatrix form orthogonal vectors of unequal length, then their lengths represent nonuniform scale factors.

### 2.1.7 Description of Transformations in the Display Pipeline

Now that the basic transformations have been described in some detail, the previously described transformations of the display pipeline can be explained in terms of compositions of the basic transformations. It should be noted that the descriptions of eye space and the perspective transformation are not unique. They vary among the introductory graphics texts depending on where the observer is placed along the *z*-axis to define eye space, whether the eye space coordinate system is left-handed or right-handed, exactly what information is required from the user in describing the perspective transformation, and the range of visible *z*-values in image space. While functionally equivalent, the various approaches produce transformation matrices that differ in the values of the individual elements.

**Object Space to World Space Transformation** In a simple implementation, the transformation of an object from its object space into world space is a series of rotations, translations, and scales (i.e., an affine transformation) that are specified by the user to place a transformed copy of the object data into a world space data structure. In some systems, the user is required to specify this transformation in terms of a predefined order of basic transformations such as scale, rotation around the *x*-axis, rotation around the *y*-axis, rotation around the *z*-axis, and translation. In other systems, the user may be able to specify an arbitrarily ordered sequence of basic transformations. In either case, the series of transformations are compounded into a single object space to world space transformation matrix.

The object space to world space transformation is usually the transformation that is modified over time to produce motion. In more complex animation systems, this transformation may include manipulations of arbitrary complexity not suitable for representation in a matrix, such as nonlinear shape deformations.

**World Space to Eye Space Transformation** In preparation for the perspective transformation, a rigid transformation is performed on all of the object data in world space. The transformation is designed so that, in eye space, the observer is positioned at the origin, the view vector aligns with the positive *z*-axis in left-handed space, and the up vector aligns with the positive *y*-axis. The transformation is formed as a series of basic transformations. First, the observer is translated to the origin. Then, the observer's coordinate system (view vector, up vector, and the third vector required to complete a left-handed coordinate system) is transformed by up to three rotations so as to align the view vector with the global negative *z*-axis and the up vector with the global *y*-axis. Finally, the *z*-axis is flipped by negating the *z*-coordinate. All of the individual transformations can be represented by 4x4 transformation matrices, which are multiplied together to produce a single, compound, world space to eye space transformation matrix. This transformation prepares the data for the perspective transformation by putting it in a form in which the perspective divide is simply dividing by a point's *z*-coordinate.

**Perspective Matrix Multiply** The perspective matrix multiplication is the first part of the perspective transformation. The fundamental computation being performed by the perspective transformation is that of dividing the *x*- and *y*-coordinates by their *z*-coordinate and normalizing the visible range in *x* and *y* to [–1, +1]. This is accomplished by using a homogeneous representation of a point and, as a result of the perspective matrix multiplication, producing a representation in which the fourth element is $Z_e \cdot \tan \phi$. $Z_e$ is the point's *z*-coordinate in eye space and $\phi$ is the half angle of view. The *z*-coordinate is transformed so that planarity is preserved and so that the visible range in *z* is mapped into [0, +1]. (These ranges are arbitrary and can be set to anything by appropriately forming the perspective matrix. For example, sometimes the visible range in *z* is set to [–1, +1].) In addition, the aspect ratio of the viewport can be used in the matrix to modify either the *x* or *y* half angle of view so that no distortion results in the viewed data.

**Perspective Divide** Each point produced by the perspective matrix multiplication has a nonunit fourth component that represents the perspective divide by *z*. Dividing each point by its fourth component completes the perspective transformation. This is considered a separate step from the perspective matrix multiply because a commonly used clipping procedure operates on the homogeneous representation of points produced by the perspective matrix multiplication but before perspective divide.

**Image to Screen Space Mapping** The result of the perspective transformation (the perspective matrix multiply followed by perspective divide) maps visible elements into the range of minus one to plus one ([–1, +1]) in *x* and *y*. This range is now mapped into the user-specified viewing area of the screen-based pixel coordinate system. This is a simple linear transformation represented by a scale and a translation and thus can be easily represented in a 4x4 transformation matrix.

Clipping, removing data that are outside the view frustum, can be implemented in a variety of ways. It is computationally simpler if clipping is performed after the world space to eye space transformation. It is important to perform clipping in z before perspective divide, to prevent divide by zero and from projecting objects behind the observer onto the picture plane. However, the details of clipping are not relevant to the discussion here. Interested readers should refer to one of the standard computer graphics texts (e.g., [2]) for the details of clipping procedures.

### 2.1.8 Round-off Error Considerations

Once the object space to world space transformation matrix has been formed for an object, the object is transformed into world space by simply multiplying all of the object's object space points by the transformation matrix. When an object is animated, its points will be repeatedly transformed over time—as a function of time. One way to do this is to repeatedly modify the object's world space points. However, incremental transformation of world space points usually leads to the accumulation of round-off errors. For this reason, it is almost always better to modify the transformation from object to world space and reapply the transformation to the object space points rather than repeatedly transform the world space coordinates. To further transform an object that already has a transformation matrix associated with it, one simply has to form a transformation matrix and premultiply it by the existing transformation matrix to produce a new one. However, round-off errors can also accumulate when one repeatedly modifies a transformation matrix. The best way is to build the transformation matrix anew for each application.

An affine transformation matrix can be viewed as a 3x3 rotation/scale submatrix followed by a translation. Most of the error accumulation occurs because of the operations resulting from multiplying the 3x3 submatrix and the *x*-, *y*-, and *z*-coordinates of the point. Therefore, the following round-off error example will focus on the errors that accumulate as a result of rotations.

Consider the case of the moon orbiting the earth. For the sake of simplicity, the assumption is that the center of the earth is at the origin, and initially the moon data are defined with the moon's center at the origin. The moon data are first transformed to an initial position relative to the earth, for example ($r$, 0, 0) (see Figure 2.9). There are three approaches that could be taken to animate the rotation of the moon around the earth, and these will be used to illustrate various effects of round-off error.

The first approach is, for each frame of the animation, to apply a delta *y*-axis transformation matrix to the moon's points, in which each delta represents the angle it moves in one frame time (see Figure 2.10). Round-off errors will accumulate in the world space object points. Points that began as coplanar will no longer be coplanar. This can have undesirable effects, especially in display algorithms, which linearly interpolate values to render a surface.

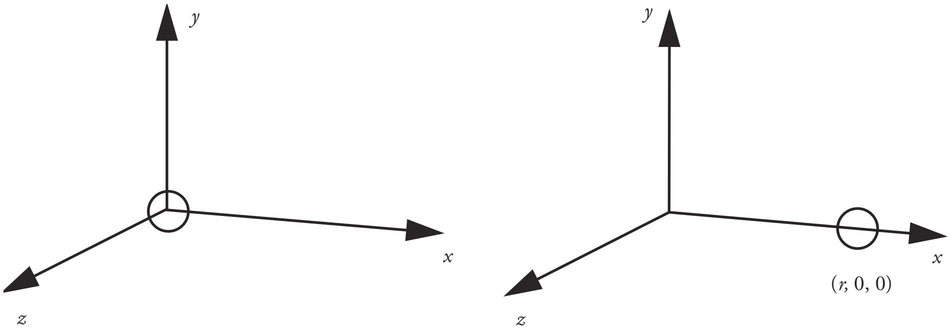
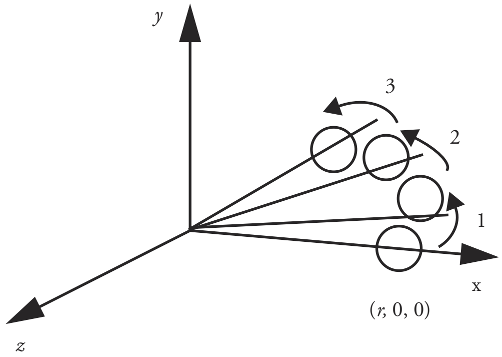


**Figure 2.9** Translation of moon out to its initial position on the *x*-axis

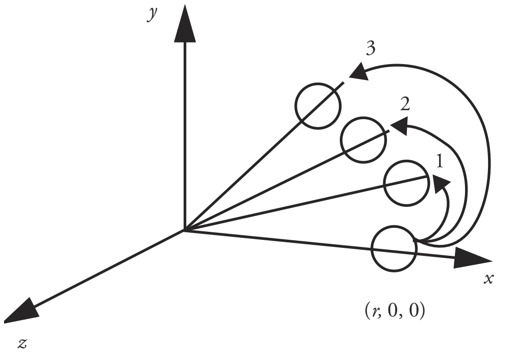


```
for each point P of the moon {
  P′ = P
}
$R_{dy}$ = y-axis rotation of 5 degrees
repeat until (done) {
  for each point P’ of the moon {
    P′ = $R_{dy}$*P′
  }
  record a frame of the animation
}
```

**Figure 2.10** Rotation by applying incremental rotation matrices to points

The second approach is, for each frame, to incrementally modify the transformation matrix that takes the object space points into the world space positions. In the example of the moon, the transformation matrix is initialized with the *x*-axis translation matrix. For each frame, a delta *y*-axis transformation matrix multiplies the current transformation matrix and then that resultant matrix is applied to the moon's object space points (see Figure 2.11). Round-off error will accumulate in the transformation matrix. Over time, the matrix will deviate from representing a rigid transformation. Shearing effects will begin to creep into the transformation and angles will cease to be preserved. While a square may begin to look like something other than a square, coplanarity will be preserved (because any matrix multiplication is, by definition, a linear transformation), so that rendering results will not be compromised.

The third approach is to add the delta value to an accumulating angle variable and then build the *y*-axis rotation matrix from that angle parameter. This would



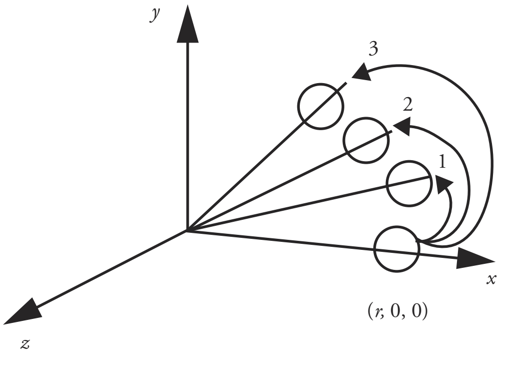
```
R = identity matrix
Rdy = y-axis rotation of 5 degrees
repeat until (done) {
  for each point P of the moon {
    P′ = R*P
  }
  record a frame of the animation
  R = R*Rdy
}
```

**Figure 2.11** Rotation by incrementally updating the rotation matrix

then be multiplied with the *x*-axis translation matrix, and the resultant matrix would be applied to the original moon points in object space (see Figure 2.12). In this case, any round-off error will accumulate in the angle variable so that it may begin to deviate from what is desired. This may have unwanted effects when one tries to coordinate motions, but the transformation matrix, which is built anew every frame, will not accumulate any errors itself. The transformation will always represent a valid rigid transformation with both planarity and angles being preserved.

### Orthonormalization

The rows of a matrix that represent a rigid transformation are perpendicular to each other and are of unit length (orthonormal). The same can be said of the matrix columns. If values in a rigid transformation matrix have accumulated errors, then the rows cease to be orthonormal and the matrix ceases to represent a rigid transformation; it will have the effect of introducing shear into the transfor-



```
y = 0
repeat until (done) {
R = y-axis rotation matrix of ‘y’ degrees
for each point P of the moon {
  P′ = R*P
}
record a frame of the animation
y = y+5
}
```

**Figure 2.12** Rotation by forming the rotation matrix anew for each frame

mation. However, if it is known that the matrix is supposed to represent a rigid transformation, it can be massaged back into a rigid transformation matrix. A rigid transformation matrix (assume for now that this means not any uniform scale) has an upper 3x3 submatrix with specific properties: the rows (columns) are unit vectors orthogonal to each other. A simple procedure to reformulate the transformation matrix to represent a rigid transformation is to take the first row (column) and normalize it. Take the second row (column), compute the cross product of this row (column) and the first row (column), normalize it, and place it in the third row (column). Take the cross product of the third row (column) and the first row (column), normalize it, and put it in the second row (column). See Figure 2.13. Notice that this does not necessarily produce the correct transformation; it merely forces the matrix to represent a rigid transformation. The error has just been shifted around so that the columns of the matrix are orthonormal and the error may be less noticeable.

If the transformation might contain a uniform scale, then take the length of one of the rows, or the average length of the three rows, and, instead of normalizing the vectors by the steps described above, make them equal to this length. If the transformation might include nonuniform scale, then the difference between shear and error accumulation cannot be determined unless something more is known about the transformations represented.

## 2.2 Orientation Representation

A common issue that arises in computer animation is deciding the best way to represent the position and orientation of an object in space and how to interpolate the represented transformations over time to produce motion. A typical scenario is one in which the user specifies an object in two transformed states and the computer is used to interpolate intermediate states, thus producing animated keyframe motion. This section discusses possible orientation representations and identifies strengths and weaknesses; the next chapter addresses the best way to interpolate orientations using these representations. In this discussion, it is assumed that the final transformation applied to the object is a result of rotations and translations only, so that there is no scaling involved, nonuniform or otherwise; that is, the transformations considered are *rigid body.*

The first obvious choice for representing the orientation and position of an object is by a 4x4 transformation matrix. For example, a user may specify a series of rotations and translations to apply to an object. This series of transformations is compiled into 4x4 matrices and multiplied together to produce a compound 4x4 transformation matrix. In such a matrix the upper left 3x3 submatrix represents a

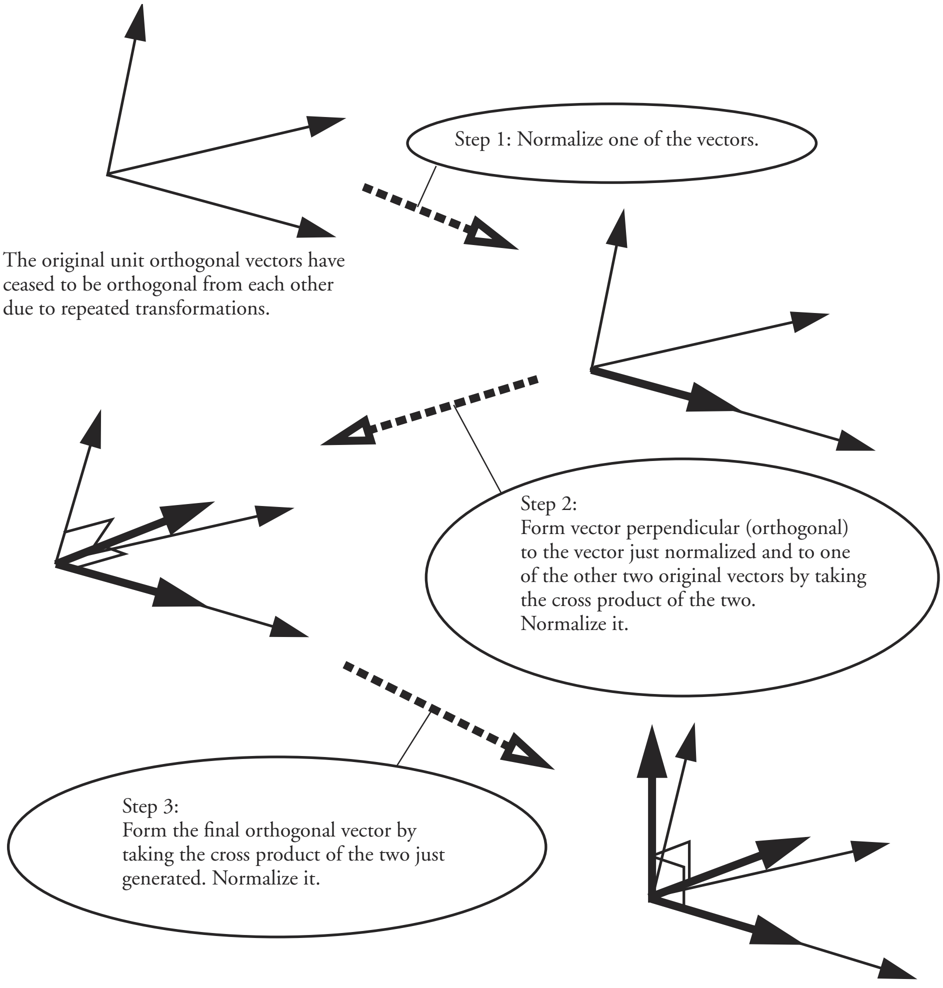

Step 1: Normalize one of the vectors.
The original unit orthogonal vectors have ceased to be orthogonal from each other due to repeated transformations.
Step 2:
Form vector perpendicular (orthogonal) to the vector just normalized and to one of the other two original vectors by taking the cross product of the two.
Normalize it.
Step 3:
Form the final orthogonal vector by taking the cross product of the two just generated. Normalize it.

**Figure 2.13** Orthonormalization

rotation to apply to the object, while the first three elements of the fourth column represent the translation (assuming points are represented by column vectors that are premultiplied by the transformation matrix). No matter how the 4x4 transformation matrix was formed (no matter in what order the transformations were given by the user, such as "rotate about *x,* translate, rotate about *x,* rotate about *y,* translate, rotate about *y*), the final 4x4 transformation matrix produced by multiplying all of the individual transformation matrices in the specified order will result in a matrix that specifies the final position of the object by a 3x3 rotation matrix followed by a translation. The conclusion is that the rotation can be interpolated independently from the translation. (For now, only linear interpolation is considered, although higher-order interpolations are possible; see Appendix B.)

Now consider two such transformations that the user has specified as key states with the intention of generating intermediate transformations by interpolation. While it should be obvious that interpolating the translations is straightforward, it is not at all clear how to go about interpolating the rotations. In fact, it is the objective of this discussion to show that interpolation of orientations can be a problem. A property of 3x3 rotation matrices is that the rows and columns are orthonormal (unit length and perpendicular to each other). Simple linear interpolation between the nine pairs of numbers that make up the two 3x3 rotation matrices to be interpolated will not produce intermediate 3x3 matrices that are orthonormal and are, therefore, not rigid body rotations. It should be easy to see that interpolating from a rotation of +90 degrees about the *y*-axis to a rotation of –90 degrees about the *y*-axis results in an intermediate transformation that is nonsense (Figure 2.14).

So direct interpolation of transformation matrices is not acceptable. There are alternative representations that are more useful than transformation matrices in performing such interpolations: fixed angle, Euler angle, axis angle, and quaternions.

## 2.2.1 Fixed Angle Representation

*Fixed angle* representation[4] really refers to “angles used to rotate about fixed axes.” A fixed order of three rotations is implied, such as *x-y-z.* This means that orientation is given by a set of three ordered parameters that represent three ordered rotations about fixed axes, first around *x,* then around *y,* and then around *z.* There are many possible orderings of the rotations, and, in fact, it is not necessary to use all three coordinate axes. For example, *x-y-x* is a feasible set of rotations. The only orderings that do not make sense are those in which an axis immediately follows itself, such as in *x-x-y.*

In any case, the main point is that the orientation of an object is given by three angles, such as (10, 45, 90). In this example, the orientation represented is obtained by rotating the object first about the *x*-axis by 10 degrees, then about the *y*-axis by 45 degrees, and then about the *z*-axis by 90 degrees. In Figure 2.15, the aircraft is shown in its initial orientation and in the orientation represented by the fixed point values of (10, 45, 90). The following notation will be used to represent such a sequence of rotations: $R_z(90) R_y(45)\ R_x(10)$ (in this text, transformations are

[4] Terms referring to rotational representations are not used consistently in the literature. This book follows the usage found in *Robotics* [1], where *fixed angle* refers to rotation about the fixed (global) axes and *Euler angle* refers to rotation about the rotating (local) axes.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Positive 90-degree $y$-axis rotation

$$\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Negative 90-degree $y$-axis rotation

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Halfway between orientation representations

**Figure 2.14** Direct interpolation of transformation matrix values can result in nonsense

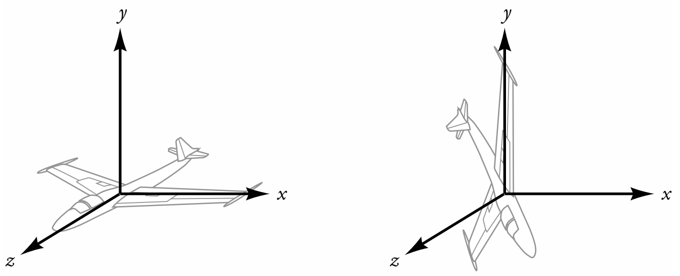


**Figure 2.15** Fixed angle representation

implemented by premultiplying points by transformation matrices; thus the rotation matrices appear in right to left order).

From this orientation, changing the $x$-axis rotation value, which is applied first to the data points, will make the aircraft's nose dip either more or less in the $y$-$z$ plane. Changing the $y$-axis rotation will change the amount the aircraft, which has been rotated around the $x$-axis, rotates out of the $y$-$z$ plane. Changing the $z$-axis rotation value, the rotation applied last, will change how much the twice rotated aircraft will rotate about the $z$-axis.

The problem when using this scheme is that two of the axes of rotation can effectively line up on top of each other when an object can rotate freely in space or around a 3-degrees-of-freedom[5] joint. Consider an object in an orientation repre-

[5] The degrees of freedom (DOFs) that an object possesses is the number of independent variables that have to be specified to completely locate that object (and all of its parts).

sented by (0, 90, 0), as shown in Figure 2.16. Examine what effect a slight change in the first and third parametric values has on the object in that orientation. A slight change of the third parameter will rotate the object slightly about the global *z*-axis because that is the rotation applied last to the data points. However, note that the effect of a slight change of the first parameter, which rotates the original data points around the *x*-axis, will also have the effect of rotating the transformed object slightly about the *z*-axis (Figure 2.17). This effect results because the 90-degree *y*-axis rotation has essentially made the first axis of rotation align with the third axis of rotation. This effect is called *gimbal lock*. From the orientation (0, 90, 0) the object can no longer be rotated about the global *x*-axis by a simple change in its orientation representation (actually, the representation that will effect such an orientation is (90, 90 + ε, 90)—not very intuitive).

This same problem makes interpolation between key positions problematic in some cases. Consider the key orientations (0, 90, 0) and (90, 45, 90), as shown in Figure 2.18. The second orientation is a 45-degree *x*-axis rotation from the first position. However, as discussed above, the object can no longer directly rotate about the *x*-axis from the first key orientation because of the 90-degree *y*-axis rotation. Direct interpolation of the key orientation representations would produce (45, 67.5, 45) as the halfway orientation, which is very different from the (90,

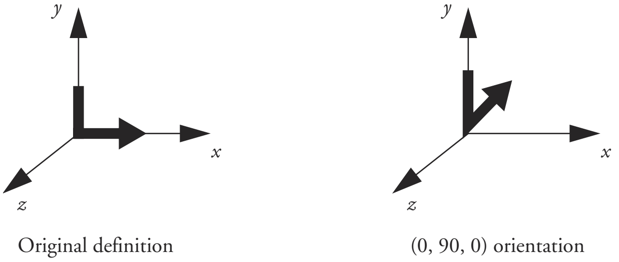


**Figure 2.16** Fixed angle representation of (0, 90, 0)

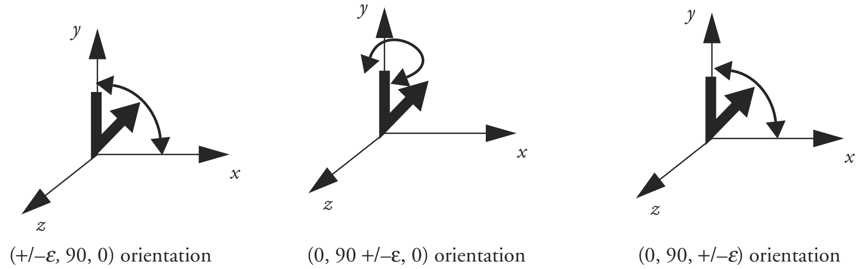
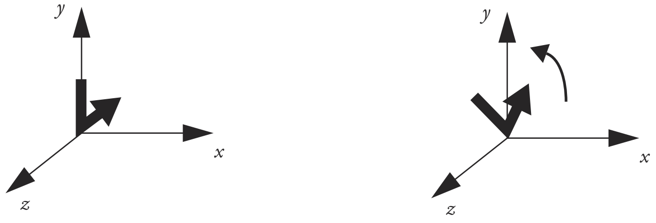

y
x
z
(+/–ε, 90, 0) orientation
y
x
z
(0, 90 +/–ε, 0) orientation
y
x
z
(0, 90, +/–ε) orientation

**Figure 2.17** Effect of slightly altering values of fixed angle representation (0, 90, 0)



(0, 90, 0) orientation (90, 45, 90) orientation; the object lies in the *y*-*z* plane

**Figure 2.18** Example orientations to interpolate

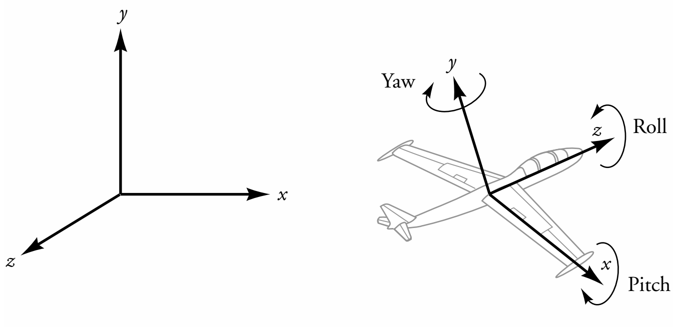
22.5, 90) orientation that is desired (desired because that is the representation of the orientation that is intuitively halfway between the two given orientations). The result is that the object will swing out of the *y*-*z* plane during the interpolation, which is not the behavior expected.

In its favor, the fixed angle representation is compact, fairly intuitive, and easy to work with. However, it is often not the most desirable representation to use because of the gimbal lock problem.

## 2.2.2 Euler Angle Representation

In a *Euler angle* representation, the axes of rotation are the axes of the local coordinate system fixed to the object, as opposed to the global axes. A typical example of using Euler angles is found in the roll, pitch, and yaw of an aircraft (Figure 2.19).

As with the fixed angle representation, the Euler angle representation can use any of various orderings of three axes of rotation as its representation scheme.



Global coordinate system Local coordinate system attached to object

**Figure 2.19** Euler angle representation

Consider a Euler angle representation that uses an *x*-*y*-*z* ordering and is specified as (α, β, γ). The *x*-axis rotation, represented by the transformation matrix $R_x(\alpha)$, is followed by the *y*-axis rotation, represented by the transformation matrix $R_y(\beta)$, around the *y*-axis of the local, rotated coordinate system. Using a prime symbol to represent rotation about a rotated frame and remembering that points are represented as column vectors and are premultiplied by transformation matrices, one achieves a result of $R_y'(\beta)R_x(\alpha)$. Using global axis rotation matrices to implement the transformations, the *y*-axis rotation around the rotated frame can be effected by $R_x(\alpha)R_y(\beta)R_x(-\alpha)$. Thus, the result after the first two rotations is Equation 2.18.

$$R_y'(\beta)R_x(\alpha) = R_x(\alpha)R_y(\beta)R_x(-\alpha)R_x(\alpha) = R_x(\alpha)R_y(\beta) \qquad \text{(Eq. 2.18)}$$

The third rotation, $R_z(\gamma)$, is around the now twice rotated frame. This rotation can be effected by undoing the previous rotations with $R_x(-\alpha)$ followed by $R_y(-\beta)$, then rotating around the global *z*-axis by $R_z(\gamma)$, and then redoing the previous rotations. Putting all three rotations together, and using a double prime to denote rotation about a twice rotated frame, results in Equation 2.19.

$$\begin{aligned} R_z''(\gamma)R_y'(\beta)R_x(\alpha) &= R_x(\alpha)R_y(\beta)R_z(\gamma)R_y(-\beta)R_x(-\alpha)R_x(\alpha)R_y(\beta) \\ &= R_x(\alpha)R_y(\beta)R_z(\gamma) \qquad \text{(Eq. 2.19)} \end{aligned}$$

Thus, this system of Euler angles is precisely equivalent to the fixed angle system in reverse order. This is true for any system of Euler angles. For example, *z*-*y*-*x* Euler angles are equivalent to *x*-*y*-*z* fixed angles. Therefore, the Euler angle representation has exactly the same advantages and disadvantages as the fixed angle representation.

### 2.2.3 Angle and Axis

Euler showed that one orientation can be derived from another by a single rotation about an axis. This is known as Euler's rotation theorem [1]. Thus, any orientation can be represented by a four-tuple consisting of an angle and an (*x, y, z*) vector (Figure 2.20).

In some cases, this can be a useful representation. Interpolation between representations ($A_1$, $\theta_1$) and ($A_2$, $\theta_2$) can be implemented by interpolating the axes of rotation and the angles separately (Figure 2.21).[6] An intermediate axis can be determined by rotating one axis partway toward the other. The axis of rotation is formed by taking the cross product of two axes, and the angle between the two axes is determined by taking the inverse cosine of the dot product of normalized

[6] A small raised dot (·) is used often to help disambiguate expressions involving scalar multiplication. A large raised dot (•) is used to represent the dot product operation involving vectors.

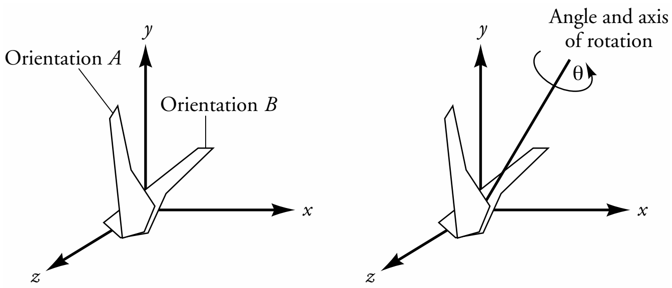
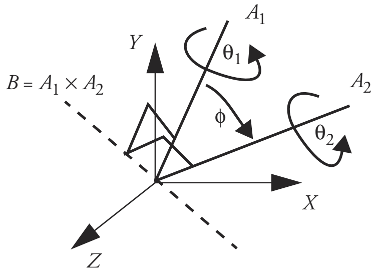


**Figure 2.20** Euler's rotation theorem implies that for any two orientations of an object, one can be produced from the other by a single rotation about an arbitrary axis



$$B = A_1 \times A_2$$

$$\phi = \cos^{-1}\left(\frac{A_1 \bullet A_2}{|A_1||A_2|}\right)$$

$$A_k = R_B(k \cdot \phi)A_1$$

$$\theta_k = (1-k) \cdot \theta_1 + k \cdot \theta_2$$

**Figure 2.21** Interpolating axis-angle representations

versions of the axes. An interpolant, *k,* can then be used to form an intermediate axis and angle pair. Note that the axis-angle representation cannot be easily used when compiling a series of rotations. However, the information contained in this representation can be put in a form in which these operations are easily implemented: quaternions.

## 2.2.4 Quaternions

As shown, the representations above have drawbacks when interpolating intermediate orientations when an object or joint has three degrees of rotational freedom. A better approach is to use *quaternions* to represent orientation [4]. A quaternion is a four-tuple of real numbers, [*s, x, y, z*] or, equivalently, [*s, v*], consisting of a scalar, *s,* and a three-dimensional vector, *v.*

The quaternion is an alternative to the axis and angle representation that contains the same information in a different form. Importantly, it is in a form that can be interpolated as well as used in compiling a series of rotations into a single representation. The axis and angle information of a quaternion can be viewed as an orientation of an object relative to its initial object space definition, or it can be considered as the representation of a rotation to apply to an object definition. In the former view, being able to interpolate between represented orientations is important in generating key-frame animation. In the latter view, compiling a series of rotations into a simple representation is a common and useful operation to perform to apply a single, compound transformation to an object definition.

### Basic Quaternion Math

Before interpolation can be carried out, one must first understand some basic quaternion math. *Quaternion addition* is simply the four-tuple addition of quaternion representations, $[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2]$. *Quaternion multiplication* is defined as Equation 2.20. Notice that quaternion multiplication is not commutative, $q_1 \cdot q_2 \neq q_2 \cdot q_1$, but associative, $(q_1 \cdot q_2) \cdot q_3 = q_1 \cdot (q_2 \cdot q_3)$.

$$[s_1, v_1] \cdot [s_2, v_2] = [s_1 \cdot s_2 - v_1 \bullet v_2, s_1 \cdot v_2 + s_2 \cdot v_1 + v_1 \times v_2] \qquad \text{(Eq. 2.20)}$$

In these equations, the small dot represents scalar multiplication, the large dot represents dot product, and "×" denotes cross product. A point in space, *v,* or, equivalently, the vector from the origin to the point, is represented as [0, *v*]. It is easy to see that quaternion multiplication of two orthogonal vectors ($v_1 \bullet v_2 = 0$) computes the cross product of those vectors (Equation 2.21).

$$[0, v_1] \cdot [0, v_2] = [0, v_1 \times v_2] \qquad \text{iff } v_1 \bullet v_2 = 0 \qquad \text{(Eq. 2.21)}$$

The quaternion [1, (0, 0, 0)] is the *multiplicative identity;* that is, [*s, v*] · [1 (0, 0, 0)] = [*s, v*]. The *inverse of a quaternion*, $[s, v]^{-1}$, is obtained by negating its vector part and dividing both parts by the magnitude squared (the sum of the squares of the four components), as shown in Equation 2.22.

$$q^{-1} = (1/\|q\|)^2 \cdot [s, -v]$$
$$\text{where} \qquad \|q\| = \sqrt{s^2 + x^2 + y^2 + z^2} \qquad \text{(Eq. 2.22)}$$

Multiplication of a quaternion, *q,* by its inverse, $q^{-1}$, results in the *unit-length quaternion* [1, (0, 0, 0)]. A unit-length quaternion (also referred to here as a *unit quaternion*) is created by dividing each of the four components by the square root of the sum of the squares of those components (Equation 2.23).

$$q/(\|q\|) \qquad \text{(Eq. 2.23)}$$

### Rotating Vectors Using Quaternions

To *rotate a vector, v,* using quaternion math, represent the vector as [0, *v*] and represent the rotation by a quaternion, *q* (how to represent rotations by a quaternion is discussed below). The vector is rotated according to Equation 2.24.

$$v' = Rot(v) = q \cdot v \cdot q^{-1} \qquad \text{(Eq. 2.24)}$$

A series of rotations can be compiled into a single representation by quaternion multiplication. Consider a rotation represented by a quaternion *p* followed by a rotation represented by a quaternion *q* on a vector, *v* (Equation 2.25).

$$\begin{aligned} Rot_q(Rot_p(v)) &= q \cdot (p \cdot v \cdot p^{-1}) \cdot q^{-1} \\ &= ((qp) \cdot v \cdot (qp)^{-1}) \\ &= Rot_{qp}(v) \end{aligned} \qquad \text{(Eq. 2.25)}$$

The inverse of a quaternion represents rotation about the same axis by the same amount but in the reverse direction. Equation 2.26 shows that rotating a vector by a quaternion *q* followed by rotating the result by a quaternion *q*-inverse produces the original vector.

$$Rot^{-1}(Rot(v)) = q^{-1} \cdot (q \cdot v \cdot q^{-1}) \cdot q = v \qquad \text{(Eq. 2.26)}$$

Also notice that in performing rotation, $q \cdot v \cdot q^{-1}$, all effects of magnitude are divided out due to the multiplication by the inverse of the quaternion. Thus, any scalar multiple of a quaternion represents the same rotation as the corresponding unit quaternion, as with the homogeneous representation of points.

### Representing Rotations Using Quaternions

A rotation is represented in a quaternion form by encoding axis-angle information. Equation 2.27 shows a unit quaternion representation of a rotation of an angle, θ, about an axis of rotation (*x, y, z*).

$$q = Rot_{\theta,\,(x,\,y,\,z)} = [\cos(\theta/2),\ \sin(\theta/2) \cdot (x, y, z)] \qquad \text{(Eq. 2.27)}$$

Notice that a quaternion and its negation, [–*s*, –*v*] both represent the same rotation because the negated values indicate a negative rotation around the negated axis. The two negatives in this case cancel each other out and produce the same rotation (Equation 2.28).

$$\begin{aligned} -q &= Rot_{-\theta,\,-(x,\,y,\,z)} \\ &= [\cos(-\theta/2),\ \sin((-\theta)/2) \cdot (-(x, y, z))] \\ &= [\cos(\theta/2),\ -\sin(\theta/2) \cdot (-(x, y, z))] \\ &= [\cos(\theta/2),\ \sin(\theta/2) \cdot x, y, z] \\ &= Rot_{\theta,\,(x,\,y,\,z)} \\ &= q \end{aligned} \qquad \text{(Eq. 2.28)}$$

## 2.3 Chapter Summary

Linear transformations represented by 4x4 matrices are a fundamental operation in computer graphics and animation. Understanding their use, how to manipulate them, and how to control round-off error is an important first step in mastering graphics and animation techniques.

There are several orientation representations to choose from. The most robust representation of orientation is quaternions, but fixed angle, Euler angle, and axis-angle are more intuitive, easier to implement, and often sufficient for a given situation. Appendix B contains useful conversions between quaternions and other representations.

## References

1. J. Craig, *Robotics,* Addison-Wesley, New York, 1989.
2. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice,* 2d ed., Addison-Wesley, New York, 1990.
3. M. Mortenson, *Geometric Modeling,* John Wiley & Sons, New York, 1997.
4. K. Shoemake, "Animating Rotation with Quaternion Curves," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 143–152 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.

CHAPTER 3

# Interpolation and Basic Techniques

This chapter presents methods for precisely specifying the motion of objects. Using these techniques, the animator directly controls how the objects will move. There is little uncertainty about the positions and orientations to be produced, and the computer is used only to calculate the actual values. Procedures and algorithms are used, to be sure, but in a very direct manner in which the animator has very specific expectations about the motion that will be produced on a frame-by-frame basis. Chapter 4, on advanced algorithms, addresses more sophisticated techniques, in which the animator gives up some precision to produce motion with certain desired qualities. The high-level algorithms of the next chapter leave some uncertainty as to how the objects will be positioned for any particular frame.

## 3.1 Interpolation

The foundation of most animation is the interpolation of values. One of the simplest examples of animation is the interpolation of the position of a point in space. But even to do this correctly is nontrivial and requires some discussion of several issues: the appropriate interpolating function; the parameterization of the function based on distance traveled; and maintaining the desired control of the interpolated position over time.

Most of this discussion will be in terms of interpolating spatial values. The reader should keep in mind that any changeable value involved in the animation (and display) process such as an object's transparency, the camera's focal length, or the color of a light source could be subjected to interpolation over time.

Often, an animator has a list of values associated with a given parameter at specific frames (called *key frames* or *extremes*) of the animation. The question to be answered is how best to generate the values of the parameter for the frames between the key frames. The parameter to be interpolated may be a coordinate of the position of an object, a joint angle of an appendage of a robot, the transparency attribute of an object, or any other parameter used in the manipulation and display of computer graphics elements. Whatever the case, values for the parameter of interest must be generated for all of the frames between the key frames.

For example, if the animator wants the position of an object to be (–5, 0, 0) at frame 22 and the position to be (5, 0, 0) at frame 67, then values for the position need to be generated for frames 23 to 66. Linear interpolation could be used. But what if the object should appear to be stopped at frame 22 and needs to accelerate from that position, reach a maximum speed by frame 34, start to decelerate at frame 50, and come to a stop by frame 67? Or perhaps instead of stopping at frame 67, the object should continue to position (5, 10, 0) and arrive there at frame 80 by following a nice curved path? The next several sections address these issues of generating points along a path defined by control points and distributing the points along the path according to timing considerations.

### 3.1.1 The Appropriate Function

Appendix B contains a discussion of various specific interpolation techniques. In this section, the discussion covers general issues that determine how to choose the most appropriate interpolation technique and, once it is chosen, how to apply it in the production of an animated sequence.

The following issues need to be considered in order to choose the most appropriate interpolation technique: interpolation vs. approximation, complexity, continuity, and global vs. local control

#### Interpolation versus Approximation

Given a set of points to describe a curve, one of the first decisions one must make is whether the given values represent actual positions that the curve should pass through (interpolation) or whether they are meant merely to control the shape of the curve and do not represent actual positions that the curve will intersect (approximation). See Figure 3.1. This distinction is usually dependent on whether the data points are sample points of a desired curve or whether they are being used to design a new curve. In the former case, the desired curve is assumed to be constrained to travel through the sample points, which is, of course, the definition of an interpolating spline. In the latter case, an approximating spline can be used as the animator quickly gets a feel for how repositioning the control points influences the shape of the curve.

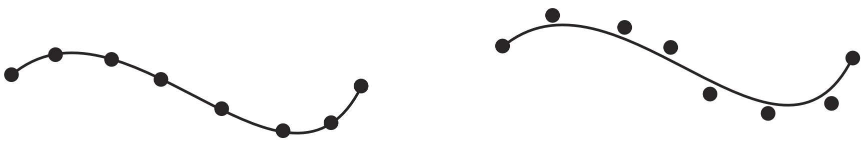
Commonly used interpolating functions are the Hermite formulation and the Catmull-Rom spline. The Hermite formulation requires tangent information at the endpoints, whereas Catmull-Rom uses only positions the curve should pass through. Functions that approximate some or all of the control information include Bezier and B-spline curves. See Appendix B for a more detailed discussion of these functions.

### Complexity

The complexity of the underlying interpolation equation is of concern because this translates into its computational efficiency. The simpler the underlying equations of the interpolating function, the faster its evaluation. In practice, polynomials are easy to compute, and, piecewise, cubic polynomials are the lowest-order polynomials that provide sufficient smoothness while still allowing enough flexibility to satisfy other constraints such as beginning and ending positions and tangents. A polynomial whose order is lower than cubic does not provide for a point of inflection between two endpoints and therefore may not fit smoothly to certain data points. Using a polynomial whose order is higher than cubic typically does not provide any significant advantages and is more costly to evaluate.

### Continuity

The smoothness in the resulting curve is a primary consideration. Mathematically, smoothness is determined by how many of the derivatives of the curve equation are continuous. Zeroth-order continuity refers to the continuity of values of the curve itself. Does the curve make any discontinuous jumps in its values? If a small



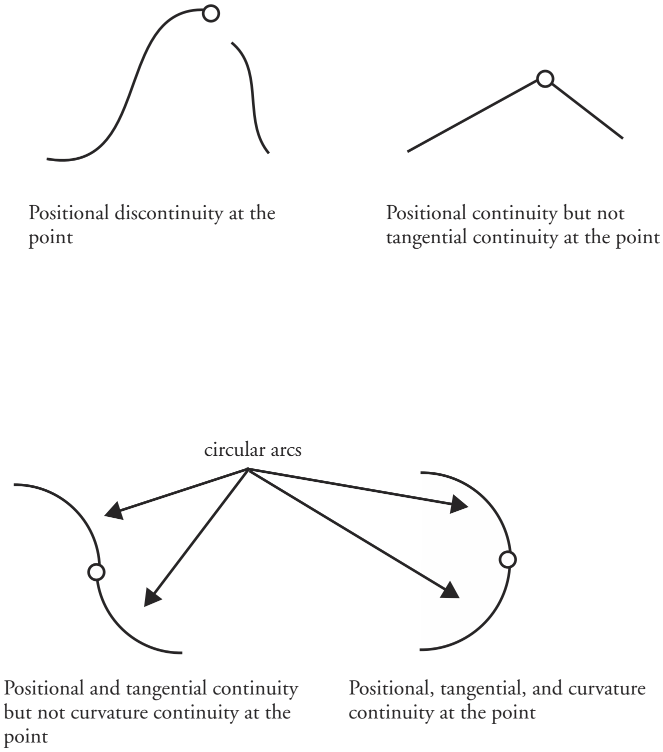
An interpolating spline in which the spline passes through the interior control points

An approximating spline in which only the endpoints are interpolated; the interior control points are used only to design the curve

**Figure 3.1** Comparing interpolation and approximating splines

change in the value of the parameter always results in a small change in the value of the function, then the curve has zeroth-order, or positional, continuity. If the same can be said of the first derivative of the function (the instantaneous change in values of the curve), then the function has first-order, or tangential, continuity. Second-order continuity refers to continuous curvature or instantaneous change of the tangent vector. See Figure 3.2. In some geometric design environments, second-order continuity of curves and surfaces may be needed, but in most animation applications, first-order continuity suffices.

In most applications, the curve is compound; it is made up of several segments. The issue then becomes, not the continuity within a segment (which in the case of polynomials is of infinite order), but the continuity enforced at the junction between adjacent segments. Hermite, Catmull-Rom, parabolic blending, and cubic Bezier curves (see Appendix B) can all produce first-order continuity between curve segments. There is a form of compound Hermite curves that produces second-order continuity between segments at the expense of local control (see next page). Cubic B-spline is second-order continuous everywhere. All of these curves provide sufficient continuity for most animation applications. Appendix B dis-


Positional discontinuity at the point

Positional continuity but not tangential continuity at the point

Positional and tangential continuity but not curvature continuity at the point

Positional, tangential, and curvature continuity at the point

**Figure 3.2** Continuity (at the point indicated by the small circle)

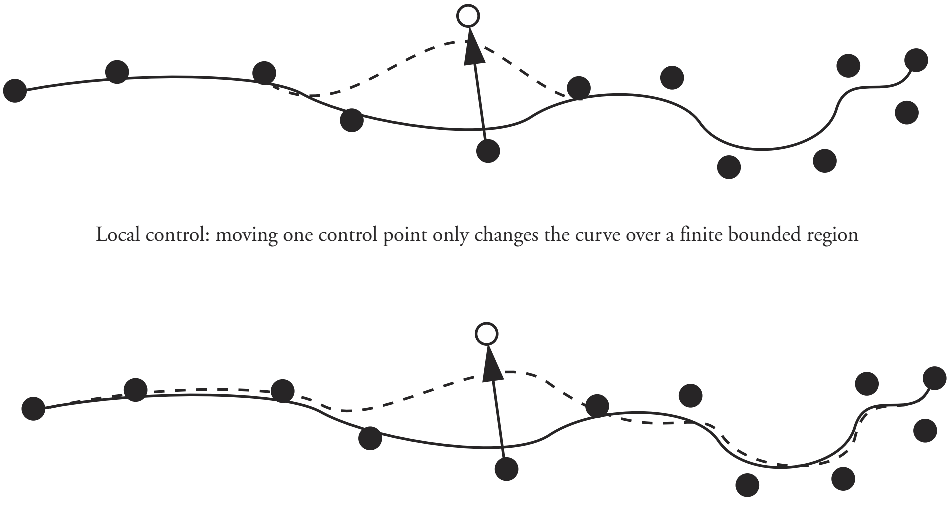
cusses continuity in more mathematical terms, with topics including the difference between *parametric continuity* and *geometric continuity*.

### Global versus Local Control

When designing a curve, a user often repositions one or just a few of the points that control the shape of the curve in order to tweak just part of the curve. It is usually considered an advantage if a change in a single control point has an effect on only a limited region of the curve as opposed to changing the entire curve. A formulation in which control points have a limited effect on the curve is referred to as providing *local control*. If repositioning one control point redefines the entire curve, however slightly, then the formulation provides *global control*. See Figure 3.3. Local control is almost always viewed as being the more desirable of the two. Almost all of the composite curves provide local control: parabolic blending, Catmull-Rom splines, composite cubic Bezier, and cubic B-spline. The form of Hermite curves that enforces second-order continuity at the segment junctions does so at the expense of local control. Higher-order Bezier and B-spline curves have less localized control than their cubic forms.

### Summary

There are many formulations that can be used to interpolate values. The specific formulation chosen depends on the desired continuity, whether local control is needed, the degree of computational complexity involved, and the information



Local control: moving one control point only changes the curve over a finite bounded region

Global control: moving one control point changes the entire curve; distant sections may change only slightly

**Figure 3.3** Comparing local and global effect of moving a control point

required from the user. The Catmull-Rom spline is often used in animation path movement because it is an interpolating spline and requires no additional information from the user other than the points that the path is to pass through. Bezier curves that are constrained to pass through given points are also often used. Parabolic blending is an often-overlooked technique that also affords local control and is interpolating. Formulations for these curves appear in Appendix B. See Mortenson [30] and Rogers and Adams [35] for more in-depth discussions.

## 3.2 Controlling the Motion Along a Curve

Designing the shape of a curve is only the first step in animation. The speed at which the curve is traced out as the parametric value is increased has to be under the direct control of the animator to produce predictable results. If the relationship between a change in parametric value and the corresponding distance along the curve is not known, then it becomes much more difficult for the animator to produce desired effects. The first step in giving the animator control is to establish a method for stepping along the curve in equal increments. Once this is done, methods for speeding up and slowing down along the curve can be made available to the animator.

For this discussion, it is assumed that an interpolating technique has been chosen and that a function $P(u)$ has been selected that, for a given value of $u$, will produce a value $p = P(u)$. If a "position in space" is being interpolated, then three functions are being represented in the following manner. The *x*-, *y*-, and *z*-coordinates for the positions at the key frames are specified by the user. Key frames are associated with specific values of the time parameter, $u$. The *x*-, *y*-, and *z*-coordinates are considered independently so that, for example, the *x*-coordinates of the points are used as control values for the interpolating curve so that $X = P_x(u)$, where $P$ denotes an interpolating function, and the subscript $x$ is used to denote that this specific curve was formed using the *x*-coordinates of the key positions. Similarly, $Y = P_y(u)$ and $Z = P_z(u)$ are formed so that for any specified time, $u$, a position $(X, Y, Z)$ can be produced as $(P_x(u), P_y(u), P_z(u))$.

It is very important to note that varying the parameter of interpolation (in this case $u$) by a constant amount does not mean that the resulting values (in this case Euclidean position) will vary by a constant amount. Thus, if positions are being interpolated by varying $u$ at a constant rate, the positions that are generated will not necessarily, in fact will seldom, represent a constant speed (e.g., see Figure 3.4).

To ensure a constant speed for the interpolated value, the interpolating function has to be parameterized by arc length, that is, distance along the curve of interpolation. Some type of reparameterization by arc length should be performed for

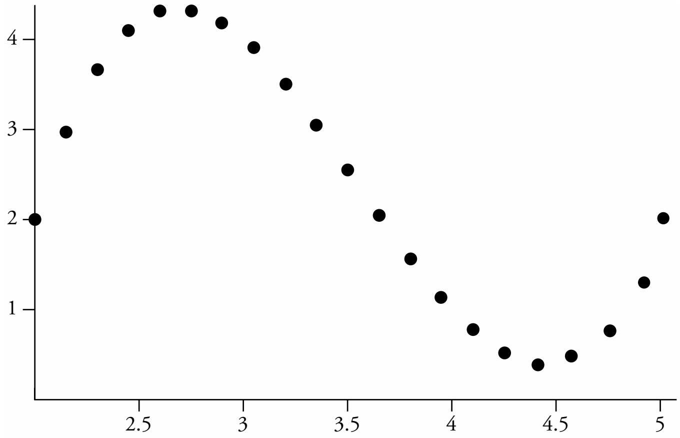


**Figure 3.4** Example of points produced by equal increments of an interpolating parameter for a typical cubic curve

most applications. Usually this reparameterization can be approximated without adding undue overhead or complexity to the interpolating function.

There are three approaches to establishing the reparameterization by arc length. The first two methods create a table of values to establish a relationship between parametric value and approximate arc length. This table can then be used to approximate parametric values at equal-length steps along the curve. The first method constructs the table by supersampling the curve and uses summed linear distances to approximate arc length. The second method uses Gaussian quadrature to numerically estimate the arc length. Both methods can benefit from an adaptive subdivision approach to controlling error. The third method analytically computes arc length. Unfortunately, many curves do not lend themselves to the analytic method.

### 3.2.1 Computing Arc Length

To specify how fast the object is to move along the path defined by the curve, an animator may want to specify the time at which positions along the curve should be attained. Referring to Figure 3.5 as an example in two-dimensional space, the animator may specify the following frame number and position pairs: (0, *A*), (10, *B*), (35, *C*), (60, *D*).

Alternatively, the animator may want to specify the relative velocities that an object should have along the curve. For example, the animator may specify that an object, initially at rest at position *A,* should smoothly accelerate until frame 20, maintain a constant speed until frame 35, and then smoothly decelerate until frame 60 at the end of the curve at position *D.* These kinds of constraints can be

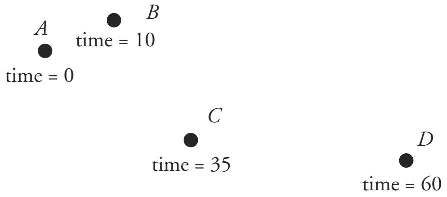


**Figure 3.5** Position-time pairs constraining the motion

accommodated in a system that can compute the distance traveled along any span of the curve.

Assume that the position of an object in three-space is being interpolated; the parameterized interpolation function is a *space curve*. The path of the object is a cubic polynomial as a function of a single parametric variable, that is, Equation 3.1.

$$P(u) = au^3 + bu^2 + cu + d \qquad \text{(Eq. 3.1)}$$

Notice that in three-space this really represents three equations: one for the *x*-coordinate, one for the *y*-coordinate, and one for the *z*-coordinate. Each of the three equations has its own constants *a, b, c,* and *d.* The equation can also be written explicitly representing these three equations, as in Equation 3.2.

$$\begin{aligned} P(u) &= (x(u), y(u), z(u)) \\ x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \qquad \text{(Eq. 3.2)}$$

Each of the three equations is a cubic polynomial of the form given in Equation 3.2. The curve itself can be specified using any of the standard ways of generating a spline (see Appendix B or texts on the subject, e.g., [35]). Once the curve has been specified, an object is moved along it by choosing a value of the parametric variable, and then the *x*-, *y*-, and *z*-coordinates of the corresponding point on the curve are calculated. It is important to remember that in animation the path swept out by the curve is not the only important thing. Equally important is how the path is swept out over time. A very different effect will be evoked by the animation if an object travels over the curve at a strictly constant speed instead of smoothly accelerating at the beginning and smoothly decelerating at the end. As a consequence, it is important to discuss both the curve that defines the path to be followed by the object and the function that relates time to distance traveled. To avoid confusion, the term *space curve* will be used to refer to the former and the term *distance-time function* will be used to refer to the latter. In discussing the distance-time function, the curve that represents the function will be referred to often. As a result, the terms *curve* and *function* will be used interchangeably in some contexts.

Notice that a function is desired that relates time to a position on the space curve. The user supplies, in one way or another (to be discussed later), the distance-time function that relates time to the distance traveled along the curve. The distance along a curve is referred to as *arc length* and, in this text, is denoted by $s$. When the arc length computation is given as a function of a variable $u$ and this dependence is noteworthy, then $s(u)$ is used. The arc length at a specific parametric value, such as $s(u_i)$, is often denoted as $s_i$. If the arc length computation is specified as a function of time, then $s(t)$ is used.

The interpolating function relates parametric value to position on the space curve. The relationship between distance along the curve and parametric value needs to be established. This relationship is the *arc length parameterization* of the space curve. It allows movement along the curve at a constant speed by evaluating the curve at equal arc length intervals. Further, it allows acceleration and deceleration along the curve by controlling the distance traveled in a given time interval.

For an arbitrary curve, it is usually not the case that a constant change in the parameter will result in a constant distance traveled. Because the value of the parameterizing variable is not the same as arc length, it is difficult to choose the values of the parameterizing variable so that the object moves along the curve at a desired speed. The relationship between the parameterizing variable and arc length is usually nonlinear. In the special case when a unit change in the parameterizing variable results in a unit change in curve length, the curve is said to be parameterized by arc length. Many seemingly difficult problems in control of motion in animation become very simple if the motion control curve can be parameterized by arc length or, if arc length can be numerically computed, given a value for the parameterizing variable.

Let the function $LENGTH(u_1, u_2)$ be the length along the space curve from the point $P(u_1)$ to the point $P(u_2)$. See Figure 3.6. Then the two problems to solve are

1. Given parameters $u_1$ and $u_2$, find $LENGTH(u_1, u_2)$.
2. Given an arc length $s$ and a parameter value $u_1$, find $u_2$ such that $LENGTH(u_1, u_2) = s$. This is equivalent to finding the zero point of the expression $s - LENGTH(u_1, u_2)$.

The first step in controlling the timing along a space curve is to establish the relationship between parametric values and arc length. This can be done by specifying the function $s = G(u)$, which computes, for any given parametric value, the

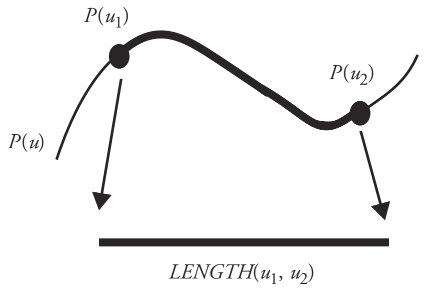


**Figure 3.6** *$LENGTH(u_1, u_2)$*

length of the curve from its starting point to the point that corresponds to that parametric value. Then if the inverse, $G^{-1}$, can be computed (or estimated), the curve can be effectively parameterized by arc length, that is, $P(G^{-1}(s))$. Once this is done, the second step is for the animator to specify, in one way or another, the distance the object should move along the curve for each time step.

### Establishing the Relationship between Parametric Value and Arc Length

In general, neither of the problems above has an analytic solution, so numerical solution techniques must be used. As stated above, the first step in controlling the motion of an object along a space curve is to establish the relationship between the parametric value and arc length. If possible, the curve should be explicitly parameterized by arc length by analyzing the space curve equation. Unfortunately, most types of parametric space curves are difficult or impossible to parameterize by arc length. For example, it has been shown [21] that B-spline curves cannot, in general, be parameterized by arc length. Since parameterization by arc length is not possible for many useful types of curves, several approximate parameterization techniques have been developed.

### The Analytic Approach to Computing Arc Length

The length of the curve from a point $P(u_1)$ to any other point $P(u_2)$ can be found by evaluating the arc length integral [25] (Equation 3.3) thus

$$s = \int_{u1}^{u2} |dP/du|\, du \qquad \text{(Eq. 3.3)}$$

where the derivative of the space curve with respect to the parameterizing variable is defined to be that shown in Equation 3.4 and Equation 3.5.

$$dP/du = ((dx(u)/(du)), (dy(u)/(du)), (dz(u)/(du))) \qquad \text{(Eq. 3.4)}$$

$$|dP/du| = \sqrt{(dx(u)/du)^2 + (dy(u)/du)^2 + (dz(u)/du)^2} \quad \text{(Eq. 3.5)}$$

For a cubic curve in which $P(u) = a \cdot u^3 + b \cdot u^2 + c \cdot u + d$ (remembering that, for a curve in three-space, this is really three equations, one for each coordinate), the derivative of one of the equations with respect to $u$ is $dx(u)/du = 3 \cdot a_x \cdot u^2 + 2 \cdot b_x \cdot u + c_x$. The equation inside the radical becomes Equation 3.6. For the two-dimensional case, the coefficients are given in Equation 3.7; the extension to 3D is straightforward.

$$A \cdot u^4 + B \cdot u^3 + C \cdot u^2 + D \cdot u + E \quad \text{(Eq. 3.6)}$$

$$\begin{aligned} A &= 9 \cdot (a_x^2 + a_y^2) \\ B &= 12 \cdot (a_x \cdot b_x + a_y \cdot b_y) \\ C &= 6 \cdot (a_x \cdot c_x + a_y \cdot c_y) + 4 \cdot (b_x^2 + b_y^2) \\ D &= 4 \cdot (b_x \cdot c_x + b_y \cdot c_y) \\ E &= c_x^2 + c_y^2 \end{aligned} \quad \text{(Eq. 3.7)}$$

### Estimating Arc Length by Forward Differencing

The easiest and conceptually simplest strategy for establishing the correspondence between parameter value and arc length is to sample the curve at a multitude of parametric values. Each parametric value produces a point along the curve. These sample points can then be used to estimate the arc length by computing the linear distance between adjacent points. As this is done, a table is built up of arc lengths indexed by parametric values. For example, given a curve $P(u)$, compute the positions along the curve for $u = 0.00, 0.05, 0.10, 0.15, \ldots, 1.0$. The table, indexed by $u$ and represented here as $G(u)$ would be computed as follows:

$G(0.0) = 0$

$G(0.05) =$ the distance between $P(0.00)$ and $P(0.05)$

$G(0.10) = G(0.05)$ plus the distance between $P(0.05)$ and $P(0.10)$

$G(0.15) = G(0.10)$ plus the distance between $P(0.10)$ and $P(0.15)$

. . .

$G(1.00) = G(0.95)$ plus the distance between $P(0.95)$ and $P(1.00)$

For example, consider the table of values for $u$ and corresponding values of the function $G$ in Table 3.1.

**Table 3.1** Parameter, Arc Length Pairs

| Index | Parametric Entry | Arc Length (G) |
|---|---|---|
| 0 | 0.00 | 0.000 |
| 1 | 0.05 | 0.080 |
| 2 | 0.10 | 0.150 |
| 3 | 0.15 | 0.230 |
| 4 | 0.20 | 0.320 |
| 5 | 0.25 | 0.400 |
| 6 | 0.30 | 0.500 |
| 7 | 0.35 | 0.600 |
| 8 | 0.40 | 0.720 |
| 9 | 0.45 | 0.800 |
| 10 | 0.50 | 0.860 |
| 11 | 0.55 | 0.900 |
| 12 | 0.60 | 0.920 |
| 13 | 0.65 | 0.932 |
| 14 | 0.70 | 0.944 |
| 15 | 0.75 | 0.959 |
| 16 | 0.80 | 0.972 |
| 17 | 0.85 | 0.984 |
| 18 | 0.90 | 0.994 |
| 19 | 0.95 | 0.998 |
| 20 | 1.00 | 1.000 |

As a simple example of how such a table could be used, consider the case in which the user wants to know the distance (arc length) from the beginning of the curve to the point on the curve corresponding to a parametric value of 0.73. The parametric entry closest to 0.73 must be located in the table. Because the parametric entries are evenly spaced, the location in the table of the closest entry to the given value can be determined by direct calculation. Using Table 3.1, the index, *i,* is determined by Equation 3.8.

$$\begin{aligned} i &= (int)\left(\frac{given\ parametric\ value}{distance\ between\ entries} + 0.5\right) \\ &= (int)\left(\frac{0.73}{0.05} + 0.5\right) = 15 \end{aligned} \quad \text{(Eq. 3.8)}$$

A crude approximation to the arc length is obtained by using the arc length entry located in the table at index 15, that is, 0.959. A better approximation can be obtained by interpolating between the arc lengths corresponding to entries on either side of the given parametric value. In this case, the index of the largest parametric entry that is less than the given value is desired (Equation 3.9).

$$i = (int)\left(\frac{given\ parametric\ value}{distance\ between\ entries}\right) = (int)\left(\frac{0.73}{0.05}\right) = 14 \qquad \textbf{(Eq. 3.9)}$$

An arc length, *L,* can be linearly interpolated from the arc lengths in the table by using the differences between the given parametric value and the parametric entries on either side of it in the table, as in Equation 3.10.

$$\begin{aligned} L &= ArcLength[i] + \frac{(GivenValue - Value[i])}{(Value[i+1] - Value[i])} \\ &\quad \cdot (ArcLength[i+1] - ArcLength[i]) \\ &= 0.944 + \frac{0.73 - 0.70}{0.75 - 0.70} \cdot (0.959 - 0.944) \\ &= 0.953 \end{aligned} \qquad \textbf{(Eq. 3.10)}$$

The solution to the first problem cited above (given two parameters $u_1$ and $u_2$, find the distance between the corresponding points) can be found by applying this calculation twice and subtracting the respective distances.

The reverse situation, that of finding the value of *u* given the arc length, is dealt with in a similar manner. The table is searched for the closest arc length entry to the given arc length value, and the corresponding parametric entry is used to estimate the parametric value. This situation is a bit more complicated because the arc length entries are not evenly spaced; the table must actually be searched for the closest entry. Because the arc length entries are monotonically increasing, a binary search is an effective method of locating the closest entry. As before, once the closest arc length entry is found, the corresponding parametric entry can be used as the approximate parametric value, or the parametric entries corresponding to arc length entries on either side of the given arc length value can be used to linearly interpolate an estimated parametric value.

For example, if the task is to estimate the location of the point on the curve that is 0.75 unit of arc length from the beginning of the curve, the table is searched for the entry whose arc length is closest to that value. In this case the closest arc length is 0.72 and the corresponding parametric value is 0.40. For a better estimate, the values in the table that are on either side of the given distance are used to linearly interpolate a parametric value. In this case, the value of 0.75 is three-eighths of the way between the table values 0.72 and 0.80. Therefore, an estimate of the parametric value would be calculated as in Equation 3.11.

$$4.0 + \left(\frac{3}{8}\right) \cdot (4.5 - 4.0) = 4.1875 \qquad \textbf{(Eq. 3.11)}$$

The solution to the second problem (given an arc length $s$ and a parameter value $u_1$, find $u_2$ such that $LENGTH(u_1, u_2) = s$) can be found by using the table to estimate the arc length associated with $u_1$, adding that to the given value of $s$, and then using the table to estimate the parametric value of the resulting length.

The advantages of this approach are that it is easy to implement, intuitive, and fast to compute. The downside is that both the estimate of the arc length and the interpolation of the parametric value introduce errors into the calculation. These errors can be reduced in a couple of ways.

The curve can be supersampled to help reduce errors in forming the table. For example, ten thousand equally spaced values of the parameter could be used to construct a table consisting of a thousand entries by breaking down each interval into ten subintervals. This is useful if the curve is given beforehand and the table construction can be performed as a preprocessing step.

Better methods of interpolation can be used to reduce errors in estimating the parametric value. Instead of linear interpolation, higher-order interpolation procedures can be used in computing the parametric value. Of course, higher-order interpolants slow down the computation somewhat, so a decision about the speed/accuracy trade-off has to be made.

These techniques reduce the error somewhat blindly. There is no measure for the error in the calculation of the parametric value; these techniques only reduce the error globally instead of investing more computation in the areas of the curve in which the error is highest.

### Adaptive Approach

To better control error, an approach can be used that invests more computation in areas estimated to have large errors. As before, a table is to be constructed in which the first element of each entry is a parametric value and the second element of each entry is the arc length of the curve from its start to the position corresponding to that parametric value. (A linked list is an appropriate structure to hold this list because the entries will typically not be generated in sorted order and a sorted list is desired as the final result.) In addition, a sorted list of segments to be tested is maintained. A segment on the list is defined and sorted by a range of parametric values.

Initially the element <0, 0> is put in the table and the entire curve is put on the list of segments to be tested. The procedure operates on segments from the list to be tested until the list is empty. The first segment on the list is always the one tested next. The segment's midpoint is computed by evaluating the curve at the middle of the range of its parametric value. The curve is also evaluated at the endpoint of the segment; the position of the start of the segment is already in the table and can be retrieved from there. The length of the segment and the lengths of each of its halves are estimated by the linear distance between the points on the curve. The sum of the estimated lengths of the two halves of the segment is compared to the estimated length of the segment. If the difference between these two values is above some user-specified threshold, then both halves, in order, are added to the list of segments to be tested. If the values are within tolerance, then the parametric value of the midpoint is recorded in the table along with the arc length of the first point of the segment plus the distance from the first point to the midpoint. Also added to the table is the last parametric value of the segment along with the arc length to the midpoint plus the distance from the midpoint to the last point. When the list of segments to be tested becomes empty, a list of <parametric value, arc length> has been generated for the entire curve.

One problem with this approach is that at a premature stage in the procedure two half segments might indicate that the subdivision can be terminated (Figure 3.7). It is usually wise to force the subdivision down to a certain level and then embark on the adaptive subdivision.

Because the table has been adaptively built, it is not possible to directly compute the index of a given parametric entry as it was with the nonadaptive approach. Depending on the data structure used to hold the final table, either a sequential search or, possibly, a binary search must be used. Once the appropriate entries are found, then, as before, either a corresponding table entry can be used as an estimate for the value or entries can be interpolated to produce better estimates.

### Computing Arc Length Numerically

For cases in which efficiency of both storage and time are of concern, calculating the arc length numerically may be desirable. Calculating the length function involves evaluating the arc length integral (refer to Equation 3.3). Many numerical integration techniques approximate the integral of a function with the weighted sum of values of the function at various points in the interval of integration. Techniques such as Simpson's and trapezoidal integration use evenly spaced sample intervals. Gaussian quadrature [4] uses unevenly spaced intervals in an attempt to get the greatest accuracy using the smallest number of function evaluations. Because evaluation of the derivatives of the space curve accounts for most of the processing time in this algorithm, minimizing the number of evaluations is important for efficiency.

Gaussian quadrature, as commonly used, is defined over an integration interval from –1 to 1. The function to be integrated is evaluated at fixed points in the interval –1 to +1, and each function evaluation is multiplied by a precalculated weight. See Equation 3.12.

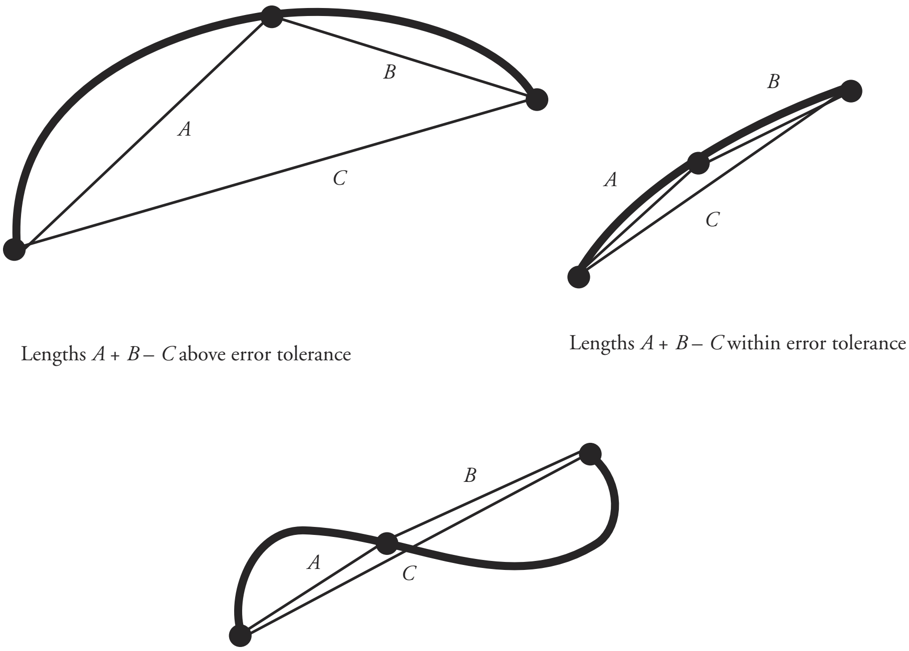
$$\int_{-1}^{1} f(u) = \sum_{i} w_i f(u_i) \qquad \text{(Eq. 3.12)}$$

A function, $g(t)$, defined over an arbitrary integration interval $(a, b)$ can be mapped to the interval [–1, 1] by the linear transformation shown in Equation 3.13 to give Equation 3.14, so that $g(t) = f(u)$.

$$u = (2 \cdot t - a - b)/(b - a) \qquad \text{(Eq. 3.13)}$$

$$\int_{a}^{b} g(t)dt = \int_{-1}^{1} f(u)du = \int_{-1}^{1} ((b-a)/2 \cdot g(((b-a) \cdot u + b + a)/2))du \qquad \text{(Eq. 3.14)}$$

The weights and evaluation points for different orders of Gaussian quadrature have been tabulated and can be found in many mathematical handbooks; see Appendix B.



Lengths $A + B - C$ above error tolerance

Lengths $A + B - C$ within error tolerance

Lengths $A + B - C$ erroneously report that the error is within tolerance

**Figure 3.7** Tests for adaptive subdivision

To perform Gaussian quadrature to compute the arc length of a cubic curve, Equation 3.5, Equation 3.6, and Equation 3.7 are used to define the arc length function shown in Equation 3.15. The actual code to carry this out is given below in the discussion on adaptive Gaussian integration.

$$\int_{-1}^{1} \sqrt{A \cdot u^4 + B \cdot u^3 + C \cdot u^2 + D \cdot u + E} \tag{Eq. 3.15}$$

### Adaptive Gaussian Integration

Some space curves have derivatives that vary rapidly in some areas and slowly in others. For such curves, Gaussian quadrature will either undersample some areas of the curve or oversample some other areas. In the former case, unacceptable error will accumulate in the result. In the latter case, time is wasted by unnecessary evaluations of the function. This is similar to what happens when using nonadaptive forward differencing.

To address this problem, an adaptive approach can be used [17]. It follows the same approach discussed in connection with arc length approximation by using adaptive forward differencing. Each interval is evaluated using Gaussian quadrature. The interval is then divided in half, and each half is evaluated using Gaussian quadrature. The sum of the two halves is compared with the value calculated for the entire interval. If the difference between these two values is less than the desired accuracy, then the procedure returns the sum of the halves; otherwise, as with the forward differencing approach, the two halves are added to the list of intervals to be subdivided.

Initially the length of the entire space curve is calculated using adaptive Gaussian quadrature. During this process, a table of the subdivision points is created. Each entry in the table is a pair $(u, s)$ where s is the arc length at parameter value $u$. When calculating $LENGTH(0, u)$, the table is searched to find the values $u_i$, $u_{i+1}$ such that $u_i \leq u \leq u_{i+1}$. The arc length from $u_i$ to $u$ is then calculated using nonadaptive Gaussian quadrature. This can be done because the space curve has been subdivided in such a way that nonadaptive Gaussian quadrature will give the required accuracy over the interval from $u_i$ to $u_{i+1}$. $LENGTH(0, u)$ is then equal to $s_i + LENGTH(u_i, u)$. $LENGTH(u_1, u_2)$ can be found by calculating $LENGTH(0, u_1)$ and $LENGTH(0, u_2)$ and then subtracting. The code for the adaptive integration and table-building procedure is shown in Figure 3.8.

This solves the first problem posed above; that is, given $u_1$ and $u_2$ find $LENGTH(u_1, u_2)$. To solve the second problem of finding $u$, which is a given distance, $s$, away from a given $u_1$, numerical root-finding techniques must be used.

```
/* -----------------------------------------------------------------------
STRUCTURES
*/

// the structure to hold entries of the table consisting of
// parameter value (u) and estimated length (length)
typedef struct table_entry_structure {
    double u,length;
} table_entry_td;

// the structure to hold an interval of the curve, defined by
// starting and ending parameter values and the estimated
// length (to be filled in by the adaptive integration
// procedure)
typedef struct interval_structure {
    double u1,u2;
    double length;
} interval_td;

// coefficients for a 2D cubic curve
typedef struct cubic_curve_structure {
    double ax,bx,cx,dx;
    double ay,by,cy,dy;
} cubic_curve_td;

// polynomial function structure; a quadric function is generated
// from a cubic curve during the arclength computation
typedef struct polynomial_structure {
    double *coeff;
    int    degree;
} polynomial_td;

/* -----------------------------------------------------------------------
ADAPTIVE INTEGRATION
this is the high-level call used whenever a curve's length is to be computed
*/
void adaptive_integration(cubic_curve_td *curve, double u1, double u2,
double tolerance)
{
 double subdivide();
 polynomial_td func;
 interval_td full_interval;
 double  total_length;
 double  integrate_func();
 double  temp;

 func.degree = 4;
 func.coeff = (double *)malloc(sizeof(double)*5);
 func.coeff[4] = 9*(curve->ax*curve->ax + curve->ay*curve->ay);
 func.coeff[3] = 12*(curve->ax*curve->bx + curve->ay*curve->by);
 func.coeff[2] = (6*(curve->ax*curve->cx + curve->ay*curve->cy) +
         4*(curve->bx*curve->bx + curve->by*curve->by)   );
 func.coeff[1] = 4*(curve->bx*curve->cx + curve->by*curve->cy);
 func.coeff[0] = curve->cx*curve->cx + curve->cy*curve->cy;
 full_interval.u1 = u1; full_interval.u2 = u2;
 temp = integrate_func(&func,&full_interval);
 printf("\nInitial guess = %lf; %lf:%lf",temp,u1,u2);
 full_interval.length = temp;
 total_length = subdivide(&full_interval,&func,0.0,tolerance);
 printf("\n total length = %lf\n",total_length);
}

/* ---------------------------------------------------------------------------
SUBDIVIDE
'total_length' is the length of the curve up to, but not including, the
'full_interval'
if the difference between the interval and the sum of its halves is less
than 'tolerance,'
    stop the recursive subdivision
'func' is a polynomial function
*/
double subdivide(interval_td *full_interval, polynomial_td *func,
       double total_length, double tolerance)
{

 interval_td left_interval, right_interval;
 double  left_length,right_length;
 double  midu;
 double  subdivide();
 double integrate_func();
 double  temp;
 void add_table_entry();

 midu = (full_interval->u1+full_interval->u2)/2;
 left_interval.u1 = full_interval->u1; left_interval.u2 = midu;
 right_interval.u1 = midu; right_interval.u2 = full_interval->u2;

 left_length = integrate_func(func, & left_interval);
 right_length = integrate_func(func, & right_interval);

 temp = fabs(full_interval->length - (left_length+right_length));
 if (temp > tolerance) {
  left_interval.length = left_length;
  right_interval.length = right_length;
  total_length = subdivide(&left_interval, func, total_length, tolerance);
  total_length = subdivide(&right_interval, func, total_length, tolerance);
  return(total_length);
 }
 else {
  total_length = total_length + left_length;
  add_table_entry(midu,total_length);
  total_length = total_length + right_length;
  add_table_entry(full_interval->u2,total_length);
  return(total_length);
 }
}
```

```
/* ----------------------------------------------------------------------
ADD TABLE ENTRY
adds an entry of the form (parametric value, length) to the table being
constructed
*/
void add_table_entry(double u, double length)
{
 /* add entry of (u, length) */
 printf("\ntable entry:  u: %lf, length: %lf",u,length);
}

/* ----------------------------------------------------------------------
INTEGRATE FUNCTION
use Gaussian quadrature to integrate square root of given function in the
given interval
*/
double integrate_func(polynomial_td *func,interval_td *interval)
{
 double x[5]={.1488743389,.4333953941,.6794095682,.8650633666,
.9739065285};
 double w[5]={.2966242247,.2692667193,.2190863625,.1494513491,.0666713443};
 double length, midu, dx, diff;
 int    i;
 double  evaluate_polynomial();
 double  u1,u2;

 u1 = interval->u1;
 u2 = interval->u2;

 midu = (u1+u2)/2.0;
 diff = (u2-u1)/2.0;
 length = 0.0;
 for (i=0; i<5; i++) {
  dx = diff*x[i];
  length += w[i]*(sqrt(evaluate_polynomial(func,midu+dx)) +
          sqrt(evaluate_polynomial(func,midu-dx)));
 }
 length *= diff;

 return (length);
}

/* ----------------------------------------------------------------------
EVALUATE POLYNOMIAL
evaluate a polynomial
 */
double evaluate_polynomial(polynomial_td *poly, double u)
{
 double  w;
 int     i;
 double  value;
 value = 0.0;
 w = 1.0;
 for (i=0; i<=poly->degree; i++) {
  value += poly->coeff[i]*w;
  w *= u;
 }
 return value;
}
```

**Figure 3.8** Adaptive Gaussian integration of arc length

### Finding *u* Given *s*

The solution of the equation $s - LENGTH(u_1, u) = 0$ gives the value of *u* that is at arc length *s* from the point $R(u_1)$. Since the arc length is a strictly monotonically increasing function of *u,* the solution is unique provided that the length of $dR(u)/du$ is not identically 0 over some integral. Newton-Raphson iteration can be used to find the root of the equation because it converges rapidly and requires very little calculation at each iteration. Newton-Raphson iteration consists of generating the sequence $\{p_n\}$ as in Equation 3.16.

$$p_n = p_{n-1} - f(p_{n-1})/f'(p_{n-1}) \qquad \text{(Eq. 3.16)}$$

In this case, $f$ is equal to $s - LENGTH(u_1, P_{n-1}) = 0$ and can be evaluated at $p_{n-1}$ using the techniques discussed above for computing arc length; $f'$ is $dP/du$ evaluated at $p_{n-1}$. Two problems may be encountered with Newton-Raphson iteration: Some of the $p_n$ may not lie on the space curve at all; $dR/du$ may be zero, or very nearly zero, at some points on the space curve. The first problem will cause all succeeding elements $p_{n+1}, p_{n+2}, \ldots$ to be undefined, while the latter problem will cause division by zero or by a very small number. A zero parametric derivative is most likely to arise when two or more control points are placed in the same position. This can easily be detected by examining the derivative of $f$ in Equation 3.16. If it is small or zero, then binary subdivision is used instead of Newton-Raphson iteration. Binary subdivision can be used in a similar way to handle the case of undefined $p_n$. When finding u such that $LENGTH(0, u) = s$, the subdivision table is searched to find the values $s_i$, $s_{i+1}$ such that $s_i \leq s \leq s_{i+1}$. The solution $u$ lies between $u_i$ and $u_{i+1}$. Newton-Raphson iteration is then applied to this subinterval. An initial approximation to the solution is made by linearly interpolating *s* between the endpoints of the interval and using this as the first iterate. Although Newton-Raphson iteration can diverge under certain conditions, it does not happen often in practice. Since each step of Newton-Raphson iteration requires evaluation of the arc length integral, eliminating the need to do adaptive Gaussian quadrature, the result is a significant increase in speed. At this point it is worth noting that the integration and root-finding procedures are completely independent of the type of space curve used. The only procedure that needs to be modified to accommodate new curve types is the derivative calculation subroutine, which is usually a short program.

### 3.2.2 Speed Control

Once a space curve has been parameterized by arc length, it is possible to control the speed at which the curve is traversed. Stepping along the curve at equally spaced intervals of arc length will result in a constant speed traversal. More interesting traversals can be generated by speed control functions that relate an equally spaced parametric value (e.g., *time*) to arc length in such a way that a controlled traversal of the curve is generated. The most common example of such speed control is *ease-in/ease-out* traversal. This type of speed control produces smooth motion as an object accelerates from a stopped position, reaches a maximum velocity, and then decelerates to a stopped position.

In this discussion, the speed control function's input parameter is referred to as *t,* for *time,* and its output is *arc length,* referred to as *distance* or simply as *s*. Constant-velocity motion along the space curve can be generated by evaluating it at equally spaced values of its arc length where arc length is a linear function of *t.* In practice, it is usually easier if, once the space curve has been reparameterized by arc length, the parameterization is then normalized so that the parametric variable varies between zero and one as it traces out the space curve; the *normalized arc length parameter* is just the arc length parameter divided by the total arc length of the curve. For this discussion, the normalized arc length parameter will still be referred to simply as the arc length.

Speed along the curve can be controlled by varying the arc length values at something other than a linear function of *t;* the mapping of time to arc length is independent of the form of the space curve itself. For example, the space curve might be linear, while the arc length parameter is controlled by a cubic function with respect to time. If $t$ is a parameter that varies between 0 and 1 and if the curve has been parameterized by arc length and normalized, then ease-in/ease-out can be implemented by a function $s(t) = ease(t)$ so that as $t$ varies uniformly between 0 and 1, $s$ will start at 0, slowly increasing in value and gaining speed until the middle values and then decelerating as it approaches 1. See Figure 3.9. Variable $s$ is then used as the interpolation parameter in whatever function produces spatial values.

The control of motion along a parametric space curve will be referred to as *speed control.* Speed control can be specified in a variety of ways and at various levels of complexity. But the final result is to produce, either explicitly or implicitly, a distance-time function $s(t)$, which, for any given time *t,* produces the distance

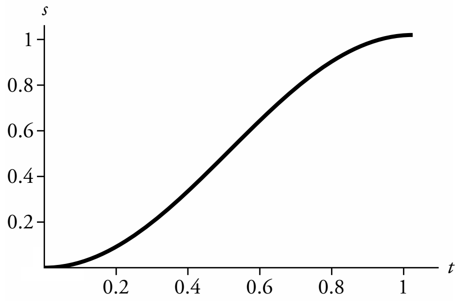


**Figure 3.9** *Ease*($t$)

traveled along the space curve (arc length). The space curve defines *where* to go, while the distance-time function defines *when*.

Such a function $s(t)$ would be used as follows. At a given time $t$, $s(t)$ is the desired distance to have traveled along the curve at time $t$. An arc length table (see Section 3.2.1 on computing arc length) can then be used to find the corresponding parametric value $u$ that corresponds to that arc length. The space curve is then evaluated at $u$ to produce a point on the space curve. The arc length of the curve segment from the beginning of the curve to the point is equal to $s(t)$ (within some tolerance). If, for example, this position is used to translate an object through space at each time step, it translates along the path of the space curve at a speed indicated by $s(t)$.

There are various ways in which the distance-time function can be specified. It can be explicitly defined by giving the user graphical curve-editing tools. It can be specified analytically. It can also be specified by letting the user define the velocity-time curve, or by defining the acceleration-time curve. The user can even work in a combination of these spaces. In the following discussion, the common assumption is that the entire arc length of the curve is to be traversed during the given total time. Some additional optional assumptions (constraints) that are typically used are listed below. In certain situations, it may be desirable to violate these constraints.

1. The distance-time function should be monotonic in $t$—that is, the curve should be traversed without backing up along the curve.
2. The distance-time function should be continuous—that is, there should be no instantaneous jumps from one point on the curve to a nonadjacent point on the curve.

Following the assumption stated above, the entire space curve is to be traversed in the given total time. This means that $0.0 = s(0.0)$ and t*otal_distance* = $s($*total_time*$)$. As mentioned previously, the distance-time function may also be normalized so that all such functions end at $1.0 = s(1.0)$. Normalizing the

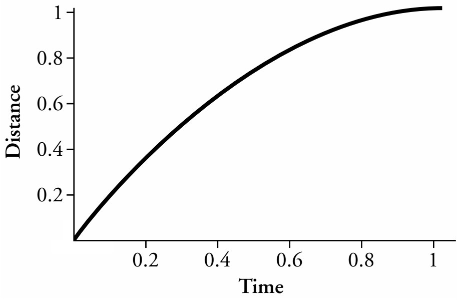


**Figure 3.10** Graph of sample analytic distance-time function $(2 - t)^*t$

distance-time function facilitates its reuse with other space curves. An example of an analytic definition is $d(t) = (2 - t)^*t$, although this does not have the shape characteristic of ease-in/ease-out motion control. See Figure 3.10.

## 3.2.3 Ease-in/Ease-out

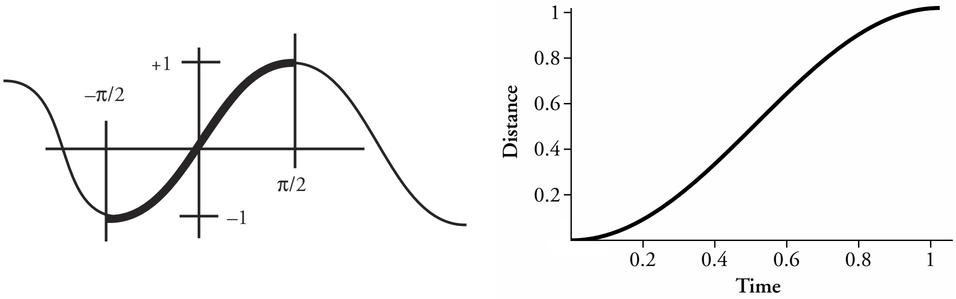
*Ease-in/ease-out* is one of the most useful and most common ways to control motion along a curve. There are several ways to incorporate ease-in/ease-out control. The standard assumption is that the motion starts and ends in a complete stop and that there are no instantaneous jumps in velocity (first-order continuity). There may or may not be an intermediate interval of constant speed, depending on the technique used to generate the speed control. The speed control function will be referred to as $s(t) = ease(t)$, where $t$ is a uniformly varying input parameter meant to represent time and $s$ is the output parameter that is the distance (arc length) traveled as a function of time.

### Sine Interpolation

One easy way to implement ease-in/ease-out is to use the section of the sine curve from –π/2 to +π/2 as the ease(t) function. This is done by mapping the parameter values of 0 to +1 into the domain of –π/2 to +π/2 and then mapping the corresponding range of the sine functions of –1 to +1 in the range 0 to 1. See Equation 3.17 and the corresponding Figure 3.11.

$$s(t) = ease(t) = \frac{\sin\left(t \cdot \pi - \frac{\pi}{2}\right) + 1}{2} \qquad \text{(Eq. 3.17)}$$

In this function, $t$ is the input that is to be varied uniformly from zero to one (e.g., 0.0, 0.1, 0.2, 0.3, . . .). The output, $s$, also goes from zero to one but does so



Sine curve segment to use as ease-in/ease-out control

Sine curve segment mapped to useful values

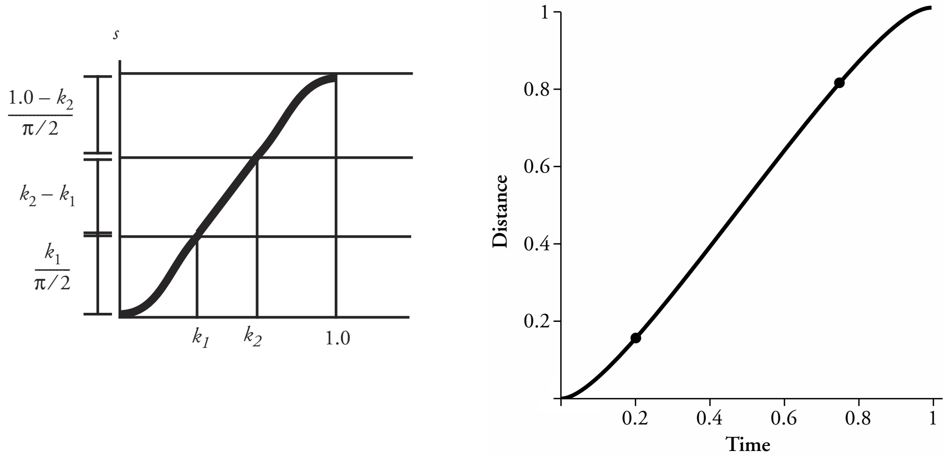
**Figure 3.11** Using a sine curve segment as the ease-in/ease-out distance-time function

by starting off slowly, speeding up, and then slowing down. For example, at $t = .25$, $s(t) = .1465$, and at $t = .75$, $s(t) = .8535$. With the sine/cosine ease-in/ease-out function presented here, the "speed" of $s$ with respect to $t$ is never constant over an interval but rather is always accelerating or decelerating, as can be seen by the ever-changing slope of the curve. Notice that the derivative of this ease function is zero at $t = 0$ and at $t = 1$. Zero derivatives at zero and one indicate a smooth acceleration from a stopped position at the beginning of the motion and a smooth deceleration to a stop at the end.

### Using Sinusoidal Pieces for Acceleration and Deceleration

To provide an intermediate section of the distance-time function that has constant speed, pieces of the sine curve can be constructed at each end of the function with a linear intermediate segment. Care must be taken so that the tangents of the pieces line up to provide first-order continuity. There are various ways to approach this, but as an example assume the user specifies fragments of the unit interval that should be devoted to acceleration and deceleration. For example, the user may specify the value of 0.3 for acceleration and 0.75 for deceleration. Acceleration occurs from time zero to 0.3 and deceleration occurs from time 0.75 to the end of the interval. Referring to the user-specified values as $k_1$ and $k_2$ respectively, a sinusoidal curve segment is used to implement an acceleration from time 0 to $k_1$. A sinusoidal curve is also used for velocity to implement deceleration from time $k_2$ to 1. Between times $k_1$ and $k_2$, a constant velocity is used.

The solution is formed by piecing together a sine curve segment from –π/2 to zero with a straight line segment (indicating constant speed) inclined at 45 degrees (because the slope of the sine curve at zero is equal to one) followed by a sine curve segment from zero to π/2. The initial sine curve segment is uniformly scaled by $k_1/\pi/2$ so that the length of its domain is $k_1$. The length of the domain of the line



Ease-in/ease-out curve as it is initially pieced together

Curve segments scaled into useful values with points marking segment junctions

**Figure 3.12** Using sinusoidal segments with constant speed intermediate interval

segment is $k_2 - k_1$. And the final sine curve segment is uniformly scaled by $1.0 - k_2/\pi/2$ so that the length of its domain is $1.0 - k_2$. The sine curve segments must be uniformly scaled to preserve $C^1$ (first-order) continuity at the junction of the sine curves with the straight line segment. This means that for the first sine curve segment to have a domain from zero to $k_1$, it must have a range of length $k_1/\pi/2$; similarly, the ending sine curve segment must have a range of length $1.0 - k_2/\pi/2$. The middle straight line segment, with a slope of one, will travel a distance of $k_2 - k_1$.

To normalize the distance traveled, the resulting three-segment curve must be scaled down by a factor equal to the total distance traveled as computed by $k_1/\pi/2 + k_2 - k_1 + 1.0 - k_2/\pi/2$. See Figure 3.12.

The ease function described above and shown in Equation 3.18 is implemented in the code of Figure 3.13.

$$ease(t)\begin{cases} = \left(k_1 \cdot \dfrac{2}{\pi} \cdot \left(\sin\left(\dfrac{t}{k_1} \cdot \dfrac{\pi}{2} - \dfrac{\pi}{2}\right) + 1\right)\right)/f & t \le k_1 \\ = \left(\dfrac{k_1}{\pi/2} + t - k_1\right)/f & k_1 \le t \le k_2 \\ = \left(\dfrac{k_1}{\pi/2} + k_2 - k_1 + \left((1 - k_2) \cdot \dfrac{2}{\pi}\right)\sin\left(\dfrac{t - k_2}{1.0 - k_2} \cdot \dfrac{\pi}{2}\right)\right)/f & k_2 \le t \end{cases}$$

where $f = k_1 \cdot 2/\pi + k_2 - k_1 + (1.0 - k_2) \cdot 2/\pi$ **(Eq. 3.18)**

```
float ease(float t, float k1, float k2)
{
float t1,t2;
float f,s;

f = k1*2/PI + k2 - k1 + (1.0-k2)*2/PI;

if (t < k1) {
s = k1*(2/PI)*(sin((t/k1)*PI/2-PI/2)+1);
}
else if (t < k2) {
s = (2*k1/PI + t-k1);
}
else {
s= 2*k1/PI + k2-k1 + ((1-k2)*(2/PI))*sin(((t-k2)/(1.0-k2))*PI/2);
}
return (s/f);
```

**Figure 3.13** Code to implement ease-in/ease-out using sinusoidal ease-in, followed by constant velocity and sinusoidal ease-out

## 3.2.4 Constant Acceleration: Parabolic Ease-In/Ease-Out

To avoid the transcendental function evaluation (sin and cos), or corresponding table look-up and interpolation, an alternative approach for the ease function is to establish basic assumptions about the acceleration that in turn establish the basic form that the velocity-time curve can assume. The user can then set parameters to specify a particular velocity-time curve that can be integrated to get the resulting distance-time function.

The simple case of no ease-in/ease-out would produce a constant zero acceleration curve and a velocity curve that is a horizontal straight line at some value $v_0$ over the time ($t$) interval from 0 to total time. The actual value of $v_0$ depends on the distance covered and is computed using the relationship *distance = speed * time* so that $v_0$*total_time=distance_covered*. In the case where normalized values of 1.0 are being used for total distance covered and total time, $v_0 = 1.0$. See Figure 3.14.

The distance-time curve is equal to the integral of the velocity-time curve and relates time and distance along the space curve through a function $s(t)$. Similarly, the velocity-time curve is equal to the integral of the acceleration-time curve and relates time and velocity along the space curve.

To implement an ease-in/ease-out function, constant acceleration and deceleration at the beginning and end of the motion and zero acceleration during the middle of the motion are assumed. The assumptions of beginning and ending with stopped positions mean that the velocity starts out at zero and ends at zero. In order for this to be reflected in the acceleration/deceleration curve, the area under the curve marked “acc” must be equal to the area above the curve marked “dec,” but the actual values of the acceleration and deceleration do not have to be equal

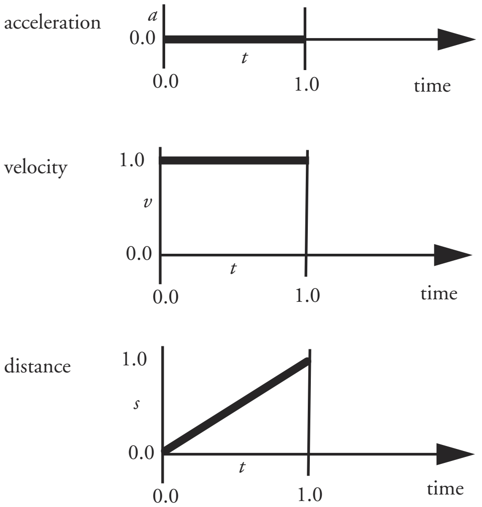
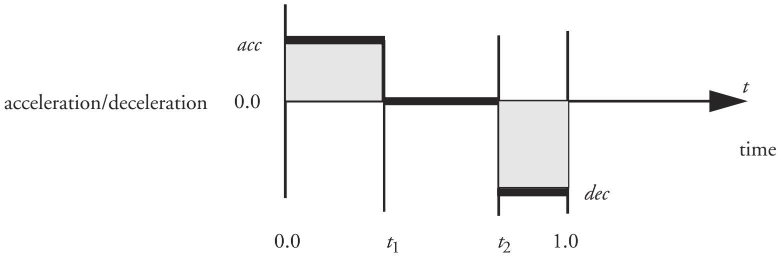


**Figure 3.14** Acceleration, velocity, and distance curves for constant speed

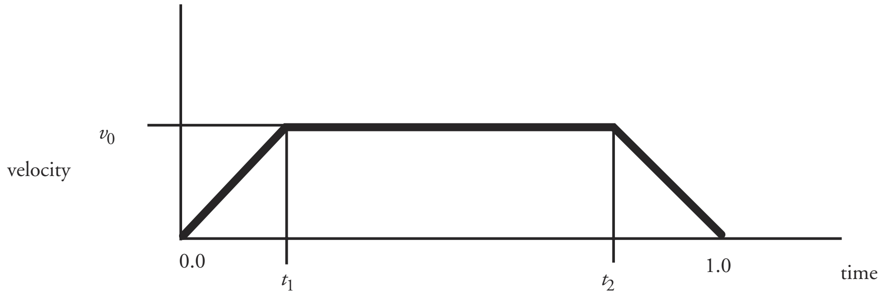


| | |
|---|---|
| $a = acc$ | $0 < t < t_1$ |
| $a = 0.0$ | $t_1 < t < t_2$ |
| $a = dec$ | $t_2 < t < 1.0$ |

**Figure 3.15** Acceleration/deceleration graph

to each other. Thus, three of the four variables (*acc, dec,* $t_1$, $t_2$) can be specified by the user and the system can solve for the fourth to enforce the constraint of equal areas. See Figure 3.15.

This piecewise constant acceleration function can be integrated to obtain the velocity function. The resulting velocity function has a linear ramp for accelerating, followed by a constant velocity interval, and ends with a linear ramp for deceleration (see Figure 3.16). The integration introduces a constant into the velocity function, but this constant is zero under the assumption that the velocity starts out at zero and ends at zero. The constant velocity attained during the middle interval



$$v = v_0 \cdot \frac{t}{t_1} \qquad 0.0 < t < t_1$$

$$v = v_0 \qquad t_1 < t < t_2$$

$$v = v_0 \cdot \left(1.0 - \frac{t - t_2}{1.0 - t_2}\right) \qquad t_2 < t < 1.0$$

**Figure 3.16** Velocity-time curve

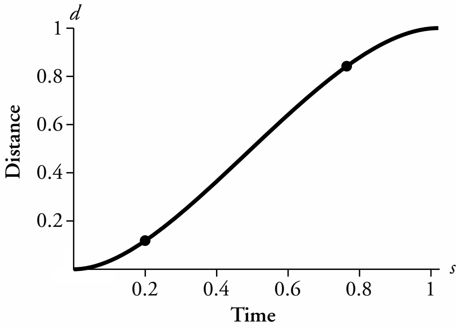
depends on the total distance that must be covered during the time interval; the velocity is equal to the area below (above) the *acc* (*dec*) segment in Figure 3.15. In the case of normalized time and normalized distance covered, the total time and total distance are equal to one. The total distance covered will be equal to the area under the velocity curve (Figure 3.16). The area under the velocity curve can be computed as in Equation 3.19.

1.0 = area under leading linear ramp
+ area under middle constant velocity interval
+ area under ending linear ramp

$$1.0 = \frac{1}{2} \cdot v_0 \cdot t_1 + v_0 \cdot (t_2 - t_1) + \frac{1}{2} \cdot v_0 \cdot (1.0 - t_2) \qquad \text{(Eq. 3.19)}$$

Because integration introduces arbitrary constants, the acceleration-time curve does not bear any direct relation to total distance covered. Therefore, it is often more intuitive for the user to specify ease-in/ease-out parameters using the velocity-time curve. In this case, the user can set two of the three variables, $t_1$, $t_2$, $v_0$, and the system can solve for the third in order to enforce the "total distance covered" constraint. For example, if the user specifies the time over which acceleration and deceleration take place, then the maximum velocity can be found by using Equation 3.20.

$$v_0 = \frac{2.0}{(t_2 - t_1 + 1.0)} \qquad \text{(Eq. 3.20)}$$



$$d = v_0 \cdot \frac{t^2}{2 \cdot t_1} \qquad 0.0 < t < t_1$$

$$d = v_0 \cdot \frac{t_1}{2} + v_0 \cdot (t - t_1) \qquad t_1 < t < t_2$$

$$d = v_0 \cdot \frac{t_1}{2} + v_0 \cdot (t_2 - t_1) + \left( v_0 - \frac{\left( v_0 \cdot \frac{t - t_2}{1 - t_2} \right)}{2} \right) \cdot (t - t_2) \qquad t_2 < t < 1.0$$

**Figure 3.17** Distance-time function

The velocity-time function can be integrated to obtain the final distance-time function. Once again, the integration introduces an arbitrary constant, but, with the assumption that the motion begins at the start of the curve, the constant is constrained to be zero. The integration produces a parabolic ease-in segment, followed by a linear segment, followed by a parabolic ease-out segment (Figure 3.17).

The methods for ease control based on the sine function are easy to implement and use but are less flexible than the acceleration-time and velocity-time functions. These latter functions allow the user to have more control over the final motion because of the ability to set various parameters.

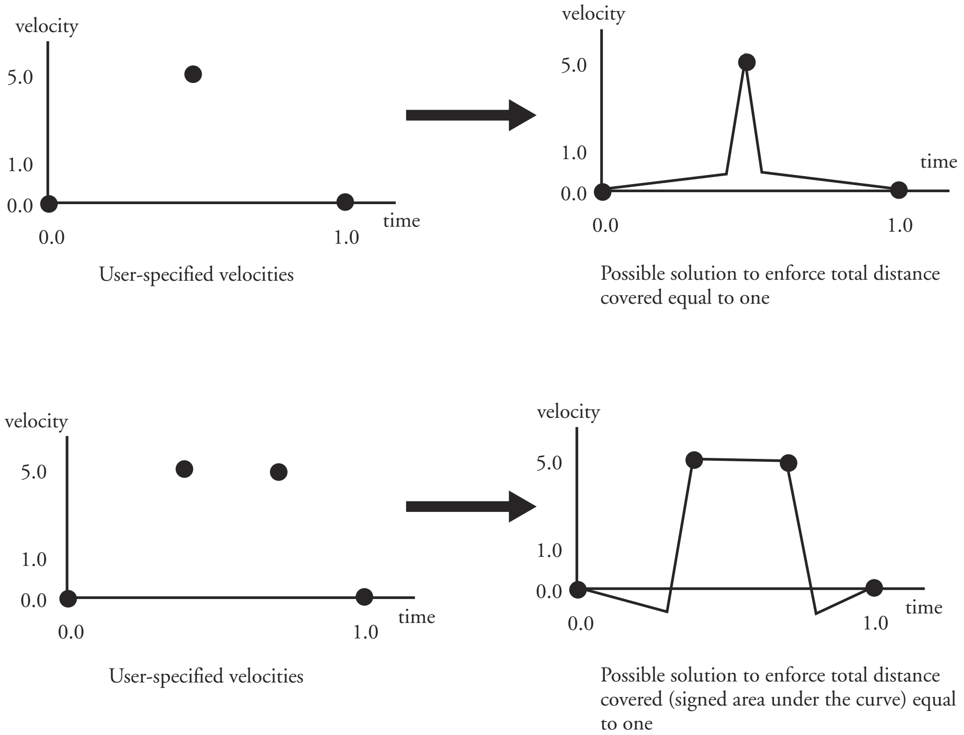
## 3.2.5 General Distance-Time Functions

When working with velocity-time curves or acceleration-time curves, one finds that the underlying assumption that the total distance is to be traversed during the total time presents some interesting issues. Once the total distance and total time are given, the average velocity is fixed. This average velocity must be maintained as the user modifies, for example, the shape of the velocity-time curve. This can create a problem if the user is working only with the velocity-time curve. One solution is to let the absolute position of the velocity-time curve float along the velocity axis as the user modifies the curve. The curve will be adjusted up or down in absolute velocity in order to maintain the correct average velocity. However, this means that if the user wants to specify certain velocities, such as starting and ending velocities, or a maximum velocity, then other velocities must change in response in order to maintain total distance covered.

An alternative way to specify speed control is to fix the absolute velocities at key points and then change the interior shape of the curve to compensate for average velocity. However, this may result in unanticipated (and undesirable) changes in the shape of the velocity-time curve. Some combinations of values may result in unnatural spikes in the velocity-time curve in order to keep the area under the curve equal to the total distance (one, in the case of normalized distance). Consequently, undesirable accelerations may be produced, as demonstrated in Figure 3.18.

Notice that negative velocities mean that a point traveling along the space curve backs up along the curve until the time is reached when velocity becomes positive again. Usually, this is not acceptable behavior.

The user may also work directly with the distance-time curve. For some users, this is the most intuitive approach. Assuming that a point on the curve starts at the beginning of the curve at $t = 0$ and traversesthe curve to arrive at the end of the curve at $t = 1.0$, then the distance-time curve is constrained to start at (0, 0) and



User-specified velocities

Possible solution to enforce total distance covered equal to one

User-specified velocities

Possible solution to enforce total distance covered (signed area under the curve) equal to one

**Figure 3.18** Some nonintuitive results of user-specified values on the velocity-time curve

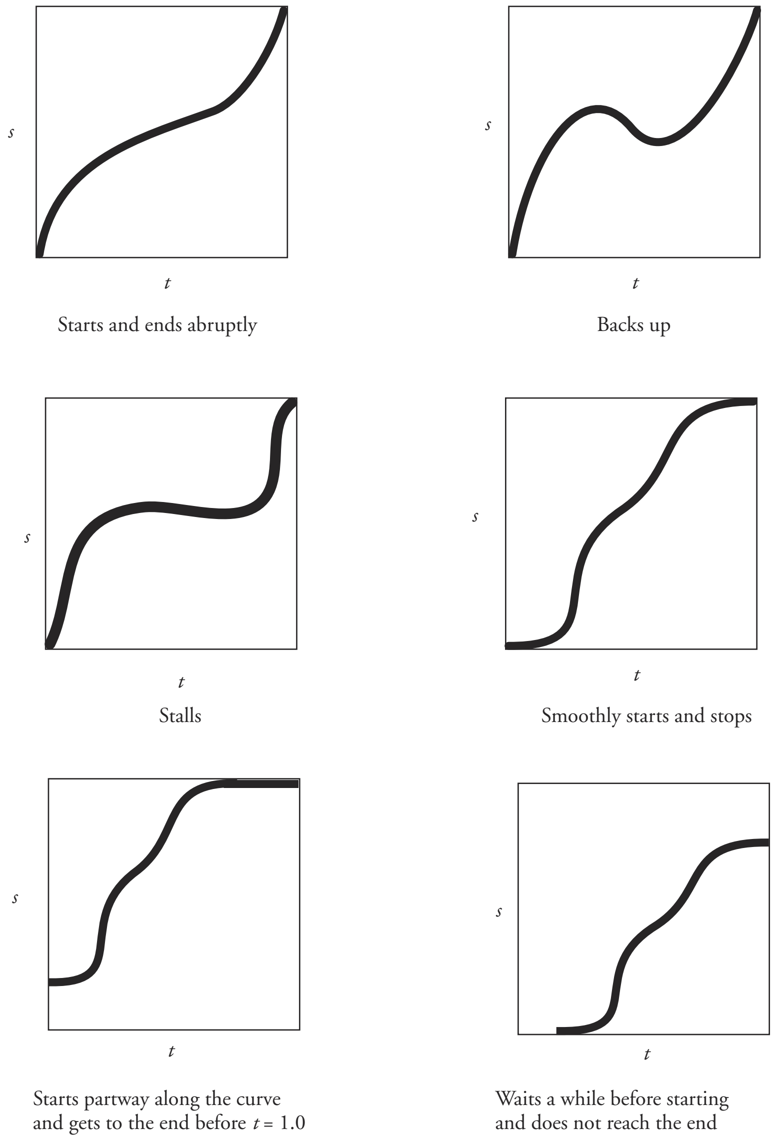
end at (1.0, 1.0). If the objective is to start and stop with zero velocity, then the slope at the start and end should be zero. The restriction that a point traveling along the curve may not back up anytimeduring the traversal means that the distance-time curve must be monotonically increasing (i.e., always have a nonnegative slope). If the point may not stop along the curve during the time interval, then there can be no horizontal sections in the distance-time curve (no internal zero slopes). See Figure 3.19.

As mentioned before, the space curve that defines the path along which the object moves is independent of the speed control curves that define the relative velocity along the path as a function of time. A single space curve could have several velocity-time curves defined along it, and one distance-time curve, for example, a standard ease-in/ease-out function, could be applied to several different space curves. Reusing distance-time curves is facilitated if normalized distance and time are used.

Motion control frequently requires specifying positions and speeds along the space curve at specific times. An example might be specifying the motion of a hand as it reaches out to grasp an object; initially the hand accelerates toward the object, and then, as it comes close, it slows down to almost zero speed before picking up the object. The motion is specified as a sequence of constraints on time, position (in this case, arc length traveled along a space curve), velocity, and acceleration. Stating the problem more formally, each point to be constrained is an n-tuple, $<t_i, s_i, v_i, a_i, \ldots ,>$, where $s_i$ is position, $v_i$ is velocity, $a_i$ is acceleration, and $t_i$ is the time at which all the constraints must be satisfied (the ellipses, . . . , indicate that higher-order derivatives may be constrained). Define the zero-order constraint problem to be that of satisfying sets of two-tuples, $<t_i, s_i>$, while velocity, acceleration, and so on are allowed to take on any values necessary to meet the position constraints at the specified times. Zero-order constrained motion is illustrated at the top of Figure 3.20. Notice that there is continuity of position but not of speed. By extension, the first-order constraint problem requires satisfying sets of three-tuples, $<s_i, v_i, t_i>$, as shown in the bottom illustration in Figure 3.20. Standard interpolation techniques (see Appendix B) can be used to aid the user in generating distance-time curves.

### 3.2.6 Curve Fitting to Position-Time Pairs

If the animator has specific positional constraints that must be met at specific times, then the time-parameterized space curve can be determined directly. Position-time pairs can be specified by the animator, as in Figure 3.21, and the control points of the curve that produce an interpolating curve can be computed from this information [35].



Starts and ends abruptly

Backs up

Stalls

Smoothly starts and stops

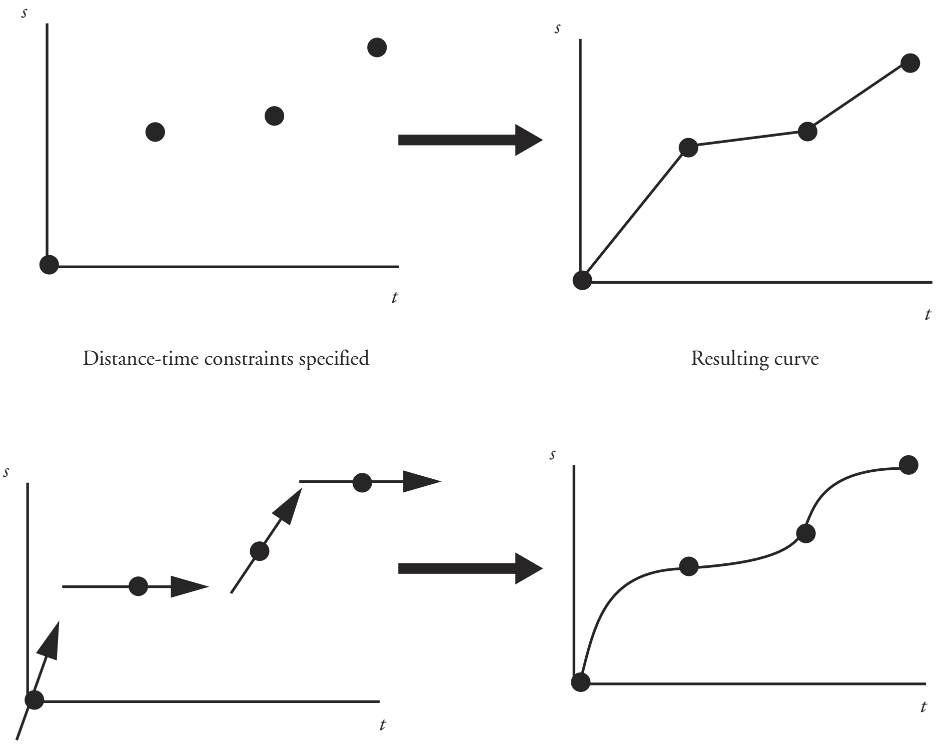
Starts partway along the curve
and gets to the end before $t = 1.0$

Waits a while before starting
and does not reach the end

**Figure 3.19** Sample distance-time functions

For example, consider the case of fitting a B-spline[1] curve to values of the form $P_i$, $t_i$), $i = 1, \ldots, j$. Given the form of B-spline curves shown in Equation 3.21 of order $k$ with $n + 1$ defining control vertices, and expanding in terms of the $j$ given

[1] Refer to Appendix B for more information on B-spline curves.



Distance-time constraints specified

Resulting curve

Velocity-distance-time constraints specified

Resulting curve

**Figure 3.20** Specifying motion constraints

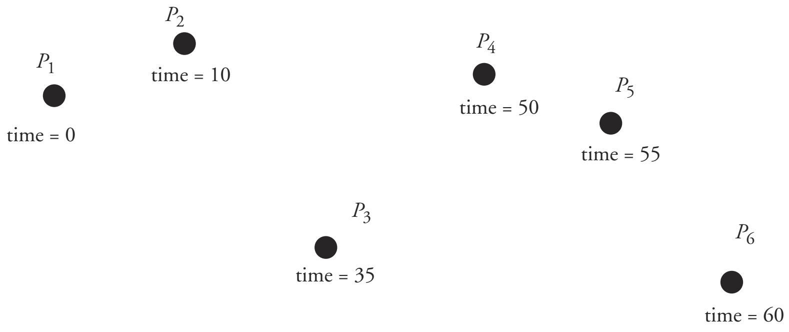


**Figure 3.21** Position-time constraints

constraints ($2 \leq k \leq n + \leq j$), Equation 3.22 results. Put in matrix form, it becomes Equation 3.23, in which the given points are in the column matrix *P,* the unknown defining control vertices are in the column matrix *B,* and *N* is the matrix of basis functions evaluated at the given times ($t_1, t_2, \ldots, t_j$).

$$P(t) = \sum_{i=1}^{n+1} B_i \cdot N_{i,k}(t) \qquad \text{(Eq. 3.21)}$$

$$\begin{aligned}
P_1 &= N_{1,k}(t_1) \cdot B_1 + N_{2,k}(t_1) \cdot B_2 + \ldots + N_{n+1,k}(t_1) \cdot B_{n+1} \\
P_2 &= N_{1,k}(t_2) \cdot B_1 + N_{2,k}(t_2) \cdot B_2 + \ldots + N_{n+1,k}(t_2) \cdot B_{n+1} \\
&\ldots \\
P_j &= N_{1,k}(t_j) \cdot B_1 + N_{2,k}(t_j) \cdot B_2 + \ldots + N_{n+1,k}(t_j) \cdot B_{n+1}
\end{aligned} \qquad \text{(Eq. 3.22)}$$

$$P = N \cdot B \qquad \text{(Eq. 3.23)}$$

If there are the same number of given data points as there are unknown control points, $2 \le k \le n+1 = j$, then $N$ is square and the defining control vertices can be solved by inverting the matrix, as in Equation 3.24.

$$B = N^{-1} \cdot P \qquad \text{(Eq. 3.24)}$$

The resulting curve is smooth but can sometimes produce unwanted wiggles. Specifying fewer control points ($2 \le k \le n+1 < j$) can remove these wiggles but $N$ is no longer square. To solve the matrix equation, the pseudoinverse of $N$ must be used, as in Equation 3.25.

$$\begin{aligned}
P &= N \cdot B \\
N^T \cdot P &= N^T \cdot N \cdot B \\
[N^T \cdot N]^{-1} \cdot N^T \cdot P &= B
\end{aligned} \qquad \text{(Eq. 3.25)}$$

## 3.3 Interpolation of Rotations Represented by Quaternions

Quaternions, as discussed in Chapter 2, are useful for representing orientations. One of the most important reasons for choosing quaternions is that they can be easily interpolated and they avoid the effects of gimbal lock, which can trouble the other commonly used representations, fixed angles and Euler angles. While quaternion representations of orientations can be interpolated to produce reasonable intermediate orientations, direct linear interpolation of the individual quantities of the quaternion four-tuples produces nonlinear motion. To avoid effects of magnitude on interpolation, unit quaternions are typically used to represent orientations. Unit quaternions can be considered as points on the unit sphere in four-dimensional space.

Given two orientations represented by unit quaternions, intermediate orientations can be produced by linearly interpolating from the first to the second. These orientations can be viewed as four-dimensional points on a straight-line path from the first quaternion to the second quaternion. Simple equal-interval, linear interpolation between the two quaternions will not produce a constant speed rotation because the unit quaternions to which the intermediate orientations map will not produce equally spaced intervals on the unit sphere. Figure 3.22 shows the analogous effect when interpolating a straight-line path between points on a 2D circle.

Intermediate orientations representing constant-speed rotation can be calculated by interpolating directly on the surface of the unit sphere, specifically along the great arc between the two quaternion points. A quaternion, $[s, v]$, and its negation, $[-s, -v]$, represent the same orientation. Interpolation from one orientation, represented by the quaternion $q_1$, to another orientation, represented by the quaternion $q_2$, can also be carried out from $q_1$to $-q_2$. The difference is that one interpolation path will be longer than the other. Usually, the shorter path is the more desirable because it represents the more direct way to get from one orientation to the other. The shorter path is the one indicated by the smaller angle between the 4D quaternion vectors. This can be determined by using the 4D dot product of the quaternions to compute the cosine of the angle between $q_1$ and $q_2$ (Equation 3.26). If the cosine is positive, then the path from $q_1$ to $q_2$ is shorter; otherwise the path from $q_1$ to $-q_2$ is shorter (Figure 3.23).

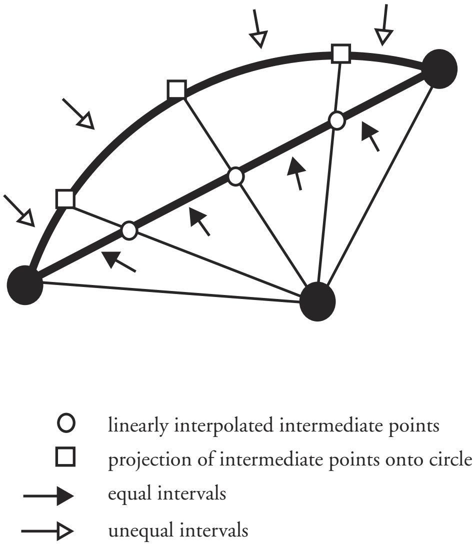

○ linearly interpolated intermediate points
□ projection of intermediate points onto circle
⟶ equal intervals
▷ unequal intervals

**Figure 3.22** Equally spaced linear interpolations of straight-line path between two points on a circle generate unequal spacing of points after projecting onto a circle

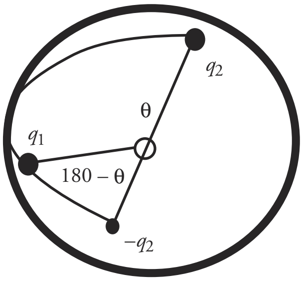


**Figure 3.23** The closer of the two representations of orientation is the better choice to use in interpolation

$$\cos\theta = q_1 \bullet q_2 = s_1 \cdot s_2 + v_1 \bullet v_2 \qquad \text{(Eq. 3.26)}$$

The formula for spherical linear interpolation (*slerp*) between unit quaternions $q_1$ and $q_2$ with parameter $u$ varying from 0 to 1 is given in Equation 3.27, where $q_1 \bullet q_2 = \cos\theta$. Notice that this does not necessarily produce a unit quaternion, so the result must be normalized if a unit quaternion is desired.

$$\begin{aligned} &\mathit{slerp}\,(q_2, q_2, u) \\ &= ((\sin((1-u)\cdot\theta))/(\sin\theta))\cdot q_1 + (\sin(u\cdot\theta))_{xu}/(\sin\theta)\cdot q_2 \end{aligned} \qquad \text{(Eq. 3.27)}$$

Notice that in the case $u = 1/2$, $\mathit{slerp}(q_1, q_2, u)$ can be easily computed within a scale factor, as $q_1 + q_2$, which can then be normalized to produce a unit quaternion.

When interpolating between a series of orientations, slerping between points on the surface of a sphere has the same problem as linear interpolation between points in Euclidean space: that of first-order discontinuity (see Appendix B). Shoemake [37] suggests using cubic Bezier interpolation to smooth the interpolation between orientations. In his paper, reasonable interior control points are automatically calculated to define cubic segments between each pair of orientations.

To discuss this technique, assume for now that there is a need to interpolate between two-dimensional points in Euclidean space [. . . , $p_{n-1}$, $p_n$, $p_{n+1}$, . . .]; these will be referred to as the *interpolated points*. (How to consider the calculations using quaternion representations is discussed later.) Between each pair of points, two control points will be constructed. For each of the interpolation points, $p_n$, two control points will be associated with it: the one immediately before it, $b_n$; and the one immediately after it, $a_n$.

To calculate the control point following any particular point $p_n$, take the vector defined by $p_{n-1}$ to $p_n$ and add it to the point $p_n$. Now take this point (marked "1" in Figure 3.24) and find the average of it and $p_{n+1}$. This becomes one of the control points (marked "$a_n$" in Figure 3.24).

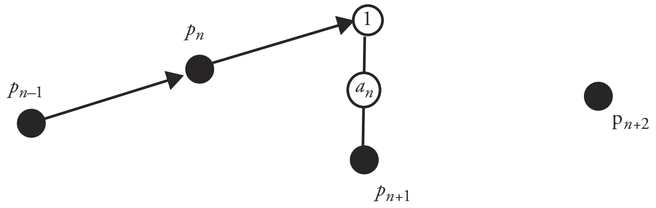


**Figure 3.24** Finding the control point after $p_n$

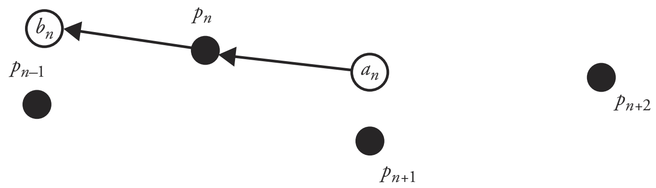


**Figure 3.25** Finding the control point before $p_n$

Next, take the vector defined by $a_n$ to $p_n$ and add it to $p_n$ to get $b_n$ (Figure 3.25). Points $b_n$ and $a_n$ are the control points immediately before and after the point $p_n$. This construction ensures first-order continuity at the junction of adjacent curve segments (in this case, $p_n$) because the control points on either side of the point are colinear with the control point itself.

The end conditions can be handled by a similar construction. For example, the first control point, $a_0$, is constructed as the vector from the third interpolated point to the second point $(p_1 - p_2)$ is added to the second point (Figure 3.26).

In forming a control point, the quality of the interpolated curve can be affected by adjusting the distance the control point is from its associated interpolated point while maintaining its direction from that interpolated point. For example, a new $b'_n$ can be computed with a user-specified constant, *k,* as in Equation 3.28.

$$b'_n = p_n + k \cdot (b_n - p_n) \qquad \text{(Eq. 3.28)}$$

Between any two interpolated points, a cubic Bezier curve segment is then defined by the points $p_n$, $a_n$, $b_{n+1}$, $p_{n+1}$. The control point $b_{n+1}$ is defined in exactly the same way that $b_n$ is defined except for using $p_n$, $p_{n+1}$, and $p_{n+2}$. The cubic curve segment is then generated between $p_n$ and $p_{n+1}$. See Figure 3.27.

It should be easy to see how this procedure can be converted into the 4D spherical world of quaternions. Instead of adding vectors, rotations are concatenated. Averaging of orientations can easily be done by slerping to the halfway orientation, which is implemented by adding quaternions (and optionally normalizing).

Once the internal control points are computed, the De Casteljau algorithm can be applied to interpolate points along the curve. An example of the De Casteljau

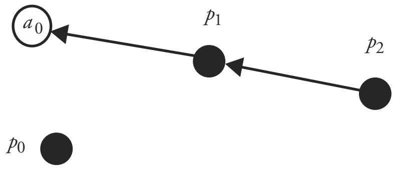


**Figure 3.26** Constructing the first interior control point

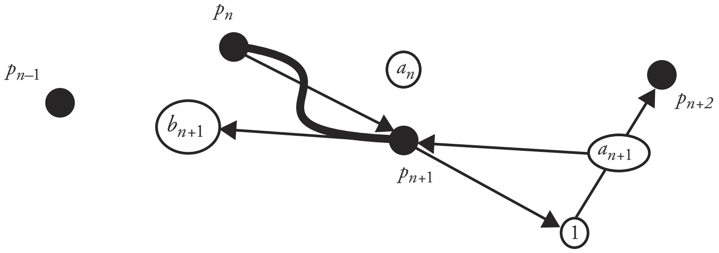
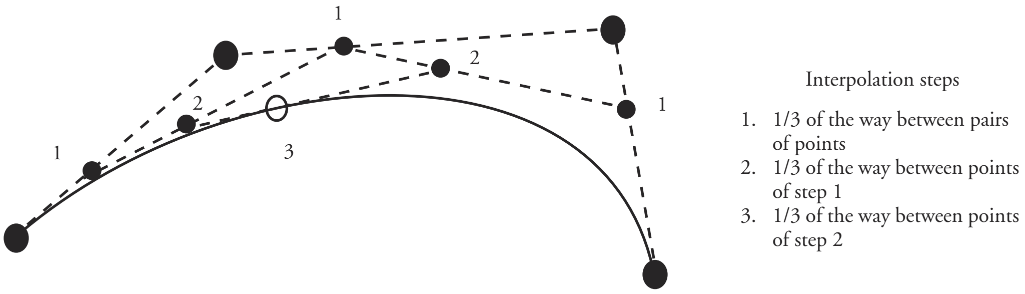


**Figure 3.27** Construction of $b_{n+1}$ and the resulting curve



Interpolation steps

1. 1/3 of the way between pairs of points
2. 1/3 of the way between points of step 1
3. 1/3 of the way between points of step 2

**Figure 3.28** De Casteljau construction of point on cubic Bezier segment at 1/3 (the point labeled "3")

construction procedure in the Euclidean case of Bezier curve interpolation is shown in Figure 3.28. See Appendix B for a more complete discussion of the procedure.

The same procedure can be used to construct the Bezier curve in four-dimensional spherical space. For example, to obtain an orientation corresponding to the $u = 1/3$ position along the curve, the following orientations are computed:

```
p1 = slerp (q_n,a_n,1/3)
p2 = slerp (a_n,b_n+1,1/3)
p3 = slerp (b_n+1,q_n+1,1/3)
p12 = slerp (p_1,p_2,1/3)
p23 = slerp (p_2,p_3,1/3)
p = slerp (p12,p23,1/3)
```

where $p$ is the quaternion representing an orientation 1/3 along the spherical cubic spline.

The procedure can be made especially efficient in the case of quaternion representations when calculating points at positions along the curve corresponding to $u$ values that are powers of 1/2. For example, consider calculating a point at $u = 1/4$.

```
temp = slerp ($q_n$,$a_n$,1/2) = $q_n$ + $a_n$
$p_1$ = slerp ($q_n$,temp,1/2) = $q_n$ + temp
temp = slerp ($a_n$,$b_{n+1}$,1/2) = $a_n$ + $b_{n+1}$
$p_2$ = slerp ($a_n$,temp,1/2) = $a_n$ + temp
temp = slerp ($b_{n+1}$,$q_{n+1}$,1/2) = $b_{n+1}$ + $q_{n+1}$
$p_3$ = slerp ($b_{n+1}$,temp,1/2) = $b_{n+1}$ + temp
temp = slerp ($p_1$,$p_2$,1/2) = $p_1$ + $p_2$
$p_{12}$ = slerp ($p_1$,temp,1/2) = $p_1$ + temp
temp = slerp ($p_2$,$p_3$,1/2) = $p_2$ + $p_3$
$p_{23}$ = slerp ($p_2$,temp,1/2) = $p_2$ + temp
temp = slerp ($p_{12}$,$p_{23}$,1/2) = $p_{12}$ + $p_{23}$
p = slerp ($p_{12}$,temp,1/2) = $p_{12}$ + temp
```

The procedure can be made more efficient if the points are generated in order according to binary subdivision (1/2, 1/4, 3/4, 1/8, 3/8, 5/8, 7/8, 1/16, . . .) and temporary values are saved for use in subsequent calculations.

## 3.4 Path Following

Animating an object by moving it along a path is a very common technique and usually one of the simplest to implement. As with most other types of animation, however, issues may arise that make the task more complicated than first envisioned. An object (or camera) following a path requires more than just translating along a space curve, even if the curve is parameterized by arc length and the motion is controlled using ease-in/ease-out. Changing the orientation of the object also has to be taken into consideration. If the path is the result of a digitization process, then often it must be smoothed before it can be used. If the path is constrained to lie on the surface of another object, then more computation is involved. These issues are discussed below.

### 3.4.1 Orientation along a Path

Typically, a local coordinate system ($u$, $v$, $w$) is defined for an object to be animated. In this discussion, a right-handed coordinate system is assumed, with the origin of the coordinate system determined by a point along the path $P(s)$. As previously discussed, this point is generated based on the frame number, arc length parameterization, and possibly ease-in/ease-out control. This position will be referred to as *POS*. The view vector is identified with the $w$-axis, the up vector is identified with the $v$-axis, and the local $u$-axis is perpendicular to these two. To form a right-handed coordinate system, the $u$-axis points to the left of the object as someone at the object's position (*POS*) looks down the $w$-axis with the head aligned with the $v$-axis (Figure 3.29).

There are various ways to handle the orientation of the camera as it travels along a path. Of course, which method to use depends on the desired effect of the animation. The orientation is specified by determining the direction of the $w$-axis and the direction of the $v$-axis; the $u$-axis is then fully specified by completing the left-handed coordinate system.

### Use of the Frenet Frame

If an object or camera is following a path, then its orientation can be made directly dependent on the properties of the curve. The Frenet frame[2] can be defined along the curve as a moving (right-handed) coordinate system, $(u, v, w)$, determined by the curve's tangent and curvature. The Frenet frame changes orientation over the length of the curve. It is defined as normalized orthogonal vectors with $w$ in the direction of the first derivative ($P'(s)$), $v$ orthogonal to $w$ and in the general direction of the second derivative ($P''(s)$), and $u$ formed by the cross product of the two (Figure 3.30). Specifically, at a given parameter value $s$, the Frenet frame is calculated according to Equation 3.29, as illustrated in Figure 3.31. The vectors are then normalized.

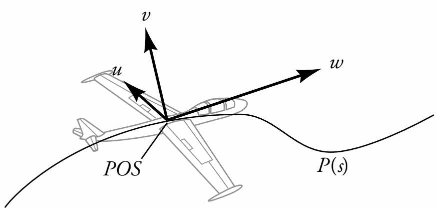


**Figure 3.29** Camera-based local coordinate system

[2] Note the potential source of confusion between the use of the *frame* to mean (1) a frame of animation and (2) the moving coordinate system of the Frenet frame. The context in which the term *frame* is used should determine its meaning.

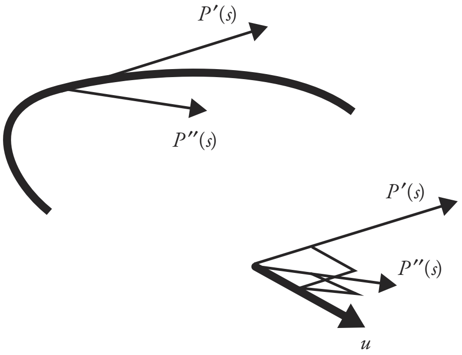


**Figure 3.30** The derivatives at a point along the curve

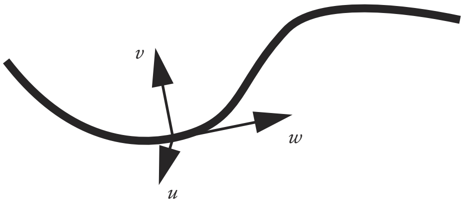
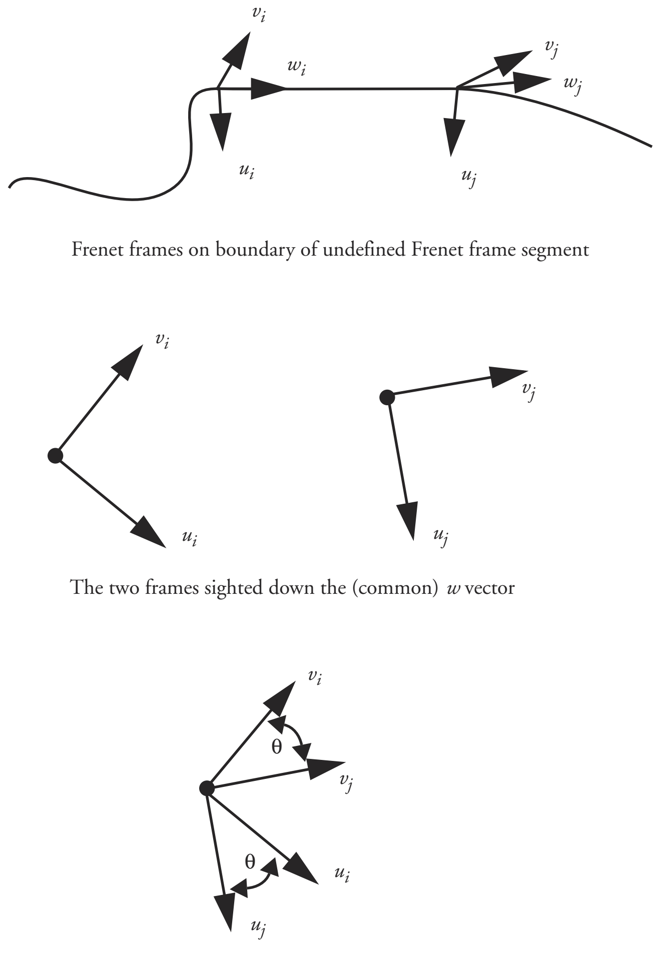


**Figure 3.31** Frenet frame at a point along a curve

$$
\begin{aligned}
w &= P'(s) \\
u &= P'(s) \times P''(s) \\
v &= w \times u
\end{aligned}
\qquad \text{(Eq. 3.29)}
$$

While the Frenet frame provides useful information about the curve, there are a few problems with using it directly to control the orientation of a camera or object as it moves along a curve. One problem occurs in segments of the curve that have no curvature ($P''(u) = 0$), because the Frenet frame is undefined. These undefined segments can be dealt with by interpolating a Frenet frame along the segment from the Frenet frames at the boundary of the segment. By definition, there is no curvature along this segment, so the boundary Frenet frames must differ by only a rotation around $w$. Assuming that the vectors have already been normalized, the angular difference between the two can be determined by taking the arccosine of the dot product between the two $v$ vectors so that $\theta = \mathrm{acos}(v_1 \bullet v_2)$. This rotation can be linearly interpolated along the no-curvature segment (Figure 3.32).

The problem is more difficult to deal with when there is a discontinuity in the curvature vector. Consider, for example, two semicircles spliced together so that



Frenet frames on boundary of undefined Frenet frame segment

The two frames sighted down the (common) $w$ vector

The two frames superimposed to identify angular difference

**Figure 3.32** Interpolating Frenet frames to determine the undefined segment

they form an S (*sigmoidal*) shape. The curvature vector, which for any point along this curve will point to the center of the semicircle that the point is on, will instantaneously switch from pointing to one center point to pointing to the other center point at the junction of the two semicircles. In this case, the Frenet frame is defined everywhere but has a discontinuous jump in orientation at the junction. See Figure 3.33.

However, the main problem with using the Frenet frame as the local coordinate frame to define the orientation of the camera or object following the path is that the resulting motions are usually too extreme and not natural looking. Using the $w$-axis (tangent vector) as the view direction of a camera can be undesirable.

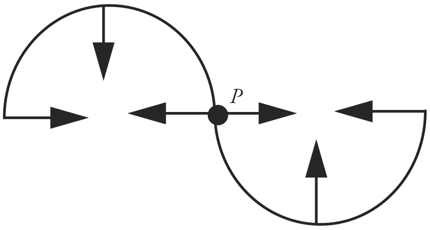


**Figure 3.33** The curvature vector, defined everywhere but discontinuous, instantaneously switches direction at point *P*

Often, the tangent vector does not appear to correspond to the direction of "where it's going" even though it is in the instantaneous (analytic) sense. The more natural orientation, for someone riding in a car or riding a bike, for example, would be to look farther ahead along the curve rather than to look tangential to the curve.

If the *v*-axis is equated with the up vector of an object, the object will rotate wildly about the path even when the path appears to mildly meander through an environment. With three-dimensional space curves, the problem becomes more obvious as the path bends down toward the ground; the camera will invert and travel along the path upside down. While the Frenet frame provides useful information to the animator, its direct use to control object orientation is clearly of limited value. When modeling the motion of banking into a turn, the curvature vector (*u*-axis) does indicate which way to bank and can be used to control the magnitude of the bank. For example, the horizontal component (the component in the *x-z* plane) of the *u*-axis can be used as the direction/magnitude indicator for the bank. For a different effect, the animator may want the object to tilt away from the curvature vector to give the impression of the object feeling a force that throws it off the path, such as when one rides a roller coaster.

### Camera Path Following: Adding a Center of Interest

The simplest method of specifying the orientation of a camera is to set its center of interest (COI) to a fixed point in the environment or, more elegantly, to use the center point of one of the objects in the environment. In either case, the center of interest is directly determined and is available for calculating the view vector, $w = COI - POS$. This is usually a good method when the path that the camera is following is one that circles some arena of action on which the camera's attention must be focused.

This still leaves one degree of freedom to be determined in order to fully specify the local coordinate system. For now, assume that the up vector, *v,* is to be kept "up." *Up* in this case means "generally in the positive *y*-axis direction," or, for a more mathematical definition, the *v*-axis is at right angles to the view vector (*w*-

axis) and is to lie in the plane defined by the local $w$-axis and the global $y$-axis. The local coordinate system can be computed as shown in Equation 3.30. The vectors can then be normalized to produce unit length vectors.

$$\begin{aligned} w &= COI - POS \\ u &= w \times y\text{-axis} \\ v &= u \times w \end{aligned} \qquad \text{(Eq. 3.30)}$$

For a camera traveling down a path, the view direction can be automatically set in several ways. As previously mentioned, the center of interest can be set to a specific point in the environment or to the position of a specific object in the environment. This works well as long as there is a single point or single object on which the camera should focus and as long as the camera does not pass too close to the point or object. Passing close to the center of interest will result in radical changes in view direction (of course, in some cases, the resulting effect may be desirable). Other methods use points along the path itself, a separate path through the environment, or interpolation between positions in the environment. The up vector can also be set in several ways. The default orientation is for the up vector to lie in the plane of the view vector and the global $y$-axis. Alternatively, a tilt of the up vector can be specified away from the default orientation as a user-specified value (or interpolated set of values). And, finally, the up vector can be explicitly specified by the user.

The simplest method of setting the view vector is to use a delta parametric value to define the center of interest. If the position of the camera on a curve is defined by $P(s)$, then the center of interest will be $P(s + \delta s)$. This, of course, should be after reparameterization by arc length. Otherwise, the actual distance to the center of interest along the path will vary over time (although if a relatively large $\delta u$ is used, the variation may not be noticeable). At the end of the curve, once $s + \delta s$ is beyond the path parameterization, the view direction can be interpolated to the end tangent vector as $s$ approaches one (in the case that distance is normalized).

Often, updating the center of interest to be a specific point along the curve can result in views that appear jerky. In such cases, averaging some number of positions along the curve to be the center-of-interest point can smooth the view. However, if the number of points used is too small or the points are too close together, the view may remain jerky. If $n$ is too large and the points are spaced out too much, the view direction may not change significantly during the traversal of the path and will appear too static. The number of points to use and their spread along the curve are dependent on the path itself and on the effect desired by the animator.

An alternative to using some function of the position path to produce the center of interest is to use a separate path altogether to define the center of interest. In this case, the camera's position is specified by $P(s)$, while the center of interest is specified by some $C(s)$. This requires more work by the animator but provides greater control and more flexibility. Similarly, an up vector path, $U(s)$, might be specified so that the general up direction is defined by $U(s) - P(s)$. This is just the general direction because a valid up vector must be perpendicular to the view vector. Thus the coordinate frame for the camera could be defined as in Equation 3.31.

$$\begin{aligned} w &= C(s) - P(s) \\ u &= w \times (U(s) - P(s)) \\ v &= u \times w \end{aligned} \tag{Eq. 3.31}$$

Instead of using a separate path for the center of interest, a simple but effective strategy is to fix it at one location for an interval of time and then move it to another location (using linear spatial interpolation and ease-in/ease-out temporal interpolation) and fix it there for a number of frames and so on. The up vector can be set as before in the default "up" direction.

## 3.4.2 Smoothing a Path

In cases in which the points making up a path are generated by a digitizing process, the resulting curve can be too jerky because of noise or imprecision. To remove the jerkiness, the coordinate values of the data can be smoothed by one of several approaches. For this discussion, the following set of data will be used: {(1, 1.6), (2, 1.65), (3, 1.6), (4, 1.8), (5, 2.1), (6, 2.2), (7, 2.0), (8, 1.5), (9, 1.3), (10, 1.4)}. This is plotted in Figure 3.34.

### Smoothing with Linear Interpolation of Adjacent Values

An ordered set of points in two-space can be smoothed by averaging adjacent points. In the simplest case, the two points, one on either side of an original point,

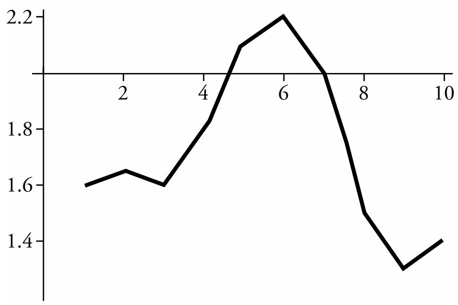


**Figure 3.34** Sample data for path smoothing

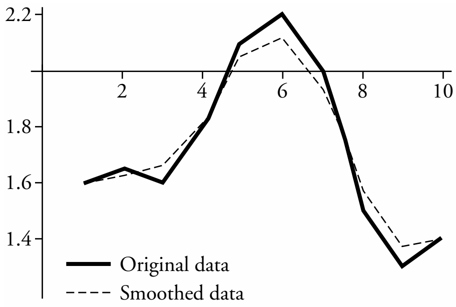


**Figure 3.35** Sample data smoothed by linear interpolation

$P_i$, are averaged. This point is averaged with the original data point (Equation 3.32). Figure 3.35 shows the sample data plotted with the original data. Notice how linear interpolation tends to draw the data points in the direction of local concavities. Repeated applications of the linear interpolation to further smooth the data would continue to draw the reduced concave sections and flatten out the curve.

$$P'_i = \frac{P_i + \dfrac{P_{i-1} + P_{i+1}}{2}}{2} = \frac{1}{4} \cdot P_{i-1} + \frac{1}{2} \cdot P_i + \frac{1}{4} \cdot P_{i+1} \qquad \text{(Eq. 3.32)}$$

## Smoothing with Cubic Interpolation of Adjacent Values

To preserve the curvature but still smooth the data, the adjacent points on either side of a data point can be used to fit a cubic curve that is then evaluated at its midpoint. This midpoint is then averaged with the original point, as in the linear case. A cubic curve has the form shown in Equation 3.33. The two data points on either side of an original point, $P_i$, are used as constraints, as shown in Equation 3.34. These equations can be used to solve for the constants of the cubic curve (*a, b, c, d*). Equation 3.33 is then evaluated at $u = 1/2$; and the result is averaged with the original data point (see Figure 3.36). Solving for the coefficients and evaluating the resulting cubic curve is a bit tedious, but the solution needs to be performed only once and can be put in terms of the original data points, $P_{i-2}$, $P_{i-1}$, $P_{i+1}$, $P_{i+2}$. This is shown geometrically in Figure 3.37.

$$P(u) = a \cdot u^3 + b \cdot u^2 + c \cdot u + d \qquad \text{(Eq. 3.33)}$$

$$
\begin{aligned}
P_{i-2} &= P(0) = d \\
P_{i-1} &= P(1/4) = a \cdot \frac{1}{64} + b \cdot \frac{1}{16} + c \cdot \frac{1}{4} + d \\
P_{i+1} &= P(3/4) = a \cdot \frac{27}{64} + b \cdot \frac{9}{16} + c \cdot \frac{3}{4} + d \\
P_{i+2} &= a + b + c + d
\end{aligned}
\qquad \text{(Eq. 3.34)}
$$

For the end conditions, a parabolic arc can be fit through the first, third, and fourth points, and an estimate for the second point from the start of the data set can be computed (Figure 3.38). The coefficients of the parabolic equation, $P(u) = a \cdot u^2 + b \cdot u + c$, can be computed from the constraints in Equation 3.35, and the equation can be used to solve for the position $P_1 = P(1/3)$.

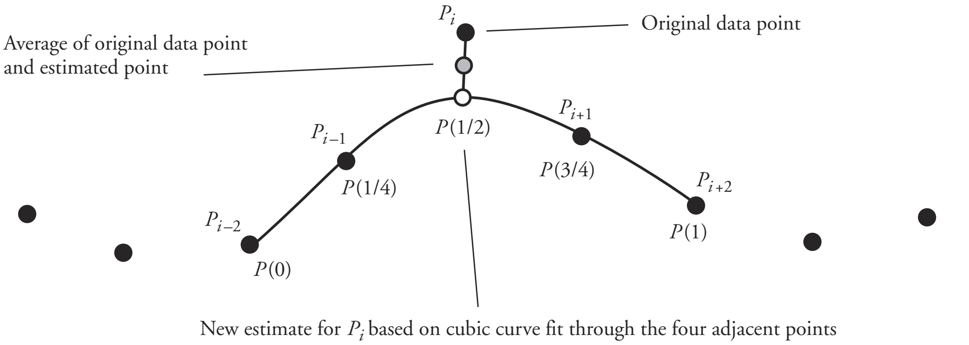
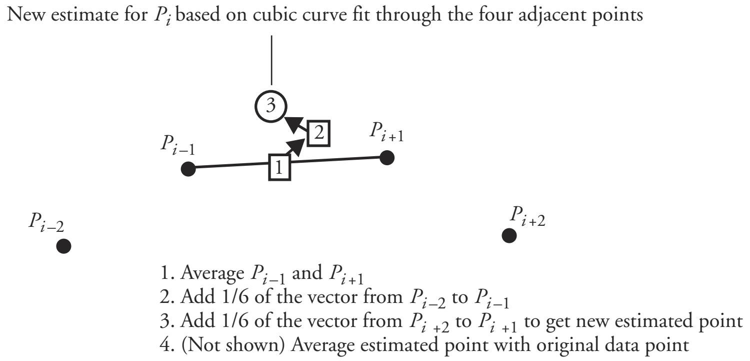

$P_i$
Original data point
Average of original data point and estimated point
$P_{i+1}$
$P_{i-1}$
$P(1/2)$
$P(3/4)$
$P(1/4)$
$P_{i+2}$
$P_{i-2}$
$P(1)$
$P(0)$
New estimate for $P_i$ based on cubic curve fit through the four adjacent points

**Figure 3.36** Smoothing data by cubic interpolation


New estimate for $P_i$ based on cubic curve fit through the four adjacent points
3
2
1
$P_{i-1}$
$P_{i+1}$
$P_{i-2}$
$P_{i+2}$

1. Average $P_{i-1}$ and $P_{i+1}$
2. Add 1/6 of the vector from $P_{i-2}$ to $P_{i-1}$
3. Add 1/6 of the vector from $P_{i+2}$ to $P_{i+1}$ to get new estimated point
4. (Not shown) Average estimated point with original data point

**Figure 3.37** Geometric construction of a cubic estimate for smoothing a data point

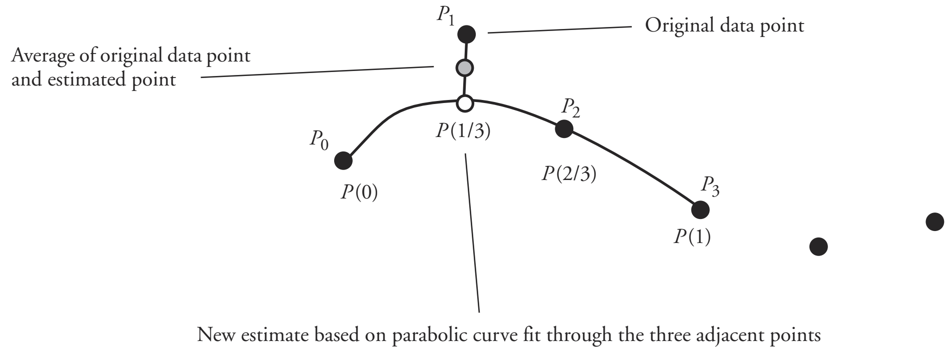

$P_1$
Original data point
Average of original data point and estimated point
$P_0$
$P(1/3)$
$P_2$
$P(0)$
$P(2/3)$
$P_3$
$P(1)$
New estimate based on parabolic curve fit through the three adjacent points

**Figure 3.38** Smoothing data by parabolic interpolation

$$P_0 = P(0),\ P_2 = P\left(\frac{2}{3}\right),\ P_3 = P(1) \qquad \textbf{(Eq. 3.35)}$$

This can be rewritten in geometric form and the point can be constructed geometrically from the three points $P_1' = P_2 + 1/3 \cdot (P_0 - P_3)$ (Figure 3.39). A similar procedure can be used to estimate the data point secondfrom the end. The very first and very last data points can be left alone if they represent hard constraints, or parabolic interpolation can be used to generate estimates for them as well, for example, $P_0' = P_3 + 3 \cdot (P_1 - P_2)$. Figure 3.40 shows cubic interpolation to smooth the data with and without parabolic interpolation for the endpoints.

## Smoothing with Convolution Kernels

When the data to be smoothed can be viewed as a value of a function, $y_i = f(x_i)$, the data can be smoothed by convolution. Figure 3.41 shows such a function

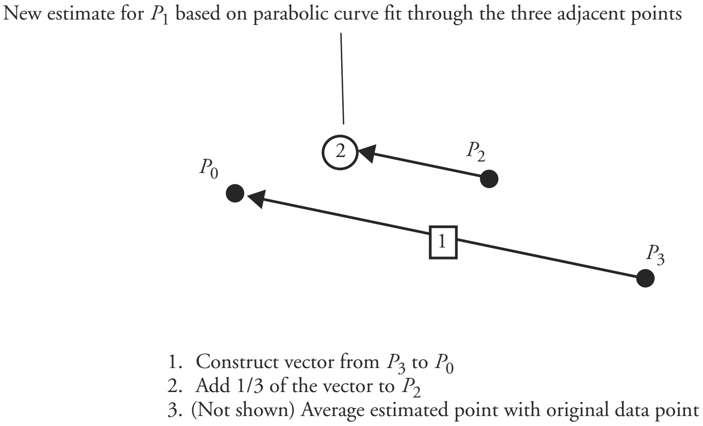

New estimate for $P_1$ based on parabolic curve fit through the three adjacent points
2
$P_2$
$P_0$
1
$P_3$
1. Construct vector from $P_3$ to $P_0$
2. Add 1/3 of the vector to $P_2$
3. (Not shown) Average estimated point with original data point

**Figure 3.39** Geometric construction of a parabolic estimate for smoothing a data point

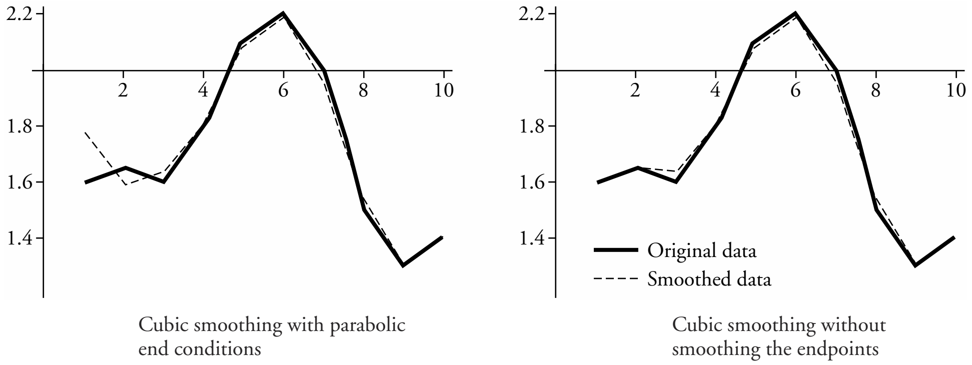


**Figure 3.40** Sample data smoothed with cubic interpolation

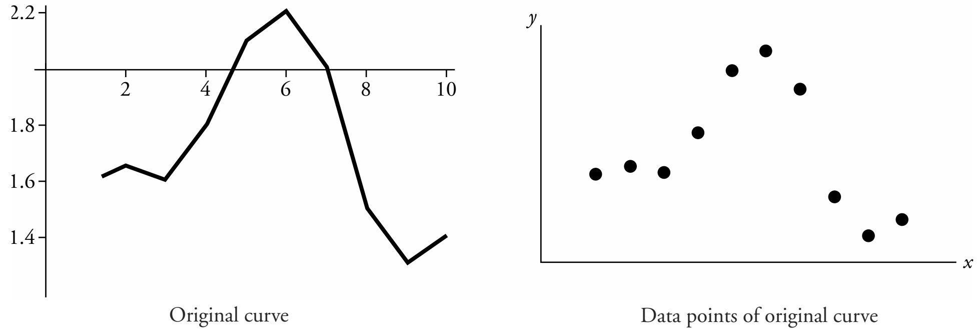


**Figure 3.41** Sample function to be smoothed

where the $x_i$ are equally spaced. A smoothing kernel can be applied to the data points by viewing them as a step function (Figure 3.42). Desirable attributes of a smoothing kernel include the following: it is centered around zero, it is symmetric, it has finite support, and the area under the kernel curve equals one. Figure 3.43 shows examples of some possibilities. A new point is calculated by centering the kernel function at the position where the new point is to be computed. The new point is calculated by summing the area under the curve that results from multiplying the kernel function, $g(u)$, by the corresponding segment of the step function, $f(x)$, beneath it (i.e., *convolution*). Figure 3.44 shows a simple tent-shaped kernel applied to a step function. In the continuous case, this becomes the integral, as shown in Equation 3.36, where $[-s, \ldots, s]$ is the extent of the support of the kernel function.

$$P(x) = \int_{-s}^{s} f(x+u) \cdot g(u)\, du \qquad \text{(Eq. 3.36)}$$

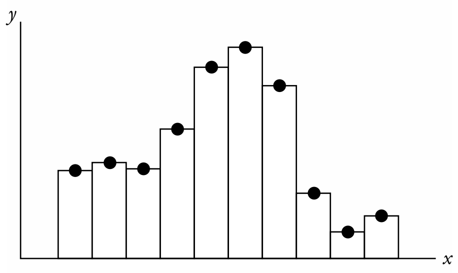


**Figure 3.42** Step function defined by data points of original curve

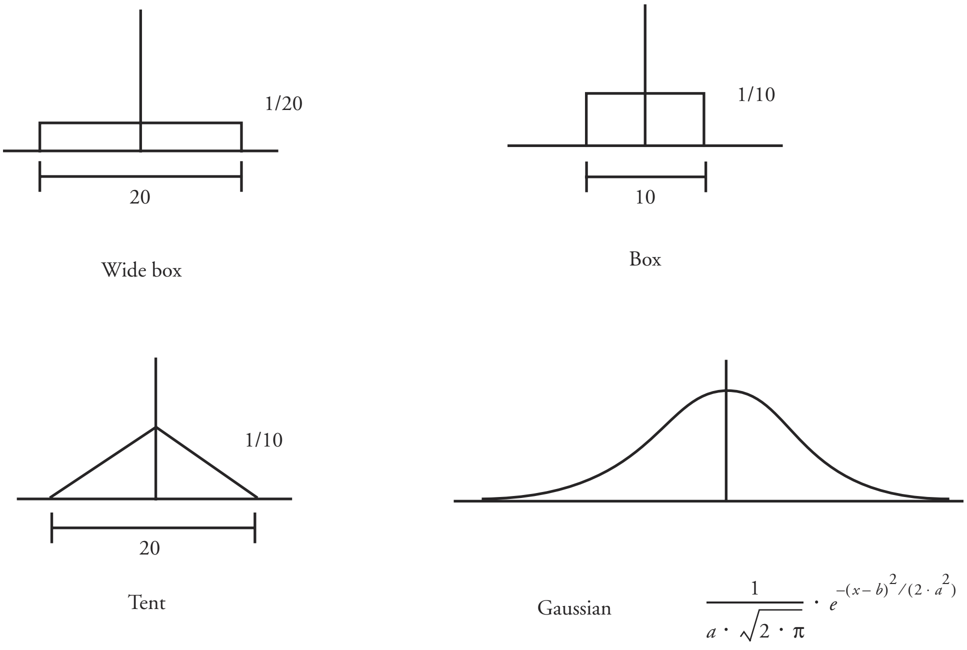
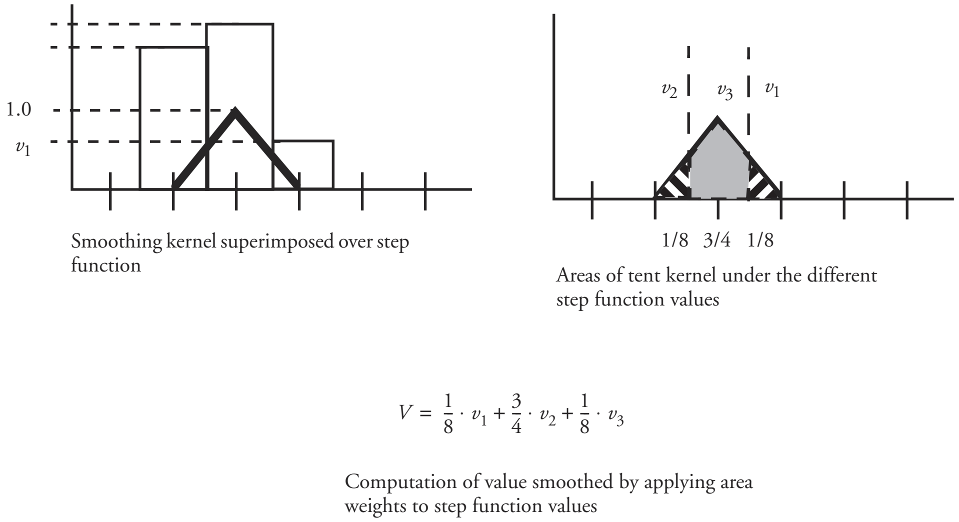

1/20
20
Wide box
1/10
10
Box
1/10
20
Tent
Gaussian
$\frac{1}{a \cdot \sqrt{2 \cdot \pi}} \cdot e^{-(x-b)^2/(2 \cdot a^2)}$

**Figure 3.43** Sample smoothing kernels

The integral can be analytically computed or approximated by discrete means. This can be done either with or without averaging down the number of points making up the path. Additional points can also be interpolated. At the endpoints, the step function can be arbitrarily extended so as to cover the kernel function when centered over the endpoints. Often the first and last points must be fixed because of animation constraints, so care must be taken in processing these. Figure 3.44 shows how a tent kernel is used to average the step function data; Figure 3.45 shows the sample data smoothed with the tent kernel.



Smoothing kernel superimposed over step function

Areas of tent kernel under the different step function values

$$V = \frac{1}{8} \cdot v_1 + \frac{3}{4} \cdot v_2 + \frac{1}{8} \cdot v_3$$

Computation of value smoothed by applying area weights to step function values

**Figure 3.44** Example of a tent-shaped smoothing filter

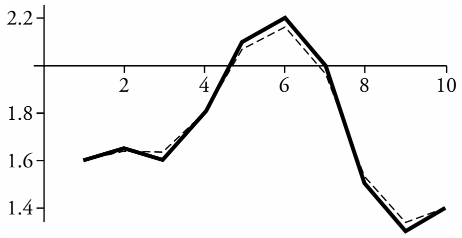


**Figure 3.45** Sample data smoothed with convolution using a tent kernel

### Smoothing by B-Spline Approximation

Finally, if an approximation to the curve is sufficient, then points can be selected from the curve, and, for example, B-spline control points can be generated based on the selected points. The curve can then be regenerated using the B-spline control points, which ensures that the regenerated curve is smooth even though it no longer passes through the original points.

## 3.4.3 Determining a Path along a Surface

If one object moves across the surface of another object, then a path across the surface must be determined. If start and destination points are known, it can be computationally expensive to find the shortest path between the points. However, often it is not necessary to find the absolute shortest path. Various alternatives exist for determining suboptimal paths.

An easy way to determine a path along a polygonal surface mesh is to determine a plane that contains the two points and is generally perpendicular to the surface. (*Generally perpendicular* can be defined, for example, as the average of the two vertex normals that the path is being formed between.) The intersection of the plane with the faces making up the surface mesh will define a path between the two points (Figure 3.46). If the surface is a higher-order surface and the known points are defined in the *u, v* coordinates of the surface definition, then a straight line (or curve) can be defined in the parametric space and transferred to the surface.

Alternatively, a greedy-type algorithm can be used to construct a path of edges along a mesh surface from a given start vertex to a given destination vertex. For each edge emanating from the current vertex (initially the start vertex), calculate the projection of the edge onto the straight line between the current vertex and the destination vertex. Divide this distance by the length of the edge to get the cosine of the angle between the edge and the straight line. The edge with the largest cosine is the edge most in the direction of the straight line; choose this edge to add to the path. Keep applying this until the destination edge is reached. Improvements to this approach can be made by allowing the path to cut across polygons to arrive at points along opposite edges rather than by going vertex to vertex. Candidate edges can be generated by projecting the straight line onto polygons containing the vertex and then computing their cosine values for consideration.

If a path downhill from an initial point on the surface is desired, then the surface normal and global up vector can be used to determine the downhill vector. The cross product of the normal and global up vector defines a vector that lies on the surface perpendicular to the downhill direction. So the cross product of this vector and the normal vector defines the downhill (and uphill) vector on a plane (Figure 3.47). This same approach works with curved surfaces to produce the instantaneous downhill vector.

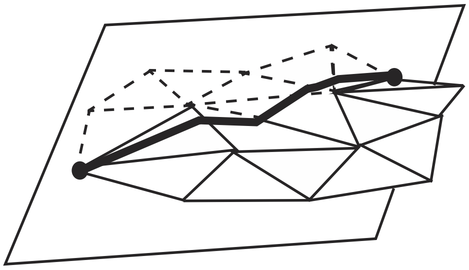


**Figure 3.46** Determining a path along a polygonal surface mesh by using plane intersection

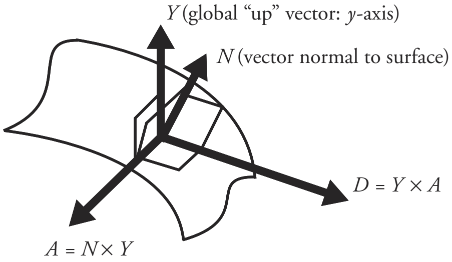


**Figure 3.47** Calculating the downhill vector, $D$

## 3.5 Key-Frame Systems

Many of the early computer animation systems were key-frame systems (e.g., [5] [6] [7] [24]). Most of these were 2D systems based on the standard procedure used for hand-drawn animation, in which master animators define and draw the key frames of the sequence to be animated. In hand-drawn animation, assistant animators have the task of drawing the intermediate frames by mentally inferring the action between the keys. The key frames occur often enough in the sequence so that the intermediate action is reasonably well defined, or the keys are accompanied by additional information to indicate placement of the intermediate frames. In computer animation, the term *key frame* has been generalized to apply to any variable whose value is set at specific key frames and from which values for the intermediate frames are interpolated according to some prescribed procedure. These variables have been referred to in the literature as *articulation variables* (*avars*) [28], and the systems are sometimes referred to as *track based*. It is common for such systems to provide an interactive interface with which the animator can specify the key values and the interpolation desired (Figure 3.48). The interpolation of variables is discussed at the beginning of this chapter; the rest of the discus-

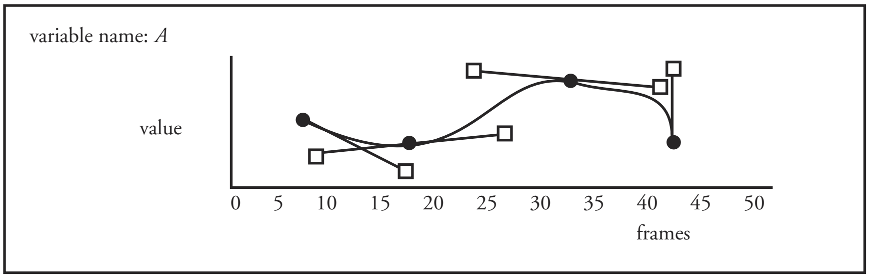


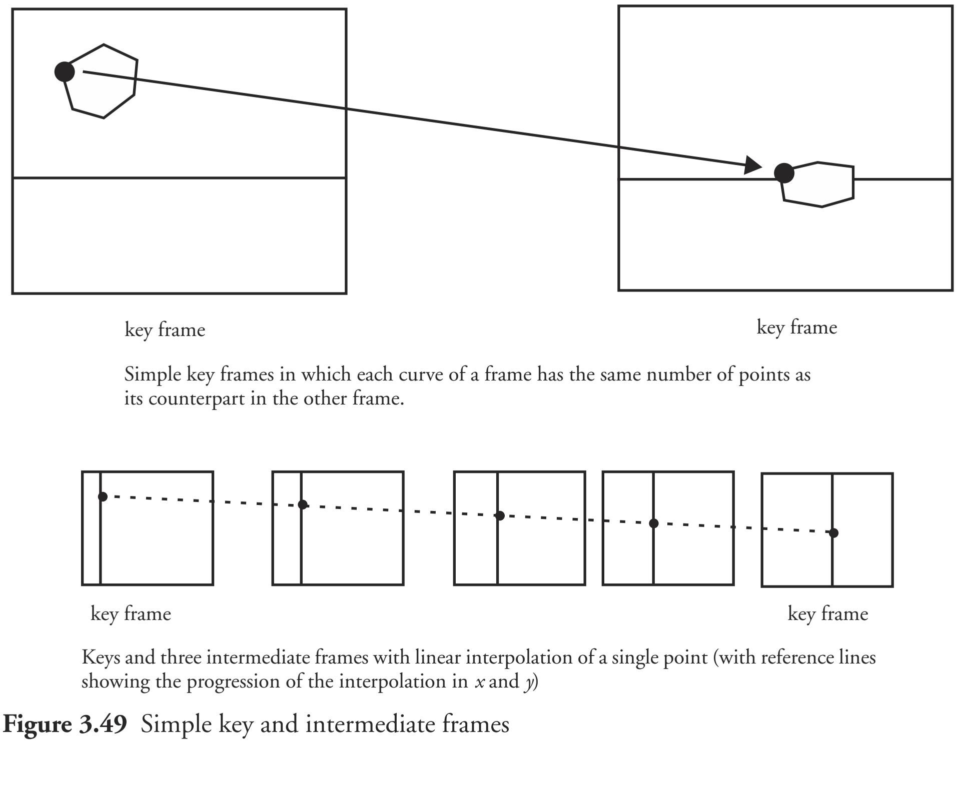
**Figure 3.48** Simple interface for specifying interpolation of key values

sion in this section focuses on shape interpolation analogous to the process of hand-drawn animation.

Because these animation systems keep the strategy of interpolating two-dimensional shapes, the basic operation is that of interpolating one (possibly closed) curve into another (possibly closed) curve. The interpolation is straightforward if the correspondence between lines in the frames is known in enough detail so that each pair of lines can be interpolated on a point-by-point basis to produce lines in the intermediate frames (and if this is desirable from the user's point of view). This interpolation requires that for each pair of curves in the key frames the curves have the same number of points and that for each curve, open or closed, the correspondence between the points can be established.

Of course, the simplest way to interpolate the points is using linear interpolation between each pair of keys (Figure 3.49). Moving in arcs and allowing for ease-in/ease-out can be accommodated by applying any of the interpolation techniques discussed in Appendix B, providing that a point can be identified over several key frames.

Point-by-point correspondence information is usually not known, and even if it is, the resulting correspondence is not necessarily what the user wants. The best



Simple key frames in which each curve of a frame has the same number of points as its counterpart in the other frame.

Keys and three intermediate frames with linear interpolation of a single point (with reference lines showing the progression of the interpolation in *x* and *y*)

**Figure 3.49** Simple key and intermediate frames

one can expect is for the curve-to-curve correspondence to be known. The problem is, given two arbitrary curves in key frames, to interpolate a curve as it "should" appear in intermediate frames. For example, observe the egg splatting against the wall in Figure 3.50.

For illustrative purposes, consider the simple case in which the key frames $f1$ and $f2$ consist of a single curve (Figure 3.51). The curve in frame $f1$ is referred to as $P(u)$, and the curve in frame $f2$ is referred to as $Q(v)$. This single curve must be interpolated for each frame between the key frames in which it is defined. For simplicity, but without loss of generality, it is assumed that the curve, while it may wiggle some, is basically a vertical line in both key frames.

Some basic assumptions are used about what constitutes reasonable interpolation, such as the fact that if the curve is a single continuous open segment in frames $f1$ and $f2$, then it should remain a single continuous open segment in all the intermediate frames. Also assumed is that the top point of $P$, $P(0)$, should interpolate to the top point in $Q$, $Q(0)$, and, similarly, the bottom points should interpolate. However, what happens at intermediate points along the curve is so far left undefined (other than for the obvious assumption that the entire curve $P$

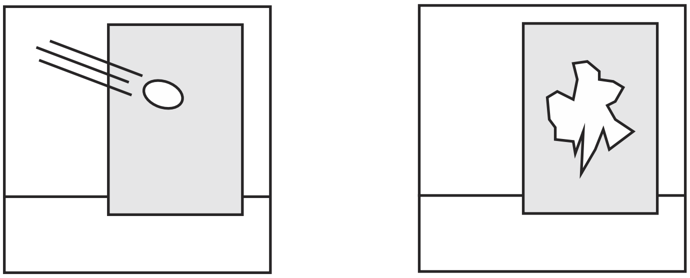


**Figure 3.50** Egg splatting against a wall whose shape must be interpolated

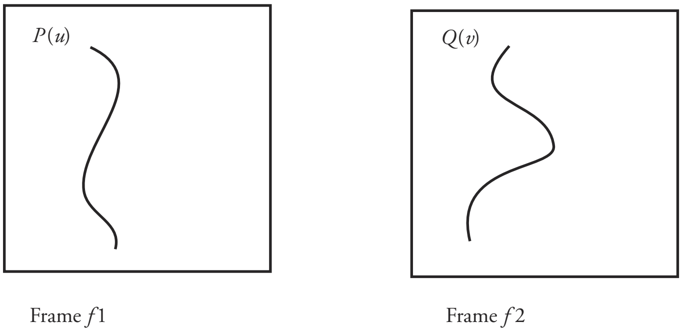


**Figure 3.51** Two frames showing a curve to be interpolated

should interpolate to the entire curve *Q;* that is, the mapping should be one-to-one and onto).

If both curves were generated with the same type of interpolation information, for example, each is a single, cubic Bezier curve, then intermediate curves could be generated by interpolating the control points and reapplying the Bezier interpolation. Another alternative would be to use interpolating functions to generate the same number of points on both curves. These points could then be interpolated on a point-by-point basis. Although these interpolations get the job done, they do not provide sufficient control to a user who has specific ideas of how they should progress.

Reeves [33] proposes a method of computing intermediate curves using *moving point constraints* that allows the user to specify more information concerning the correspondence of points along the curve and the speed of interpolation of those points. The basic approach is to use surface patch technology (two spatial dimensions) to solve the problem of interpolating a line in time (one spatial dimension, one temporal dimension).

The curve to be interpolated is defined in several key frames. Interpolation information—that is, a point's path and speed of interpolation as defined over two or more of the keys for one or more points—is defined for the curves. See Figure 3.52.

The first step is to define a segment of the curve to interpolate, bounded on top and bottom by interpolation constraints. Linear interpolation of the very top and very bottom of the curve, if not specified by a moving point constraint, is used to bound the top and bottom segments. Once a bounded segment has been formed, the task is to define an intermediate curve based on the constraints (see Figure 3.53).

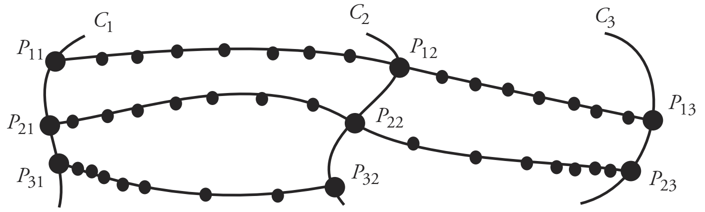


**Figure 3.52** Moving point constraints

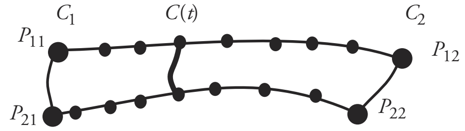


**Figure 3.53** Patch defined by interpolation constraints

Various strategies can be used to define the intermediate curve segment, $C(t)$ in Figure 3.53, and are typically applications of surface patch techniques. For example, tangent information along the curves can be extracted from the curve definitions. The endpoint and tangent information can then be interpolated along the top and bottom interpolation boundaries to define an intermediate curve.

## 3.6 Animation Languages

An animation language is a group of structured constructs that can be used to encode the information necessary to produce animations. The language can be script based, in which instructions are recorded for later evaluation, or it can be graphical, for example, in which flowchart-like diagrams encode relationships and procedures. Nadia Magnenat-Thalmann and Daniel Thalmann [26] present a good survey of early animation systems, and Steve May [28] presents a good overview of animation languages from the perspective of his interest in procedural representations and encapsulated models; much of the discussion here is taken from these sources.

The first animation systems used general-purpose programming languages (e.g., Fortran) to produce the motion sequences. However, each time an animation was to be produced, there was overhead in defining graphical primitives, object data structures, transformations, and renderer output. An animation language is any encoding of the animation to be produced. It may be written in a special script or in a general-purpose, possibly simplified, programming language that incorporates features that facilitate animation production. Typical features include built-in input/output operations for graphical objects, data structures to represent objects and support the hierarchical composition of objects, a time variable, interpolation functions, functions to animate object hierarchies, affine transformations, rendering-specific parameters, parameters for specifying camera attributes and defining the view frustum, and the ability to direct the producing, viewing, and storing of one or more frames of animation. The program written in an animation language is often referred to as a *script.*

The advantage of using an animation language is twofold. First, the specification of an animation written in the language is a hard-coded record of the animation that can be used at any time to regenerate it. The language also allows the animation to be iteratively refined because the script can be changed and the animation regenerated. Second, the availability of programming constructs allows an algorithmic approach to motion control. The animation language, if sufficiently powerful, can interpolate values, compute arbitrarily complex numerical quantities, implement behavioral rules, and so on. The disadvantage of using an animation language is that it is, fundamentally, a programming language and thus requires the user (animator) to be a programmer. The animator must be trained not only in the arts but also in the art of programming. Languages have been developed with simplified programming constructs, but such simplification usually reduces the algorithmic power of the language. As users become more familiar with a language, the tendency is to want more of the algorithmic power of a general-purpose programming language put back into the animation language. More recently, animation systems have been developed that are essentially user interfaces on top of a scripting language. Users can use the system strictly from the interface provided with the system, or they can write their own scripts in the underlying language. One example of such a system is Alias/Wavefront's MEL. MEL provides variables, control statements, procedures, expressions, access to object attributes, and the ability to customize the user interface. The standard user interface protects users who do not know how to program or who do not want to use the underlying language while giving access to the full power of the scripting language to those users who wish to use it. A sample MEL script is shown below [1].

```
global proc emitAway()
{
   emitter -pos 0 0 0 -type direction -sp 0.3 -name emit -r 50 -spd 1;
   particle -name spray;
   connectDynamic -em emit spray
   connectAttr emit.tx emitShape.dx;
   connectAttr emit.ty emitShape.dy;
   connectAttr emit.tz emitShape.dz;
   rename emit “emitAway#”;
   rename spray “sprayAway#”;
}
```

## 3.6.1 Artist-Oriented Animation Languages

To accommodate animators not trained in the art of computer programming, several simple animation languages were designed from the ground up with the intention of keeping the syntax simple and the semantics easy to understand (e.g., [18] [19] [27]). In the early days of computer graphics, almost all animation systems were based on such languages because there were few artists with technical backgrounds who were able to program in a full-fledged programming language. These systems, as well as some of those with graphical user interfaces (e.g., [15] [29] [39]), were more accessible to artists but also had limited capabilities.

In these simple animation languages, typical statements referred to named objects, a transformation, and a time at which (or a span of time over which) to apply the transformation to the object. They also tended to have a syntax that was easy to parse (for the interpreter/compiler) and easy to read (for the animator). The following examples are from ANIMA II [18].

```
set position <name> <x> <y> <z> at frame <number>
set rotation <name> [X,Y,Z] to <angle> at frame <number>
change position <name> to <x> <y> <z> from frame <number> to frame
   <number>
change rotation <name> [X,Y,Z] by <angle> from frame <number> to frame
   <number>
```

Specific values or variable names could be placed in the statements, which included an indicator as to whether the transformation was relative to the object's current position or absolute in global coordinates. The instructions operated in parallel; the script was not a traditional programming language in that sense.

As animators became more adept at writing animation scripts in such languages, they tended to demand that more capability be incorporated into the language. As May [28] points out, "By eliminating the language constructs that make learning animation languages difficult, command [artist-oriented] languages give up the mechanisms that make animation languages powerful." The developers found themselves adding looping constructs, conditional control statements, random variables, procedural calls, and data structure support. It usually became clear that adding support for graphical objects and operations to an existing language such as C or LISP was more reasonable than developing a full animation programming language from scratch.

### 3.6.2 Articulation Variables

A feature used by several languages is associating the value of a variable with a function, most notably, of time. The function is specified procedurally or designed interactively using interpolating functions. Thus a script or other type of animation system, when needing a value of the variable in its computation, passes the time variable to the articulation function, which returns its value for that time to the computation. This technique goes by a variety of names, such as *track, channel,* or *articulation variable*. The term *articulation variable,* often shortened to *avar,* stems from its common use in various systems to control the articulation of linked appendages.

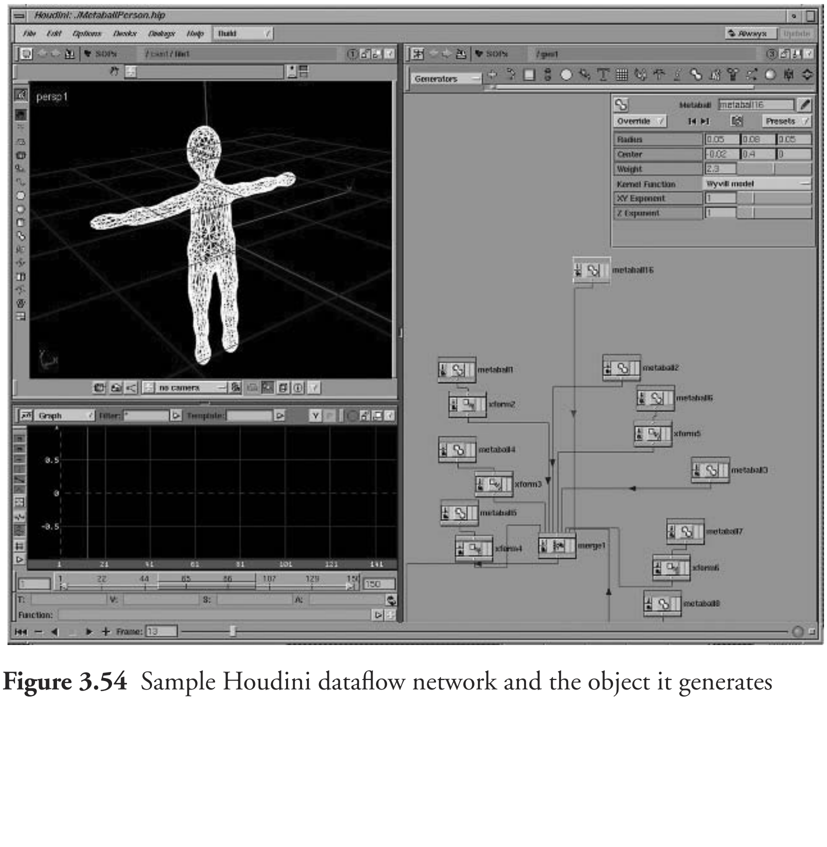
The use of avars when an interactive system is provided to the user for designing the functions allows a script-based animation language to incorporate some interaction during the development of the animation. It also allows digitized data as well as arbitrarily complex functions to be easily incorporated into an animation.

### 3.6.3 Graphical Languages

Graphical representations, such as those used in the commercial Houdini system [38], represent an animation by a dataflow network. See Figure 3.54 (Plate 2). An acyclic graph is used to represent objects, operations, and the relationships among them. Nodes of the graph have inputs and outputs that connect to other nodes, and data are passed from node to node by arcs of the graphics that are interactively specified by the user. A node represents an operation to perform on the data being passed into the node, such as an object description. A transformation node will operate on an object description passed into the node and will produce a transformed object representation as output. Inputs to a node can also be used to parameterize a transformation or a data generation technique and can be set interactively, set according to an articulation variable, or set according to an arbitrary user-supplied procedure.

### 3.6.4 Actor-Based Animation Languages

Actor-based languages are an object-oriented approach to animation in which an actor (encapsulated model [28]) is a graphical object with its associated data and



Metaball metaball16

Override Presets

| | | | |
|---|---|---|---|
| Radius | 0.05 | 0.08 | 0.05 |
| Center | -0.02 | 0.4 | 0 |
| Weight | 2.3 | | |
| Kernel Function | Wyvill model | | |
| XY Exponent | 1 | | |
| Z Exponent | 1 | | |

**Figure 3.54** Sample Houdini dataflow network and the object it generates

procedures, including geometric description, display attributes, and motion control. Reynolds [34] popularized the use of the term *actor* in reference to the encapsulated models he uses in his ASAS system.

The basic idea is that the data associated with a graphical object should not only specify its geometry and display characteristics but also describe its motion. Thus the encapsulated model of a car includes how the doors open, how the windows roll down, and, possibly, the internal workings of the engine. Communication with the actor takes place by way of message passing; animating the object is carried out by passing requests to the actor for certain motions. The current status of the actor can be extracted by sending a request for information to the actor and receiving a message from the actor.

Actor-based systems provide a convenient way of communicating the time-varying information pertaining to a model. However, the encapsulated nature of actors with the associated message passing can result in inefficiencies when they are used with simulation systems in which all objects have the potential to affect all others.

## 3.7 Deforming Objects

Deforming an object or transforming one object into another is a visually powerful animation technique. It adds the notion of malleability and density. Flexible body animation makes the objects in an animation seem much more expressive. There are physically based approaches that simulate the flexing of objects undergoing forces. However, many animators want more precise control over the shape of an object than that provided by simulations and/or do not want the computational expense of the simulating physical processes. In such cases, the animator wants to deform the object directly and define key shapes. Shape definitions that share the same edge connectivity can be interpolated on a vertex-to-vertex basis in order to smoothly change from one shape to the other. A sequence of key shapes can be interpolated over time to produce flexible body animation.

It can probably be agreed that uniform scale does not change the shape of an object, but what about nonuniform scale? Does a rectangle have the same shape as a square? Most would say no. Most would agree that shearing changes the shape of an object. Elementary schools often teach that a square and a diamond are different shapes even though they may differ by only a rotation. The affine is the simplest type of transformation that (sometimes) changes the shape of an object; affine transformations are defined by a 3x3 matrix followed by a translation. Affine transformations can be used to model the squash & stretch of an object, the jiggling of a block of Jello, and the shearing effect of an exaggerated stopping motion. Nonuniform scale can be used for simple squash & stretch, but more interesting shape distortions are possible with nonaffine transformations. User-defined distortions are discussed below; physically based approaches are discussed in the next chapter.

### 3.7.1 Warping an Object

A particularly simple way to modify the shape of an object is to displace one vertex (the *seed* vertex) or group of vertices of the object and propagate the displacement to adjacent vertices along the surface by attenuating the initial vertex's displacement. The displacement can be attenuated as a function of the distance between the seed vertex and the vertex to be displaced. See Figure 3.55. A *distance function* can be chosen to trade off quality of the results with computational complexity. One simple function uses the minimum number of edges connecting the seed vertex with the vertex to be displaced. Another such function uses the minimum distance traveled over the surface of the object between the seed vertex and the vertex to be displaced.

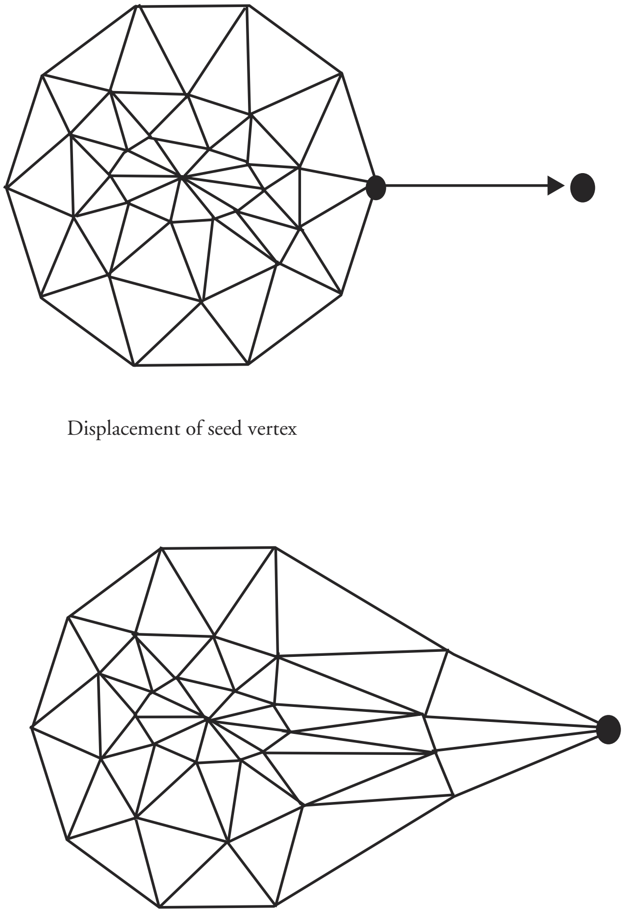
Attenuation is a function of the distance metric. In one approach [32], the user selects a function from a family of power functions to control the amount of attenuation. For a particular vertex that is *i* edges away from the seed vertex and where the range of effect has been user selected to be vertices within *n* edges of the seed vertex, a scale factor is applied to the displacement vector according to the user-selected integer value of *k* as shown in Equation 3.37.

$$\begin{aligned} S(i) &= 1.0 - \left(\frac{i}{n+1}\right)^{k+1} && k \geq 0 \\ &= \left(1.0 - \left(\frac{i}{n+1}\right)\right)^{-k+1} && k < 0 \end{aligned} \qquad \text{(Eq. 3.37)}$$

These attenuation functions are easy to compute and provide sufficient flexibility for many desired effects. When *k* equals zero it corresponds to a linear attenuation, while higher values of *k* create a more elastic impression. Values of *k* less than zero create the effect of more rigid displacements (Figure 3.56).

### 3.7.2 Coordinate Grid Deformation

A popular technique for modifying the shape of an object is credited to Sederberg [36] and is called *free-form deformation* (FFD). FFD is only one of a number of techniques that share a similar approach, establishing a local coordinate system that encases the area of the object to be distorted. The initial configuration of the local coordinate system is such that the determination of the local coordinates of a vertex



Displacement of seed vertex

Attenuated displacement propagated to adjacent vertices.

**Figure 3.55** Warping of object vertices

is a simple process. Typically the initial configuration has orthogonal axes. The object to be distorted, whose vertices are defined in global coordinates, is then placed in this local coordinate space by determining the local coordinates of its vertices. The local coordinate system is distorted by the user in some way, and the local coordinates of the vertices are used to map their positions in global space. The idea behind these techniques is that it is easier or more intuitive for the user to manipulate the local coordinate system than to manipulate the vertices of the object. Of course, the trade-off is that the manipulation of the object is restricted to possible distortions of the coordinate system. The local coordinate grid is usually distorted

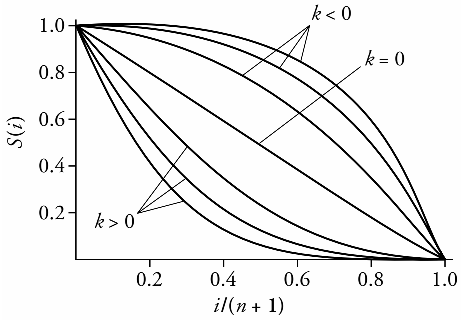


**Figure 3.56** Power functions

so that the mapping is continuous (space is not torn apart), but it can be nonlinear, making this technique more powerful than affine transformations.

### 2D Grid Deformation

Before proceeding to FFDs, a simpler scheme is presented. This is the seminal work in flexible body animation [6] demonstrated in the 1974 film *Hunger*. Peter Foldes, Nestor Burtnyk, and Marceli Wein used a 2D technique that allowed for shape deformation. In this technique, the local coordinate system is a 2D grid in which an object is placed. The grid is initially aligned with the global axes so that the mapping from local to global coordinates consists of a scale and a translate. For example, in Figure 3.57, assume that the local grid vertices are defined at global integer values from 20 to 28 in $x$ and from 12 to 15 in $y$. Vertex $A$ in the figure has global coordinates of (25.6, 14.7). The local coordinates of vertex $A$ would be (5.6, 2.7).

The grid is then distorted by the user moving the vertices of the grid so that the local space is distorted. The vertices of the object are then relocated in the distorted grid by bilinear interpolation relative to the cell of the grid in which the vertex is located (Figure 3.58).

The bilinear interpolants used for vertex $A$ would be 0.6 and 0.7. The positions of vertices of cell (5, 2) would be used in the interpolation. Assume the cell's vertices are named $P00$, $P01$, $P10$, and $P11$. Bilinear interpolation results in Equation 3.38.

$$\begin{aligned} P = \; & 0.6 \cdot 0.7 \cdot P00 + 0.6 \cdot 0.3 \cdot P01 \\ & + 0.4 \cdot 0.7 \cdot P10 + 0.4 \cdot 0.3 \cdot P11 \end{aligned} \qquad \text{(Eq. 3.38)}$$

Once this is done for all vertices of the object, the object is distorted according to the distortion of the local grid. See Figure 3.59. For objects that contain hundreds or thousands of vertices, the grid distortion is much more efficient than

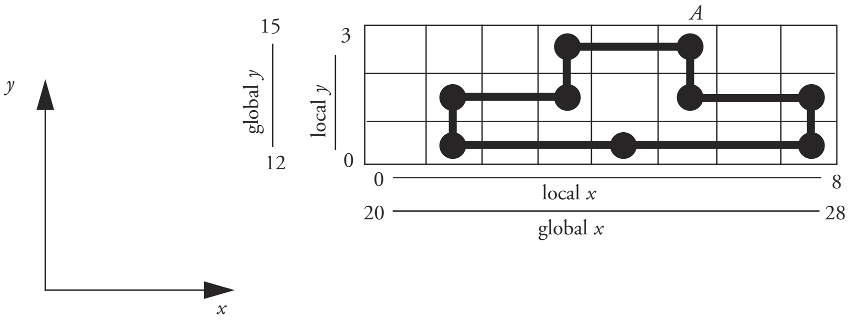


**Figure 3.57** Initial 2D coordinate grid

$$
\begin{aligned}
Pu0 &= (1-u)\cdot P00 + u\cdot P10\\
Pu1 &= (1-u)\cdot P01 + u\cdot P11\\
Puv &= (1-v)\cdot Pu0 + v\cdot Pu1\\
&= (1-u)\cdot(1-v)\cdot P00 + (1-u)\cdot v\cdot P01 + u\cdot(1-v)\cdot P10 + u\cdot v\cdot P11
\end{aligned}
$$

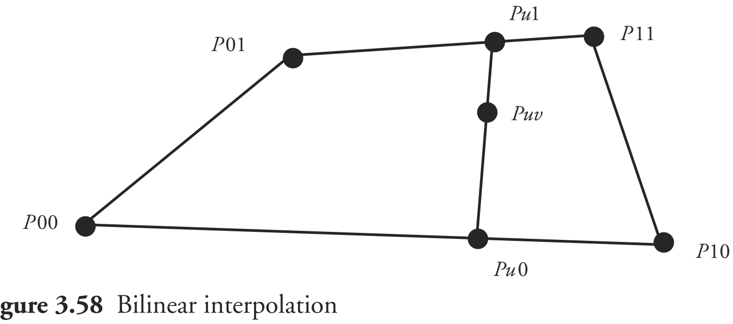


**Figure 3.58** Bilinear interpolation

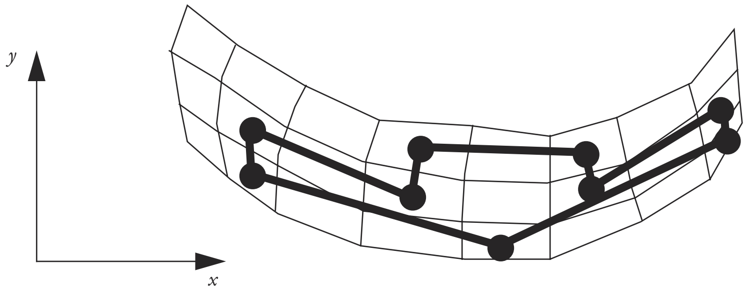


**Figure 3.59** 2D grid deformation

individually repositioning each vertex. In addition, it is more intuitive for the user to specify a deformation.

### Polyline Deformation

A 2D technique that is similar to the grid approach but lends itself to serpentine objects is based on a simple polyline (linear sequence of connected line segments) drawn by the user through the object to be deformed. Polyline deformation is similar to the grid approach in that the object vertices are mapped to the polyline, the polyline is then modified by the user, and the object vertices are then mapped to the same relative location on the polyline.

The mapping to the polyline is performed by first locating the most relevant line segment for each object vertex. To do this, intersecting lines are formed at the junction of adjacent segments, and perpendicular lines are formed at the extreme ends of the polyline. These lines will be referred to as the boundary lines; each polyline segment has two boundary lines. For each object vertex, the closest polyline segment that contains the object vertex between its boundary lines is selected (Figure 3.60).

Next, each object vertex is mapped to its corresponding polyline segment. A line segment is constructed through the object vertex parallel to the polyline segment and between the boundary lines. For a given object vertex, the following information is recorded (Figure 3.61): the closest line segment (*L*2); the line segment's distance to the polyline segment (*d*); and the object vertex's relative position on this line segment, that is, the ratio *r* of the length of the line segment (*d*1) and the distance from one end of the line segment to the object vertex (*d*2).

The polyline is then repositioned by the user and each object vertex is repositioned relative to the polyline using the information previously recorded for that vertex. A line parallel to the newly positioned segment is constructed *d* units away and the vertex's new position is the same fraction along this line that it was in the original configuration. See Figure 3.62.

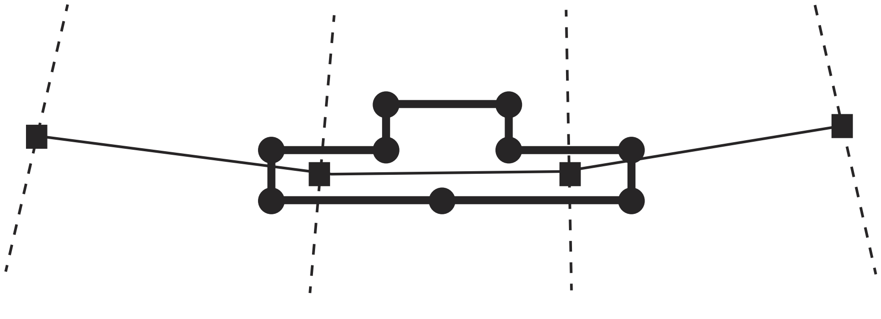


**Figure 3.60** Polyline drawn through object; bisectors and perpendiculars are drawn as dashed lines

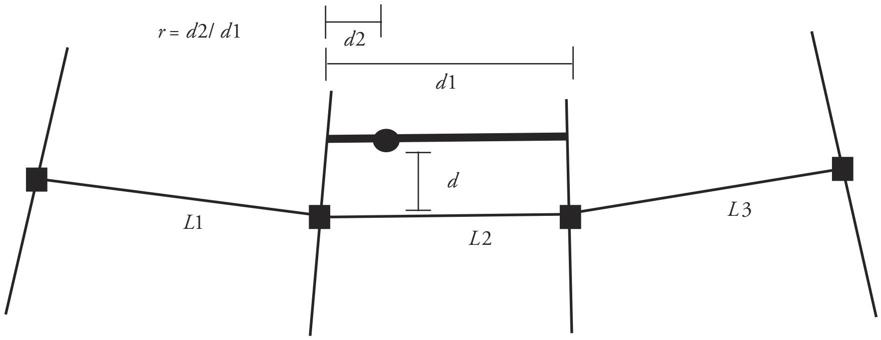


**Figure 3.61** Measurements used to map an object vertex to a polyline

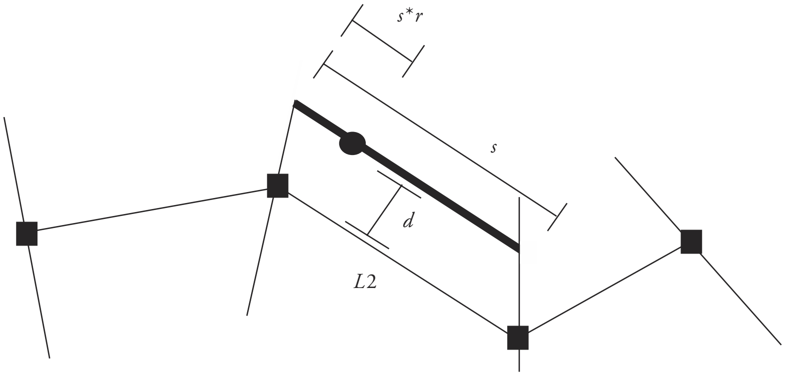
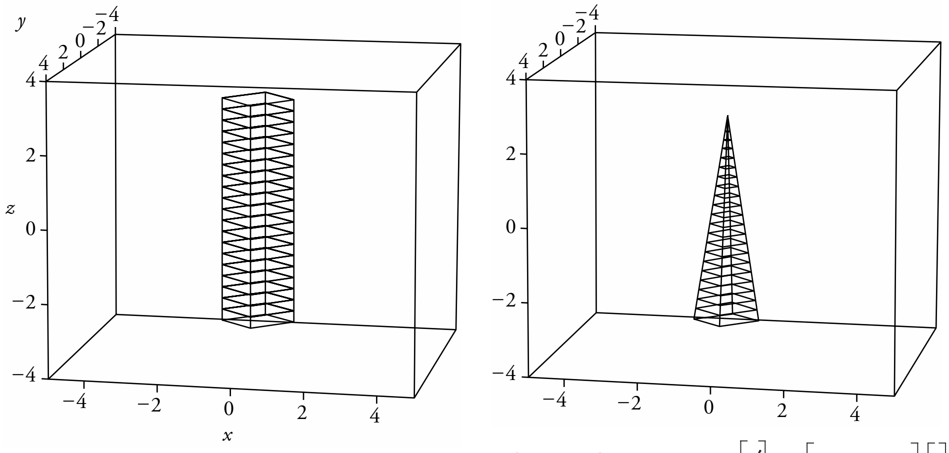


**Figure 3.62** Remapping of an object vertex relative to a deformed polyline (see Figure 3.61)

## Global Deformations

Alan Barr [2] presents a method of globally deforming the space in which an object is defined. Essentially, he applies a 3x3 transformation matrix, $M$, which is a function of the point being transformed, that is, $P' = M(P) \cdot P$, where $M(P)$ indicates the dependence of $M$ on $P$. For example, Figure 3.63 shows a simple linear 2D tapering operation. There is no reason why the function ($f(x)$ in Figure 3.63) needs to be linear; it can be whatever function produces the desired effect. Other global deformations are possible. In addition to the taper operation, twists (Figure 3.64), bends (Figure 3.65), and various combinations of these (Figure 3.66) are possible. For more details about these operations and Barr's discussion of what he terms local deformations, the interested reader is encouraged to refer to Barr's paper [2].



$$s(z) = \frac{(maxz - z)}{(maxz - minz)}$$

$$x' = s(z) \cdot x$$
$$y' = s(z) \cdot y$$
$$z' = z$$

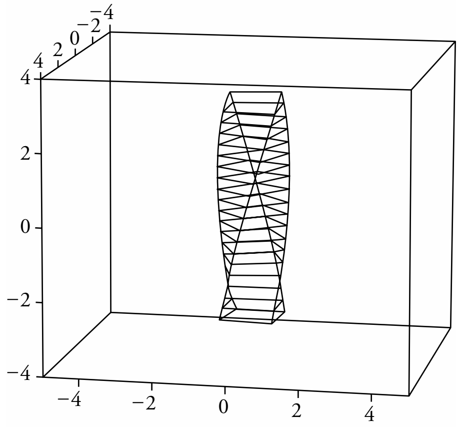
$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s(z) & 0 & 0 \\ 0 & s(z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = M(P) \cdot P$$

Original object

Tapered object

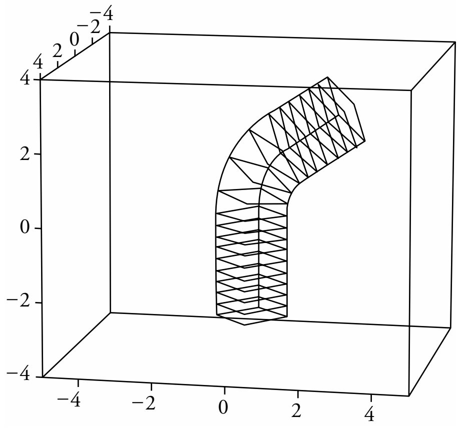
**Figure 3.63** Global tapering



$k$ = twist factor

$$x' = x \cdot \cos(k \cdot z) - y \cdot \sin(k \cdot z)$$
$$y' = x \cdot \sin(k \cdot z) + y \cdot \cos(k \cdot z)$$
$$z' = z$$

**Figure 3.64** Twist about an axis



$$\theta = \begin{cases} z - z_{\min} & z < z_{\max} \\ z_{\max} - z_{\min} & \text{otherwise} \end{cases}$$

$(z_{\min}: z_{\max})$—bend region

$(y_0, z_{\min})$—center of bend

$$C_\theta = \cos\theta$$

$$S_\theta = \sin\theta$$

$$R = y_0 - y$$

$$x' = x$$

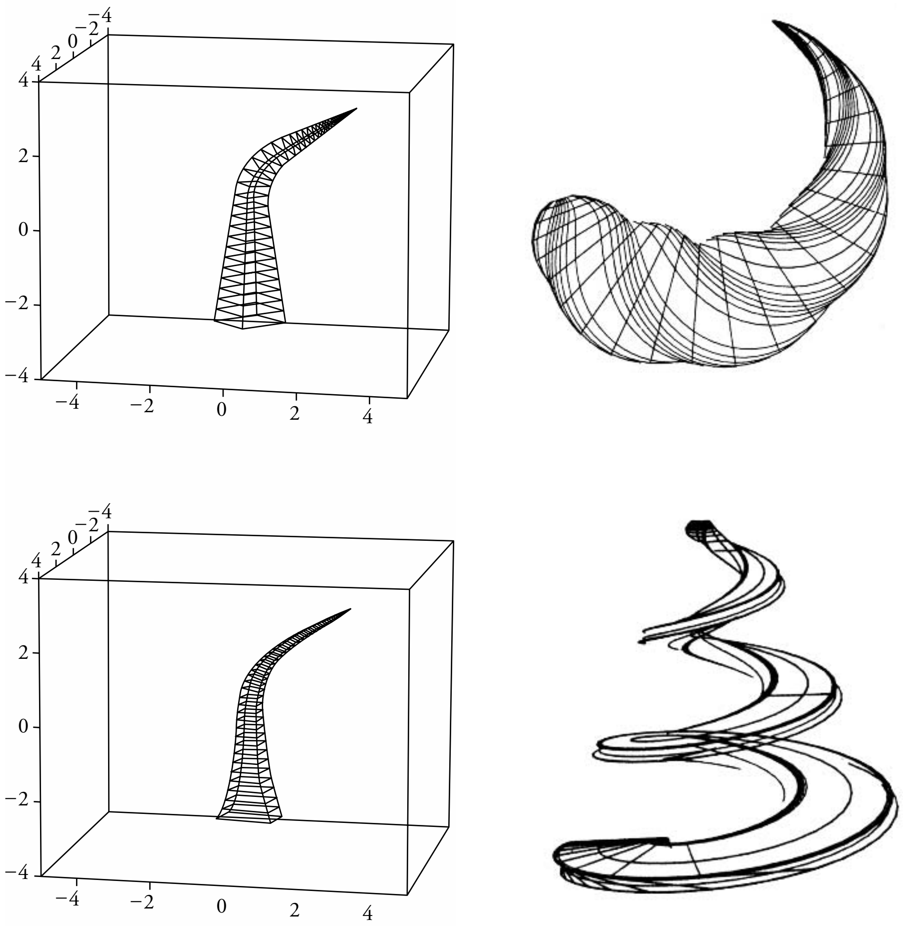
$$y' = \begin{cases} y & z < z_{\min} \\ y_0 - (R \cdot C_\theta) & z_{\min} \le z \le z_{\max} \\ y_0 - (R \cdot C_\theta) + (z - z_{\max}) \cdot S_\theta & z > z_{\max} \end{cases}$$

$$z' = \begin{cases} z & z < z_{\min} \\ z_{\min} + (R \cdot S_\theta) & z_{\min} \le z \le z_{\max} \\ z_{\min} + (R \cdot S_\theta) + (z - z_{\max}) \cdot C_\theta & z > z_{\max} \end{cases}$$

**Figure 3.65** Global bend operation

### Free-Form Deformation

FFD is essentially a 3D extension of Burtnyk's technique and incorporates higher-order interpolation. In both cases, a localized coordinate grid, in a standard configuration, is superimposed over an object. For each vertex of the object, coordinates relative to this local grid are determined that register the vertex to the grid. The grid is then manipulated by the user. Using its relative coordinates, each vertex is then mapped back into the modified grid, which relocates it in global space. Instead of linear interpolation, a cubic interpolation is typically used with FFD. In



Compound global deformations

Examples from Barr [2]

**Figure 3.66** Examples of global deformations

Sederberg's original paper [36], Bezier interpolation is suggested as the interpolating function, but any interpolation technique could be used.

Points of an object are located in a three-dimensional rectilinear grid. Initially the local coordinate system is defined by a not necessarily orthogonal set of three vectors (*S, T, U*). A point *P* is registered in the local coordinate system by determining its trilinear interpolants, as done in Equation 3.39, Equation 3.40, and Equation 3.41:

$$s = (T \times U) \bullet (P - P0) / ((T \times U) \bullet S) \qquad \textbf{(Eq. 3.39)}$$

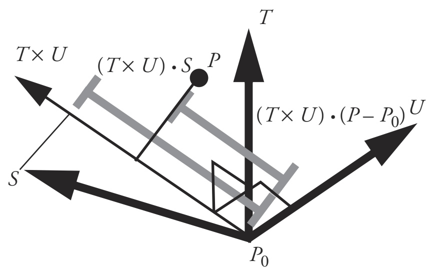


**Figure 3.67** Initial local coordinate system for FFDs

$$t = (U \times S) \bullet (P - P0) / ((U \times S) \bullet T) \qquad \text{(Eq. 3.40)}$$

$$u = (S \times T) \bullet (P - P0) / ((S \times T) \bullet U) \qquad \text{(Eq. 3.41)}$$

In the equations above, the cross product of two vectors forms a third vector that is orthogonal to the first two. The denominator normalizes the value being computed. In the first equation, for example, the projection of $S$ onto $T \times U$ determines the distance within which points will map into the range $0 < s < 1$.

Given the local coordinates (*s, t, u*) of a point and the unmodified local coordinate grid, a point's position can be reconstructed in global space by simply moving in the direction of the local coordinate axes according to the local coordinates (Equation 3.42):

$$P = P0 + s \cdot S + t \cdot T + u \cdot U \qquad \text{(Eq. 3.42)}$$

To facilitate the modification of the local coordinate system, a grid of control points is created in the parallelepiped defined by the *S, T, U* axes. There can be an unequal number of points in the three directions. For example, in Figure 3.68, there are four in the *S* direction, three in the *T* direction, and two in the *U* direction.

If there are $n_S$ points in the *S* direction, $n_T$ points in the *T* direction, and $n_U$ points in the *U* direction, the control points are located according to Equation 3.43.

$$P_{ijk} = P_0 + \frac{i}{l} \cdot S + \frac{j}{m} \cdot T + \frac{k}{n} \cdot U \qquad \text{(Eq. 3.43)}$$

The deformations are specified by moving the control points from their initial positions. The function that effects the deformation is a trivariate Bezier interpolating function. The deformed position of a point $P_{stu}$ is determined by using its (*s, t, u*) local coordinates, as defined by Equation 3.39–Equation 3.41, in the Bezier interpolating function shown in Equation 3.44. In Equation 3.44, *P*(*s, t, u*) repre-

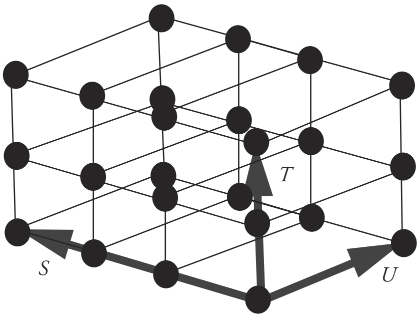


**Figure 3.68** Grid of control points

sents the global coordinates of the deformed point, and $P_{ijk}$ represents the global coordinates of the control points.

$$P(s, t, u) = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^{i} \cdot \left( \sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^{j} \cdot \left( \sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^{k} P_{ijk} \right) \right) \quad \text{(Eq. 3.44)}$$

This interpolation function of Equation 3.44 is an example of tricubic interpolation. Just as the Bezier formulation can be used to interpolate a 1D curve or a 2D surface, the Bezier formulation is being used here to interpolate a 3D solid space.

Like Bezier curves and surfaces, multiple Bezier solids can be joined with continuity constraints across the boundaries. Of course, to enforce positional continuity, adjacent control lattices must share the control points along the boundary plane. As with Bezier curves, $C^1$ continuity can be ensured between two FFD control grids by enforcing colinearity among adjacent control points across the common boundary (Figure 3.69).

Higher-order continuity can be maintained by constraining more of the control points on either side of the common boundary plane. However, for most applications, $C^1$ continuity is sufficient. One possibly useful feature of the Bezier formulation is that a bound on the change in volume induced by FFD can be analytically computed. See Sederberg [36] for details.

FFDs have been extended to include initial grids that are something other than a parallelepiped [13]. For example, a cylindrical lattice can be formed from the standard parallelepiped by merging the opposite boundary planes in one direction and then merging all the points along the cylindrical axis, as in Figure 3.70.

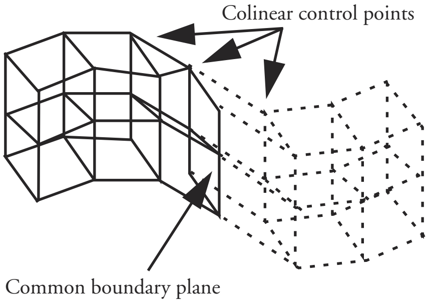


**Figure 3.69** $C^1$ continuity between adjacent control grids

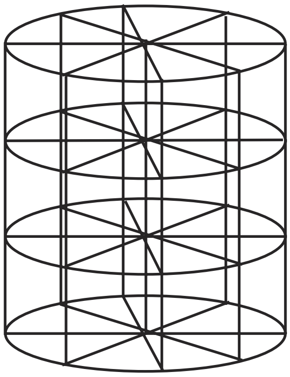


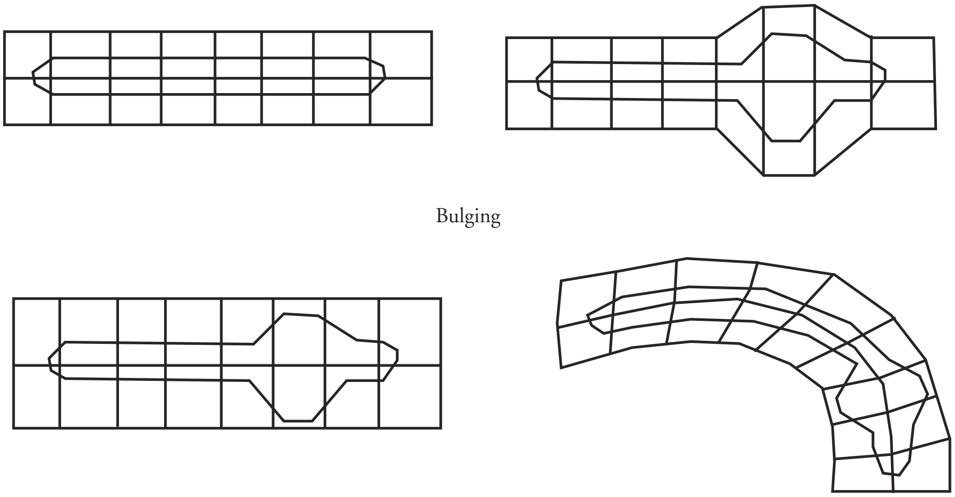
**Figure 3.70** Cylindrical grid

### Compositing FFDs—Sequential versus Hierarchical

FFDs can be composed sequentially or hierarchically. In a sequential composition an object is modeled by progressing through a sequence of FFDs, each of which imparts a particular feature to the object. In this way, various detail elements can be added to an object in stages as opposed to trying to create one mammoth, complex FFD designed to do everything at once. For example, if a bulge is desired on a bent tube, then one FFD can be used to impart the bulge on the surface while a second is designed to bend the object (Figure 3.71).

Organizing FFDs hierarchically allows the user to work at various levels of detail. Finer-resolution FFDs, usually localized, are embedded inside FFDs higher in the hierarchy.[3] As a coarser-level FFD is used to modify the object's vertices, it

[3] For this discussion, the hierarchy is conceptualized with the root note at the top, representing the coarsest level. Finer-resolution, more localized nodes are found lower in the hierarchy.



Bulging

Bending

**Figure 3.71** Sequential FFDs

also modifies the control points of any of its children FFDs that are within the space affected by the deformation. A modification made at a finer level in the hierarchy will remain well defined even as the animator works at a coarser level by modifying an FFD grid higher up in the hierarchy [20] (Figure 3.72).

If an FFD encases only part of an object, then the default assumption is that only those object vertices that are inside the initial FFD grid are changed by the modified FFD grid. Because the finer-level FFDs are typically used to work on a local area of an object, it is useful for the animator to be able to specify the part of the object that is subject to modification by a particular FFD. Otherwise, the rectangular nature of the FFD's grid can make it difficult to delimit the area that the animator actually wants to affect.

### Animated FFDs

Thus far FFDs have been considered as a method to modify the shape of an object by repositioning its vertices. Animation would be performed by, for example, linear interpolation of the object's vertices on a vertex-by-vertex basis. However, FFDs can be used to control the animation in a more direct manner in one of two ways. The FFD can be constructed so that traversal of an object through the FFD space results in a continuous transformation of its shape [14]. Alternatively, the control points of an FFD can be animated, which results in an animated deformation that automatically animates the object's shape.

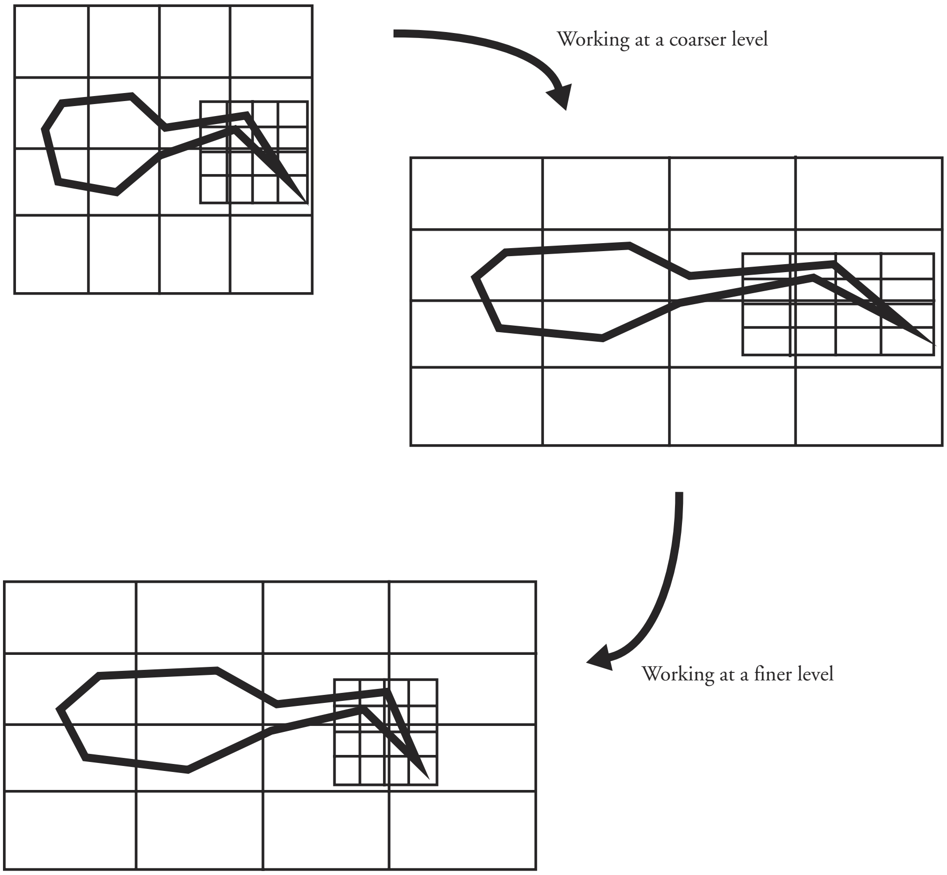
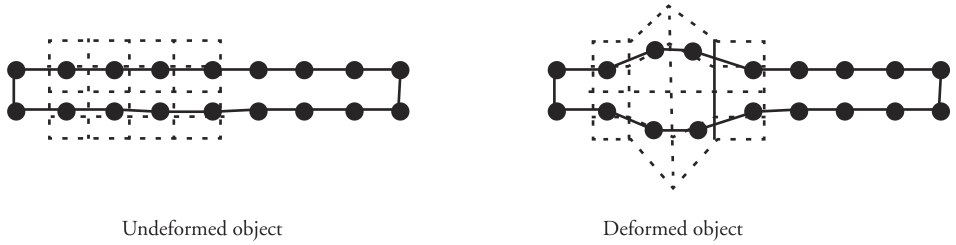


**Figure 3.72** Simple example of hierarchical FFDs

### Deformation Tools

As discussed by Coquillart [14], a *deformation tool* is defined as a composition of an initial lattice and a final lattice. The initial lattice is user defined and is embedded in the region of the model to be animated. The final lattice is a copy of the initial lattice that has been deformed by the user. While the deformation tool may be defined in reference to a particular object, the tool itself is represented in an object-independent manner. This allows for a particular deformation tool to be easily applied to any object (Figure 3.73). To deform an object, the deformation tool must be associated with the object, thus forming what Coquillart calls an *AFFD object*.

**Moving the Tool** A way to animate the object is to specify the motion of the deformation tool relative to the object. In the example of Figure 3.73, the defor-



Undeformed object

Deformed object

**Figure 3.73** Deformation tool applied to an object

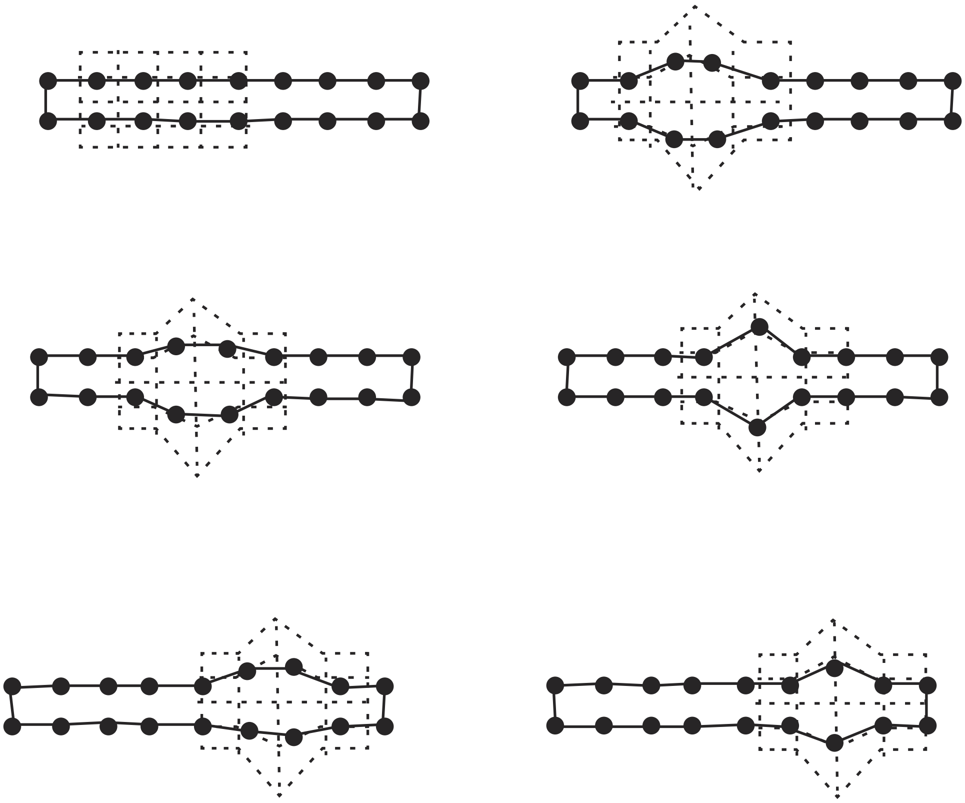


**Figure 3.74** Deformation by translating the deformation tool relative to an object

mation tool could be translated along the object over time. Thus a sequence of object deformations would be generated. This type of animated FFD works effectively when a particular deformation, such as a bulge, progresses across an object (Figure 3.74).

**Moving the Object** Alternatively, the object can translate through the local deformation space of the FFD and, as it does, be deformed by the progression through the FFD grid. The object can be animated independently in global world space while the transformation through the local coordinate grid controls the change in shape of the object. This type of animation works effectively for changing the shape of an object to move along a certain path (e.g., Figure 3.75).

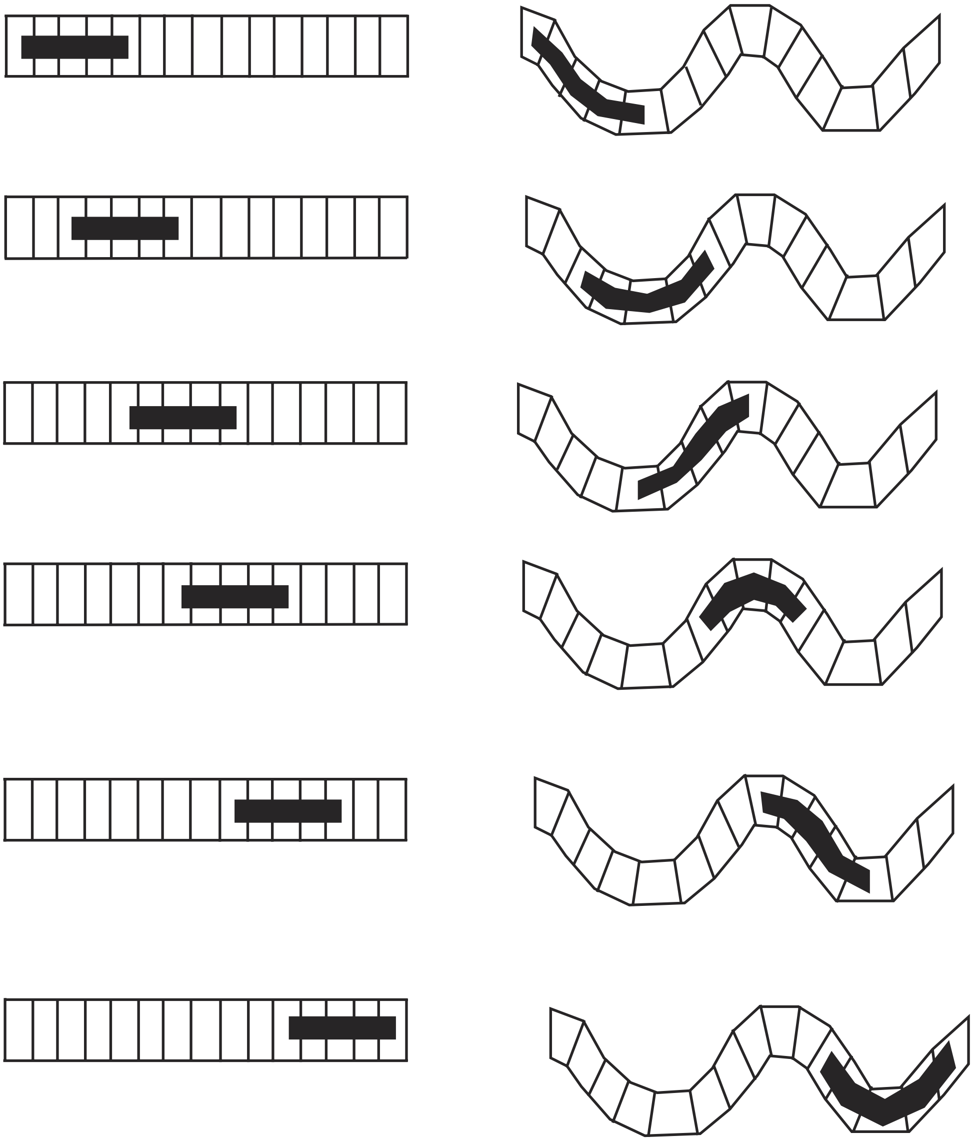
### Animating the FFD

Another way to animate an object using FFDs is to animate the control points of the FFD. For example, the FFD control points can be animated explicitly using key-frame animation, or their movement can be the result of physically based simulation. As the FFD grid points move, they define a changing deformation to be applied to the object's vertices (see Figure 3.76).

Chadwick, Haumann, and Parent [10] describe a layered approach to animation in which FFDs are used to animate a human form. The FFDs are animated in two ways. In the first technique, the positions of the FFD grid vertices are located relative to a wire skeleton the animator uses to move a figure. As the skeleton is manipulated, the grid vertices are repositioned relative to the skeleton automatically. The skin of the figure is then located relative to this local FFD coordinate grid. The FFDs thus play the role of muscular deformation of the skin. Joint articulation modifies the FFD grid, which in turn modifies the surface of the figure. The FFD grid is a mechanism external to the skin that effects the muscle deformation. The "muscles" in this case are meant not to be anatomical representations of real muscles but to provide for a more artistic style.

As a simple example, a hinge joint with adjacent links is shown in Figure 3.77; this is the object to be manipulated by the animator by specifying a joint angle. There is a surface associated with this structure that is intended to represent the skin. There are three FFDs: one for each of the two links and one for the joint. The FFDs associated with the links will deform the skin according to a stylized muscle, and the purpose of the FFD associated with the joint is to prevent interpenetration of the skin surface in highly bent configurations. As the joint bends, the central points of the link FFDs will displace upward and the interior panels of the joint FFD will rotate toward each other at the concave end in order to squeeze the skin together without letting it penetrate itself. Each of the grids is 5x4, and the joint grid is shown using dotted lines so that the three grids can be distinguished. Notice that the grids share a common set of control points where they meet.

Moving the FFD lattice points based on joint angle is strictly a kinematic method. The second technique employed by Chadwick and colleagues [10] uses



Object traversing the logical FFD coordinate space

Object traversing the distorted space

**Figure 3.75** Deformation of an object by passing through FFD space

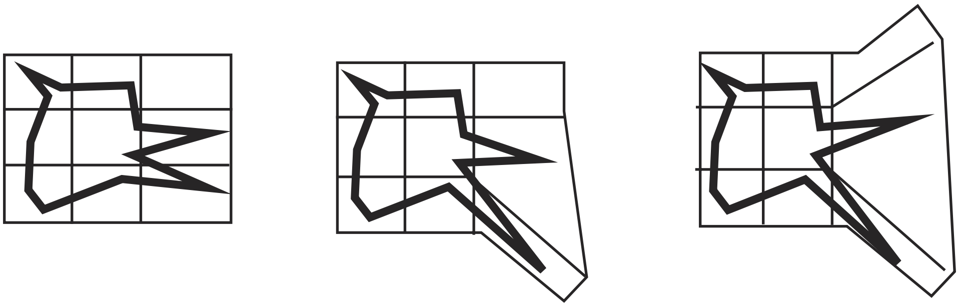


**Figure 3.76** Using an FFD to animate a figure's head

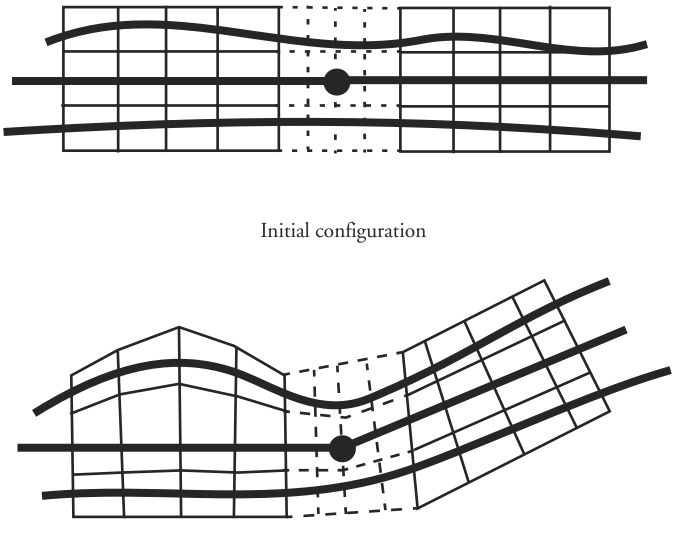


**Figure 3.77** Using FFD to deform a surface around an articulated joint

physically based animation of the FFD lattice points to animate the figure. Animation of the FFD control points is produced by modeling the lattice with springs, dampers, and mass points. The control points of the lattice can then respond to gravity as well as kinematic motion of the figure. To respond to kinematic motion, the center of the FFD is fixed relative to the figure's skeleton. The user kinematically controls the skeleton, and the motion of the skeleton moves the center point of the FFD lattice. The rest of the FFD lattice points react to the movement of this center point via the spring-mass model, and the new positions of the FFD lattice points induce movement in the surface of the figure. This approach animates the clothes and facial tissue of a figure in animations produced by Chadwick [8] [9].

## 3.8 Morphing (2D)

Two-dimensional image metamorphosis has come to be known as *morphing*. Although really an image postprocessing technique, and thus not central to the theme of this book, it has become so well known and has generated so much interest that it demands attention. Typically, the user is interested in transforming one image, called the source image, into the other image, called the destination image. There have been several techniques proposed in the literature for specifying and effecting the transformation. The main task is for the user to specify corresponding elements in the two images; these correspondences are used to control the transformation. Here, two approaches are presented. The first technique is based on user-defined coordinate grids superimposed on each image. These grids impose a coordinate space to relate one image to the other. The second technique is based on pairs of user-defined feature lines, one in each image. The lines mark corresponding features in the two images.

### 3.8.1 Coordinate Grid Approach

To transform one image into another, the user defines a curvilinear grid over each of the two images to be morphed. It is the user's responsibility to define the grids so that corresponding elements in the images are in the corresponding cells of the grids. The user defines the grid by locating the same number of grid intersection points in both images; the grid must be defined at the borders of the images in order to include the entire image (Figure 3.78). A curved mesh is then generated using the grid intersection points as control points for an interpolation scheme such as Catmull-Rom splines.

To generate an intermediate image, say $t$ $(0 < t < 1.0)$, along the way from the source image to the destination image, the vertices (points of intersection of the curves) of the source and destination grids are interpolated to form an intermediate grid. This interpolation can be done linearly, or grids from adjacent key frames can be used to perform higher-order interpolation. Pixels from the source and destination images are stretched and compressed according to the intermediate grid so that warped versions of both the source image and the destination grid are generated. A two-pass procedure is used to accomplish this (described below). A cross dissolve is then performed on a pixel-by-pixel basis between the two warped images to generate the final image. See Figure 3.79.

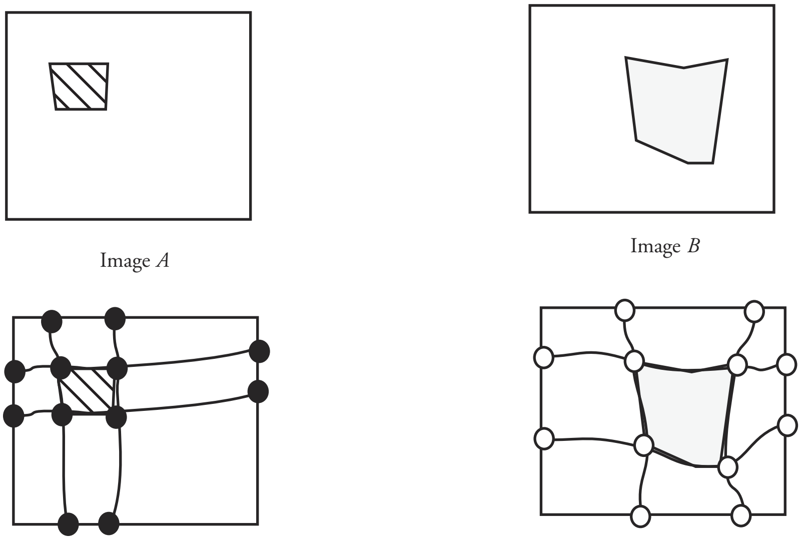


Image *A*

Image *B*

Image *A* with grid points and curves defined

Image *B* with grid points and curves defined

**Figure 3.78** Sample grid definitions

For purposes of explaining the two-pass procedure, it will be assumed that it is the source image to be warped to the intermediate grid, but the same procedure is used to warp the destination image to the intermediate grid.

First, the pixels from the source image are stretched and compressed in the *x*-direction to fit the interpolated grid. These pixels are then stretched and compressed in the *y*-direction to fit the intermediate grid. To carry this out, an auxiliary grid is computed that, for each grid point, uses the *x*-coordinate from the corresponding grid point of the source image grid and the *y*-coordinate from the corresponding point of the intermediate grid. The source image is stretched/compressed in *x* by mapping it to the auxiliary grid, and then the auxiliary grid is used to stretch/compress pixels in *y* to map them to the intermediate grid. In the discussion below, it is assumed the curves are numbered left to right and bottom to top; a curve's number is referred to as its *index*.

Figure 3.80 illustrates the formation of the auxiliary grid from the source image grid and the intermediate grid. Once the auxiliary grid is defined, the first pass uses the source image and auxiliary grids to distort the source pixels in the *x*-direction. For each column of grid points in both the source and the auxiliary grid, a cubic Catmull-Rom spline is defined in pixel coordinates. The leftmost and rightmost columns define straight lines down the sides of the images; this is necessary to include the entire image in the warping process. See the top of Figure 3.81. For each scanline, the *x*-intercepts of the curves with the scanline are computed.

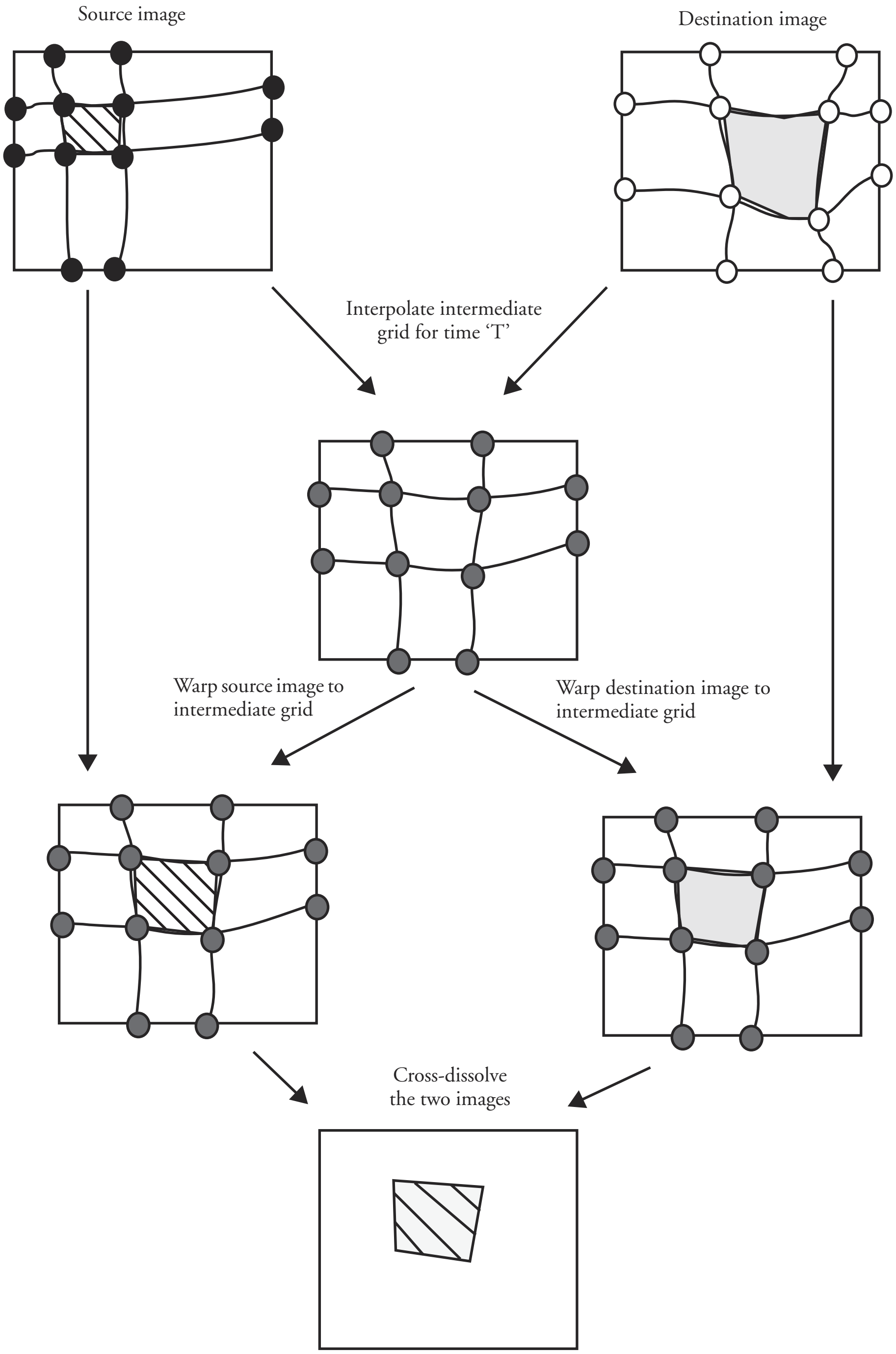

Source image
Destination image
Interpolate intermediate grid for time 'T'
Warp source image to intermediate grid
Warp destination image to intermediate grid
Cross-dissolve the two images

**Figure 3.79** Interpolating to intermediate grid and cross dissolve

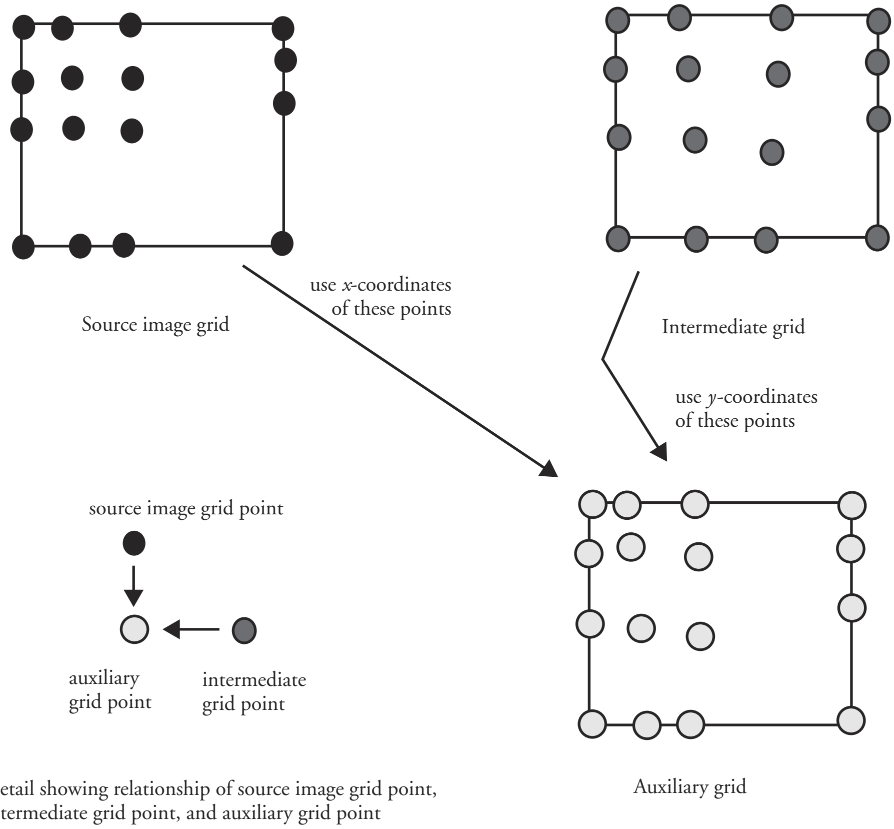
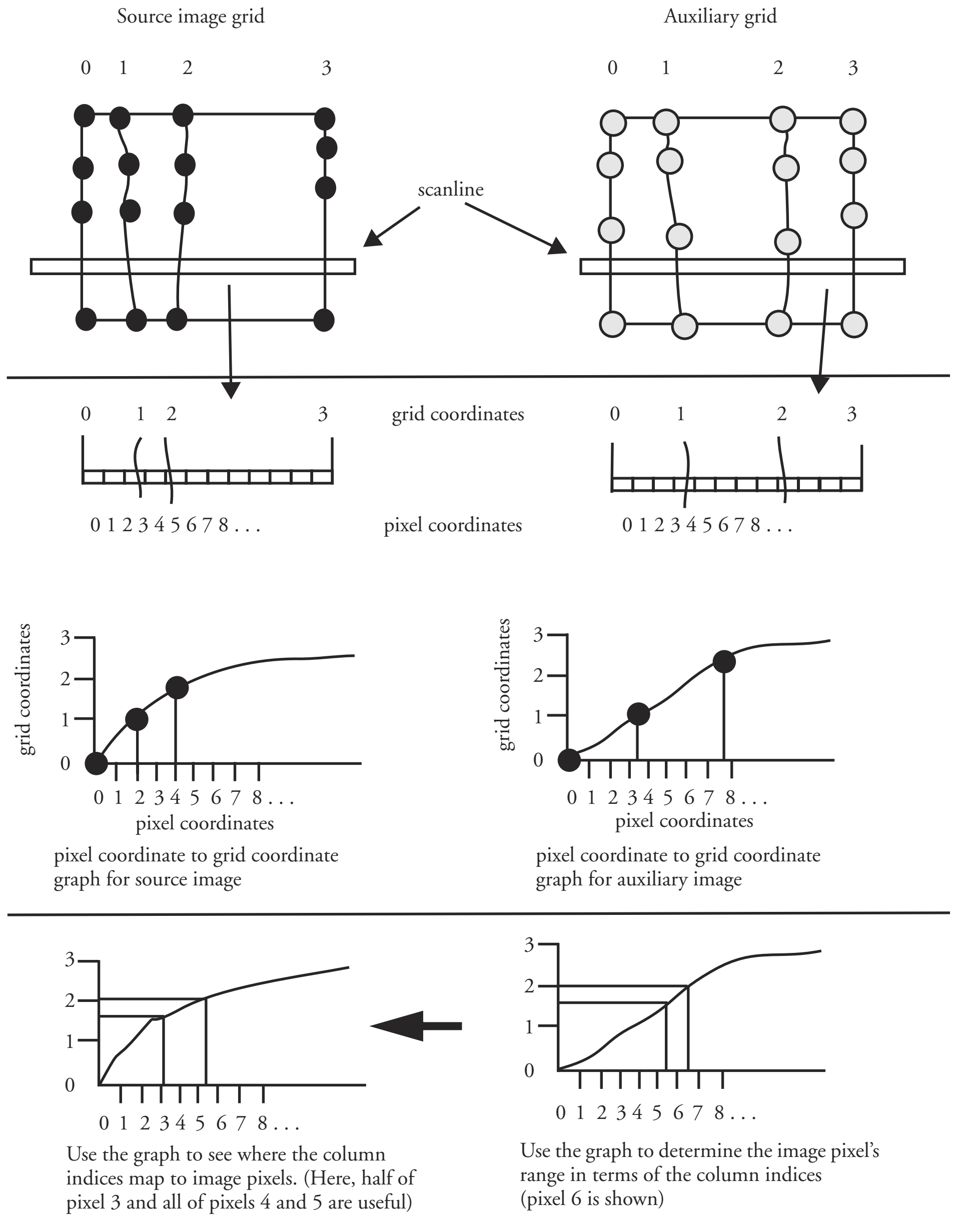

use $x$-coordinates of these points
Source image grid
Intermediate grid
use $y$-coordinates of these points
source image grid point
auxiliary grid point
intermediate grid point
Detail showing relationship of source image grid point, intermediate grid point, and auxiliary grid point
Auxiliary grid

**Figure 3.80** Formation of auxiliary grid for two-pass warping of source image to intermediate grid

These define a grid coordinate system on the scanline. The position of each pixel on the scanline in the source image is determined relative to the $x$-intercepts by computing the Catmull-Rom spline passing through the two-dimensional space of the (grid index, $x$-intercept) pairs. See the middle of Figure 3.81. The integer values of $x$ plus and minus one half, representing the pixel boundaries, can then be located in this space and the fractional index value recorded. In the auxiliary image, the $x$-intercepts of the curves with the scanline are also computed, and for the corresponding scanline, the source image pixel boundaries can be mapped into the intermediate image by using their fractional indices and locating their $x$-positions in the auxiliary scanline. See the bottom of Figure 3.81. Once this is complete, the color of the source image pixel can be used to color in auxiliary pixels by using fractional coverage to effect anti-aliasing.



pixel coordinate to grid coordinate graph for source image

pixel coordinate to grid coordinate graph for auxiliary image

Use the graph to see where the column indices map to image pixels. (Here, half of pixel 3 and all of pixels 4 and 5 are useful)

Use the graph to determine the image pixel's range in terms of the column indices (pixel 6 is shown)

**Figure 3.81** For a given pixel in the auxiliary image, determine the range of pixel coordinates in the source image (for example, pixel 6 of auxiliary grid maps to pixel coordinates 3.5 to 5 of the source image)

The result of the first phase generates colored pixels of an auxiliary image by averaging source image pixel colors on a scanline-by-scanline basis. The second phase repeats the same process on a column-by-column basis by averaging auxiliary image pixel colors to form the intermediate image. The columns are processed by using the horizontal grid curves to establish a common coordinate system between the two images. See Figure 3.82.

This two-pass procedure is applied to both the source and the destination images with respect to the intermediate grid. Once both images have been warped to the same intermediate grid, the important features are presumably (if the user has done a good job of establishing the grids on the two images) in similar positions. At this point the images can be cross-dissolved on a pixel-by-pixel basis. The cross dissolve is merely a blend of the two colors from corresponding pixels.

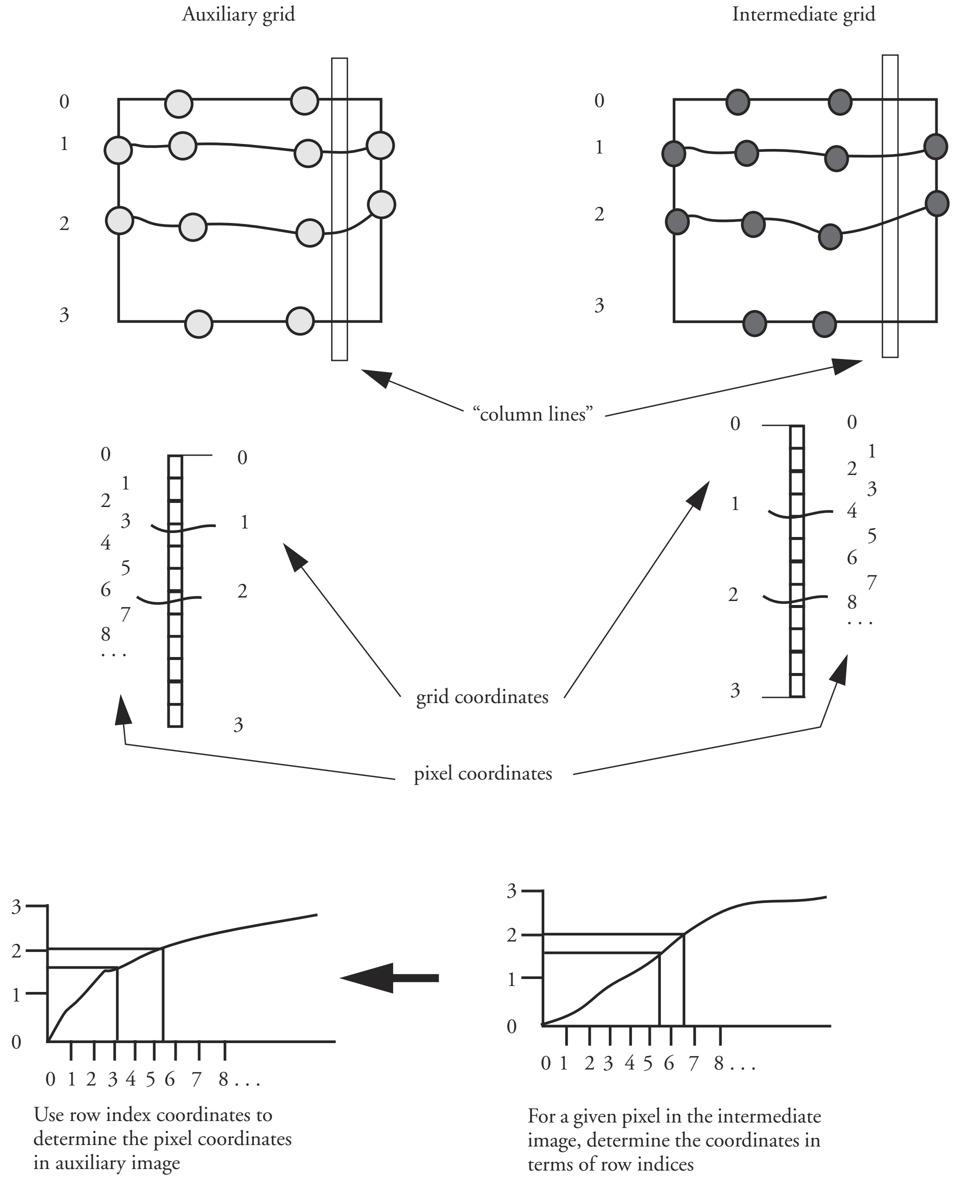
$$C[i][j] = \alpha \cdot C_1[i][j] + (1-\alpha) \cdot C_2[i][j]$$

In the simplest case, alpha might merely be a linear function in terms of the current frame number and the range of frame numbers over which the morph is to take place. However, as Wolberg [41] points out, a nonlinear blend is often more visually appealing. It is also useful to be able to locally control the cross dissolve rates based on aesthetic concerns. For example, in the well-known commercial in which a car morphs into a tiger, the front of the car is morphed into the head of the tiger at a faster rate than the tail to add to the dynamic quality of the animated morph.

Animated images are morphed by the user defining coordinate grids for various key images in each of two animation sequences. The coordinate grids for a sequence are then interpolated over time so that at any one frame in the sequence, a coordinate grid can be produced for that frame. The interpolation is carried out on the *x*-*y* positions of the grid intersection points; cubic interpolation such as Catmull-Rom is typically used. Once a coordinate grid has been produced for corresponding images in the animated sequences, the morphing procedure reduces to the static image case and proceeds according to the description above. See Figure 3.83.

### 3.8.2 Feature-Based Morphing

Instead of using a coordinate grid, the user can establish the correspondence between images by using feature lines [3]. Lines are drawn on the two images to identify features that correspond to one another; and feature lines are interpolated to form an intermediate feature line set. The interpolation can be based either on interpolating endpoints or on interpolating center points and orientation. In either case, a mapping for each pixel in the intermediate image is established to each interpolated feature line, and a relative weight is computed that indicates the



Use row index coordinates to determine the pixel coordinates in auxiliary image

For a given pixel in the intermediate image, determine the coordinates in terms of row indices

**Figure 3.82** Establishing the auxiliary pixel range for a pixel of the intermediate image (for example, pixel 6 of the intermediate grid maps to pixel coordinates 3.5 to 5 of the auxiliary image)

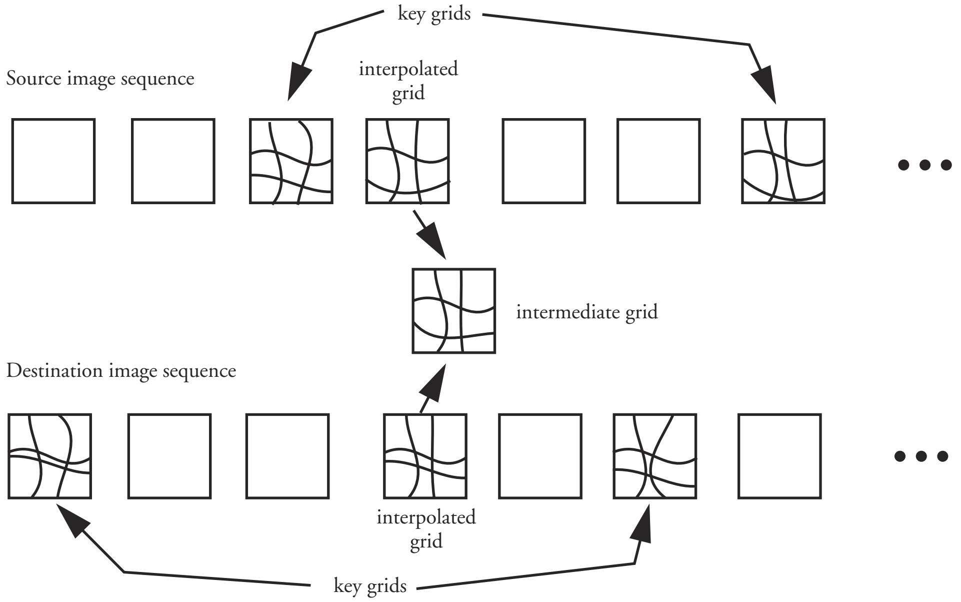
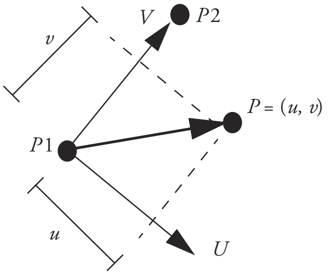


**Figure 3.83** Morphing of animated sequences

amount of influence that feature line should have on the pixel. The mapping is used in the source image to locate the source image pixel that corresponds to the intermediate image pixel. The relative weight is used to average the source image locations generated by multiple feature lines into a final source image location. This location is used to determine the color of the intermediate image pixel. This same procedure is used on the destination image to form its intermediate image. These intermediate images are then cross-dissolved to form the final intermediate image.

Consider the mapping established by a single feature line, defined by two endpoints and oriented from $P_1$ to $P_2$. In effect, the feature line establishes a local two-dimensional coordinate system $(U, V)$ over the image. For example, the first point of the line can be considered the origin. The second point of the line establishes the unit distance in the positive $V$-axis direction and scale. A line perpendicular to this line and of unit length extending to its right (as one stands on the first point and looks toward the second point) establishes the $U$-axis direction and scale. The coordinates $(u, v)$ of a pixel relative to this feature line can be found by simply computing the pixel's position relative to the $U$- and $V$-axes of a local coordinate system defined by that feature line. Variable $v$ is the projection of $(P - P_1)$ onto the direction of $(P_2 - P_1)$, normalized to the length of $(P_2 - P_1)$. $u$ is calculated similarly. See Figure 3.84.



$$v = (P - P1) \bullet \frac{(P2 - P1)}{|P2 - P1|^2}$$

$$u = \left|(P - P1) \times \frac{(P2 - P1)}{|P2 - P1|^2}\right|$$

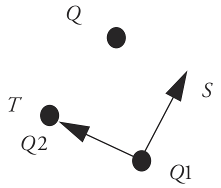
**Figure 3.84** Local coordinate system of a feature in the intermediate image

Assume that the points $P_1$ and $P_2$ are selected in the intermediate image and used to determine the (*u, v*) coordinates of a pixel. Given the corresponding feature line in the source image defined by the points *Q*1 and *Q*2, a similar 2D coordinate system, (*S, T*), is established. Using the intermediate pixel's *u*-, *v*-coordinates relative to the feature line, one can compute its corresponding location in the source image (Figure 3.85).

To transform an image by a single feature line, each pixel of the intermediate image is mapped back to a source image position according to the equations above. The colors of the source image pixels in the neighborhood of that position are then used to color in the pixel of the intermediate image. See Figure 3.86.

Of course, the mapping does not typically transform an intermediate image pixel back to the center of a source image pixel. The floating point coordinates of the source image location could be rounded to the nearest pixel coordinates, which would introduce aliasing artifacts. To avoid such artifacts in the intermediate image, the corner points of the intermediate pixel could be mapped back to the source image, which would produce a quadrilateral area in the source image; the pixels, wholly or partially contained in this quadrilateral area, would contribute to the color of the destination pixel.

The mapping described so far is a subset of the affine transformations. For image pairs to be really interesting and useful, multiple line pairs must be used to



$$T = Q2 - Q1$$

$$S = (T_y, -T_x)$$

$$Q = Q1 + u \cdot S + v \cdot T$$

**Figure 3.85** Relocating a point's position using local coordinates in the source image

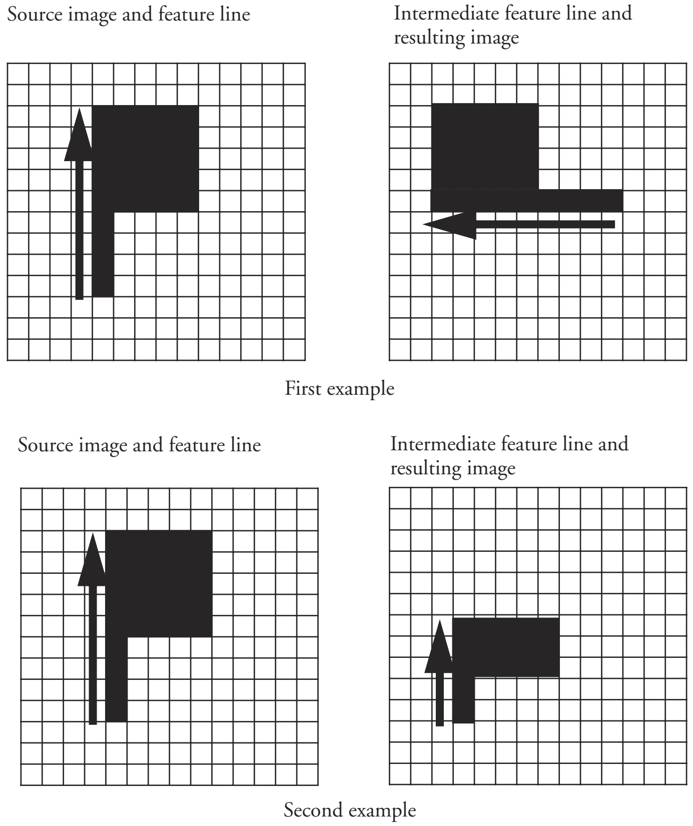

Source image and feature line
Intermediate feature line and resulting image
First example
Source image and feature line
Intermediate feature line and resulting image
Second example

**Figure 3.86** Two examples of single-feature line morphing

establish correspondences between multiple features in the images. For a pair of images with multiple feature lines, each feature line pair produces a displacement vector from an intermediate image pixel to its source image position. Associated with this displacement is a weight based on the pixel's position relative to the feature line in the intermediate image. The weight presented by Beier and Neely [3] is shown in Equation 3.45.

$$w = \left(\frac{|Q2 - Q1|^{p}}{a + dist}\right)^{b} \qquad \textbf{(Eq. 3.45)}$$

The line is defined by point $Q1$ and $Q2$, and *dist* is the distance that the pixel is from the line. The distance is measured from the finite line segment defined by $Q1$ and $Q2$ so that if the perpendicular projection of $P$ onto the infinite line defined by $Q1$ and $Q2$ falls beyond the finite line segment, then the distance is taken to be the distance to the closer of the two endpoints. Otherwise, the distance is the perpendicular distance to the finite line segment. User-supplied parameters (*a, b, p* in Equation 3.45) control the overall character of the mapping. As *dist* increases, *w* decreases but never goes to zero; as a practical matter, a lower limit could be set below which *w* is clamped to zero and the feature line's effect on the point is ignored above a certain distance. If *a* is nearly zero, then pixels on the line are rigidly transformed with the line. Increasing *a* makes the effect of lines over the image smoother. Increasing *p* increases the effect of longer lines. Increasing *b* makes the effect of a line fall off more rapidly. As presented here, these parameters are global; for more precise control these parameters could be set on a feature-line-by-feature-line basis. For a given pixel in the intermediate image, the displacement indicated by each feature line pair is scaled by its weight. The weights and the weighted displacements are accumulated. The final accumulated displacement is then divided by the accumulated weights. This gives the displacement from the intermediate pixel to its corresponding position in the source image. See the code segment in Figure 3.87.

When morphing between two images, the feature lines are interpolated over some number of frames. For any one of the intermediate frames, the feature line induces a mapping back to the source image and forward to the destination image. The corresponding pixels from both images are identified and their colors blended to produce a pixel of the intermediate frame. In this way, feature-based morphing produces a sequence of images that transform from the source image to the destination image.

The transformations implied by the feature lines are fairly intuitive, but some care must be taken in defining transformations with multiple line pairs. Pixels that lie on a feature line are mapped onto that feature line in another image. If feature lines cross in one image, pixels at the intersection of the feature lines are mapped to both feature lines in the other image. This situation essentially tries to pull apart the image and can produce unwanted results. Also, some configurations of feature lines can produce nonintuitive results. Other techniques in the literature (e.g., [25]) have suggested algorithms to alleviate these shortcomings.

## 3.9 3D Shape Interpolation

Changing one 3D object into another 3D object is a useful effect, but one with problems for which general-purpose solutions are still being developed. Several solutions exist that have various advantages and disadvantages. The techniques fall into one of two categories: surface based or volume based. The surface-based techniques use the boundary representation of the objects and modify one or both of them so that the vertex-edge topologies of the two objects match. Once this is

```
// ====================================================================

// XY structure
typedef struct xy_struct {
  float x,y;
} xy_td;

// FEATURE
//   line in image1: p1,p2;
//   line in image2: q1,q2
//   weights used in mapping: a,b,p
//   length of line in image2
typedef struct feature_struct {
  xy_td p1,p2,q1,q2;
  float  a,b,p;
  float  plength,qlength;
} feature_td;

// FEATURE LIST
typedef struct featureList_struct {
  int  num;
  feature_td  *features;
} featureList_td;

// -------------------------------------------------------------------
//   MORPH
// -------------------------------------------------------------------
void morph(featureList_td *featureList)
{
  float  a,b,p,length;
  xy_td  p1,p2,q1,q2;

  xy_td  vp,wp,vq,v,qq;
  int    ii,jj,indexS,indexD;
  float  idisp,jdisp;
  float  t,s,vx,vy;
  float  weight;
  char    background[3];
  float  fcolor[3];

  background[0] = 120;
  background[1] = 20;
  background[2] = 20;

  for (int i=0; i<HEIGHT; i++) {
    for (int j=0; j<WIDTH; j++) {
     fcolor[0] = 0.0; fcolor[1] = 0.0; fcolor[2] = 0.0;
     weight = 0;
     for (int k=0; k<featureList->num; k++) {
      // get info about kth feature line
     a = featureList->features[k].a;
     b = featureList->features[k].b;
     p = featureList->features[k].p;
     p1.x = featureList->features[k].p1.x;
     p1.y = featureList->features[k].p1.y;
     p2.x = featureList->features[k].p2.x;
     p2.y = featureList->features[k].p2.y;
     q1.x = featureList->features[k].q1.x;
     q1.y = featureList->features[k].q1.y;
     q2.x = featureList->features[k].q2.x;
     q2.y = featureList->features[k].q2.y;
     length = featureList->features[k].qlength;

     // get local feature coordinate system in image1
     vp.x = p2.x-p1.x;
     vp.y = p2.y-p1.y;
     wp.x = vp.y;
     wp.y = -vp.x;

     // get feature vector in image2
     vq.x = q2.x-q1.x;
     vq.y = q2.y-q1.y;

     // get vector from first feature point to pixel (image2)
     v.x = j-q1.x;
     v.y = i-q1.y;

     // get perpendicular distance from feature line to pixel (image2)
     s = (v.x*vq.x + v.y*vq.y)/(length*length);
     t = (v.x*vq.y-v.y*vq.x)/(length*length);

     // use (s,t) and vp, wp to map to point in image1 space
     jj = (int)(p1.x + s*vp.x + t*wp.x);
     ii = (int)(p1.y + s*vp.y + t*wp.y);
     // printf("\n %d,%d",ii,jj);

     t = length/(a+t);
     jdisp += (jj-j)*t;
     idisp += (ii-i)*t;
     weight += t;
    }
    jdisp /= weight;
    idisp /= weight;
    ii = (int)(i+idisp);
    jj = (int)(j+jdisp);
    indexD = (WIDTH*i+j)*3;
    if ( (ii<0) || (ii>=HEIGHT) || (jj<0) || (jj>=WIDTH) ) {
     image2[indexD] = background[0];
     image2[indexD+1] = background[1];
     image2[indexD+2] = background[2];
    }
    else {
     indexS = (WIDTH*ii+jj)*3;
     image2[indexD] = image1[indexS];
     image2[indexD+1] = image1[indexS+1];
     image2[indexD+2] = image1[indexS+2];
    }

   }
  }
  // same image2
}
```

**Figure 3.87** Code using feature lines to morph from source to destination image

done, the vertices of the object can be interpolated on a vertex-by-vertex basis. Surface-based techniques usually have some restriction on the types of objects they can handle, especially objects with holes through them. The number of holes through an object is an important attribute of an object's structure, or *topology.* The volume-based techniques consider the volume contained within the objects and blend one volume into the other. These techniques have the advantage of being less sensitive to different object topologies. However, volume-based techniques usually require volume representations of the objects and therefore tend to be more computationally intensive than surface-based approaches. Volume-based approaches will not be discussed further.

The terms used in this discussion are defined by Kent, Carlson, and Parent [22] and Weiler [40]. *Object* refers to an entity that has a 3D surface geometry; the *shape* of an object refers to the set of points in object space that make up the object's surface; and *model* refers to any complete description of the shape of an object. Thus a single object may have several different models that describe its shape. The term *topology* has two meanings, which can be distinguished by the context in which they are used. The first meaning, from traditional mathematics, is the connectivity of the surface of an object. For present purposes, this use of *topology* is taken to mean the number of holes an object has and the number of separate bodies represented. A doughnut and a teacup have the same topology and are said to be *topologically equivalent.* A beach ball and a blanket have the same topology. Two objects are said to be homeomorphic (or topologically equivalent) if there exists a continuous, invertible, one-to-one mapping between the points on the surfaces of the two objects. The *genus* of an object refers to how many holes, or passageways, there are through it. A beach ball is a genus 0 object; a teacup is a genus 1 object. The second meaning of the term *topology,* popular in the computer graphics literature, refers to the vertex/edge/face connectivity of an object; objects that are equivalent in this form of topology are the same except for the *x*-, *y*-, *z*-coordinate definitions of their vertices (the *geometry* of the object).

For most approaches, the shape transformation problem can be discussed in terms of the two subproblems: (1) the *correspondence problem,* or establishing the mapping from a vertex (or other geometric element) on one object to a vertex (or geometric element) on the other object; and (2) the *interpolation problem,* or creating a sequence of intermediate objects that visually represent the transformation of one object into the other. The two problems are related because the elements that are interpolated are typically the elements between which correspondences are established.

In general, it is not enough to merely come up with a scheme that transforms one object into another. An animation tool must give the user some control over which areas of one object map to which areas of the other object. This control mechanism can be as simple as aligning the object using affine transformations, or it can be as complex as allowing the user to specify an unlimited number of point correspondences between the two objects. A notable characteristic of the various algorithms for shape interpolation is the use of topological information versus geometric information. Topological information considers the logical construction of the objects and, when used, tends to minimize the number of new vertices and edges generated in the process. Geometric information considers the spatial extent of the object and is useful for relating the position in space of one object to the position in space of the other object.

While many of the techniques discussed here are applicable to sculptured surfaces, they are discussed in terms of planar polyhedra because these are the subject of shape interpolation procedures in the majority of circumstances.

### 3.9.1 Matching Topology

The simplest case of transforming one object into another is when the two objects to be interpolated share the same vertex-edge topology. Here, the objects are transformed by merely interpolating the positions of vertices on a vertex-by-vertex basis. As an example, this case arises when one of the previously discussed techniques, such as FFD, has been used to modify the shape of one object without modifying the vertex-edge connectivity to produce the second object. The correspondence between the two objects is established by the vertex-edge connectivity structure shared by the two objects. The interpolation problem is solved, as is the case in the majority of techniques presented here, by interpolating 3D vertex positions.

### 3.9.2 Star-Shaped Polyhedra

If the two objects are both *star-shaped*[4] polyhedra, then polar coordinates can be used to induce a 2D mapping between two objects. See Figure 3.88 for the two-dimensional equivalent. The object surfaces are sampled by a regular distribution of rays emanating from a central point in the *kernel* of the object, and vertices of an intermediate object are constructed by interpolating between the intersection points along a ray. A surface definition is then constructed from the vertices by forming the polygons of a regular polytope from the surface vertices. The vertices making up each surface polygon can be determined as a preprocessing step and are only dependent on how the rays are distributed in polar space. Figure 3.89 illustrates the sampling and interpolation for objects in two dimensions. The extension

[4] A *star-shaped* (2D) polygon is one in which there is at least one point from which a line can be drawn to any point on the boundary of the polygon without intersecting the boundary; a star-shaped (3D) polyhedron is similarly defined. The set of points from which the entire boundary can be seen is referred to as the *kernel* of the polygon (in the 2D case) or the polyhedron (in the 3D case).

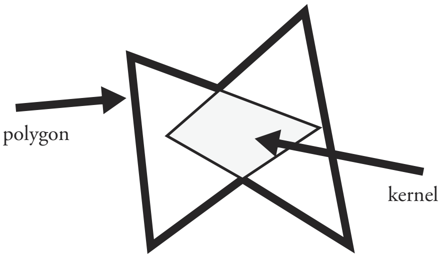
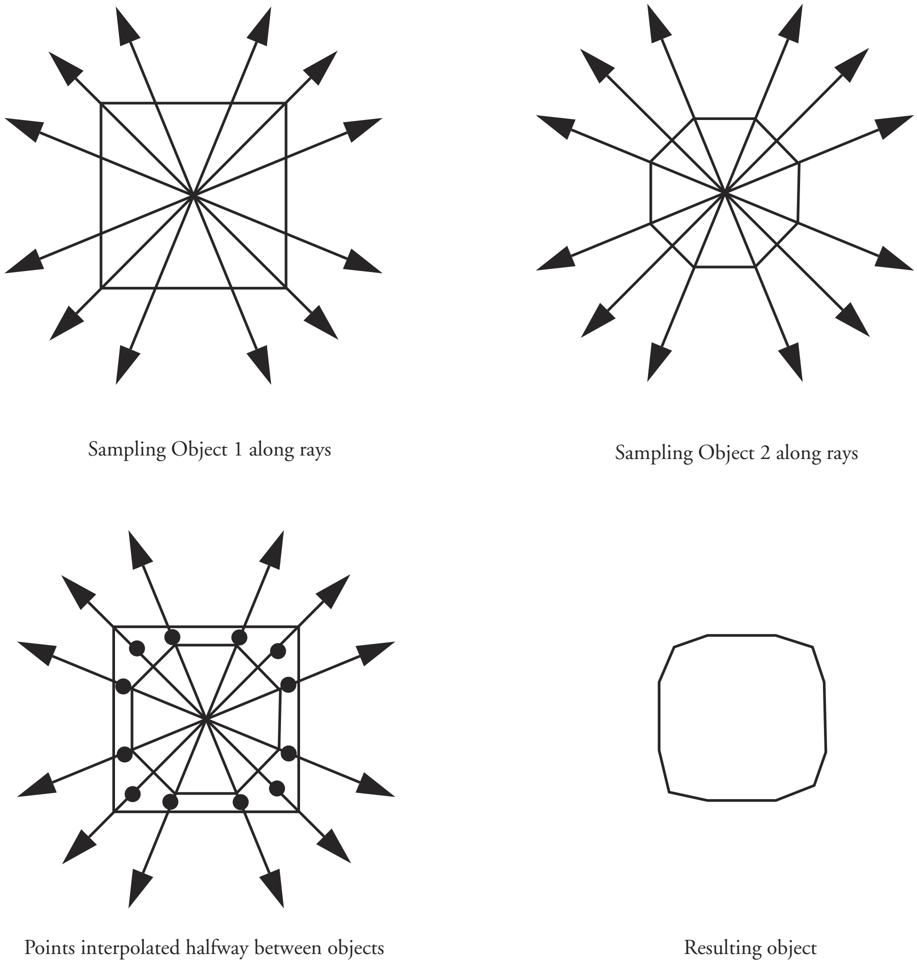


**Figure 3.88** Star-shaped polygon and corresponding kernel


Sampling Object 1 along rays

Sampling Object 2 along rays

Points interpolated halfway between objects

Resulting object

**Figure 3.89** Star-shaped polyhedral shape interpolation

to interpolating in three dimensions is straightforward. In the three-dimensional case, polygon definitions on the surface of the object must then be formed.

### 3.9.3 Axial Slices

Chen and Parent [11] interpolate objects that are star shaped with respect to a central axis. For each object, the user defines an axis that runs through the middle of the object. At regular intervals along this axis, perpendicular slices are taken of the object. These slices must be star shaped with respect to the point of intersection between the axis and the slice. This central axis is defined for both objects, and the part of each axis interior to its respective object is parameterized from zero to one. In addition, the user defines an orientation vector (or a default direction is used) that is perpendicular to the axis. See Figure 3.90.

Corresponding slices (corresponding in the sense that they use the same axis parameter to define the plane of intersection) are taken from each object. All of the slices from one object can be used to reconstruct an approximation to the original object using one of the contour-lofting techniques (e.g., [12] [16]). The 2D slices can be interpolated pairwise (one from each object) by constructing rays that emanate from the center point and sample the boundary at regular intervals with respect to the orientation vector (Figure 3.91).

The parameterization along the axis and the radial parameterization with respect to the orientation vector together establish a 2D coordinate system on the surface of the object. Corresponding points on the surface of the object are located in three-space. The denser the sampling, the more accurate the approximation to the original object. The corresponding points can then be interpolated in three-space. Each set of ray-polygon intersection points from the pair of corre-

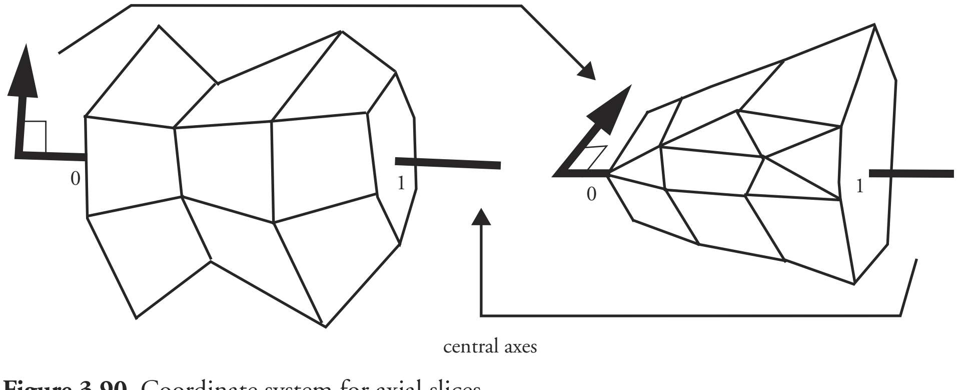


**Figure 3.90** Coordinate system for axial slices

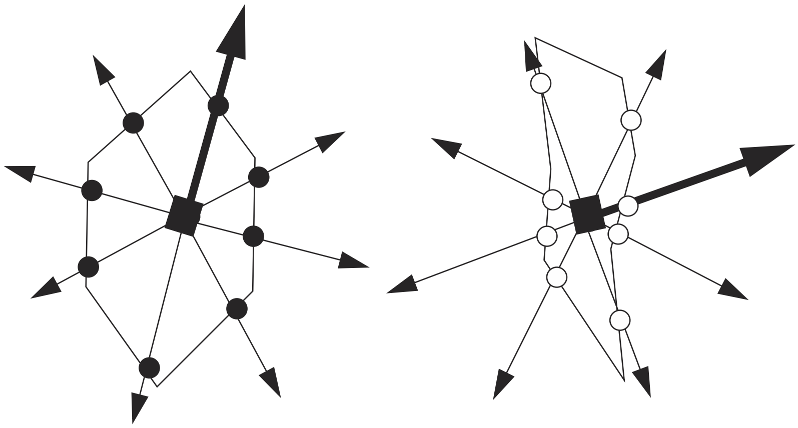


**Figure 3.91** Projection lines for star-shaped polygons from objects in Figure 3.90

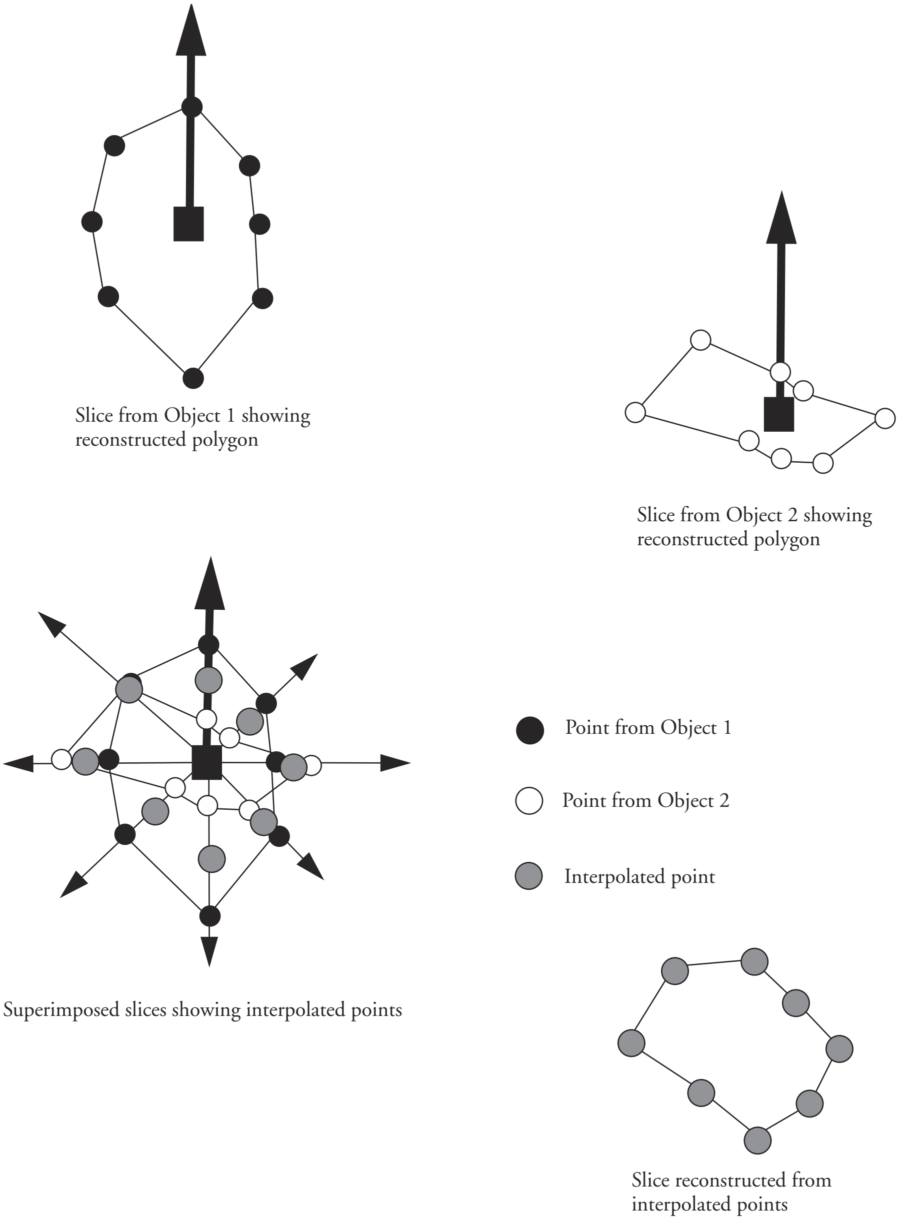
sponding slices is used to generate an intermediate slice based on an interpolation parameter (Figure 3.92). Linear interpolation is often used, although higher-order interpolations are certainly useful. See Figure 3.93 for an example from Chen and Parent [11].

This approach can also be generalized somewhat to allow for a segmented central axis, consisting of a linear sequence of adjacent line segments. The approach may be used as long as the star-shaped restriction of any slice is maintained. The parameterization along the central axis is the same as before, except this time the central axis consists of multiple line segments.

### 3.9.4 Map to Sphere

Even among genus 0 objects, more complex polyhedra may not be star shaped or allow an internal axis (single or multisegment) to define star-shaped slices. A more complicated mapping procedure may be required to establish the two-dimensional parameterization of the objects' surfaces. One approach is to map both objects onto a common surface such as a unit sphere [22]. The mapping must be such that the entire object surface maps to the entire sphere with no overlap (i.e., it must be one-to-one and onto). Once both objects have been mapped onto the sphere, a union of their vertex-edge topologies can be constructed and then inversely mapped back onto each original object. This results in a new model for each of the original shapes, but the new models now have the same topologies. These new definitions for the objects can then be transformed by a vertex-by-vertex interpolation.

There are several different ways to map an object to a sphere. No one way has been found to work for all objects, but, taken together, most of the genus 0 objects



Slice from Object 1 showing reconstructed polygon

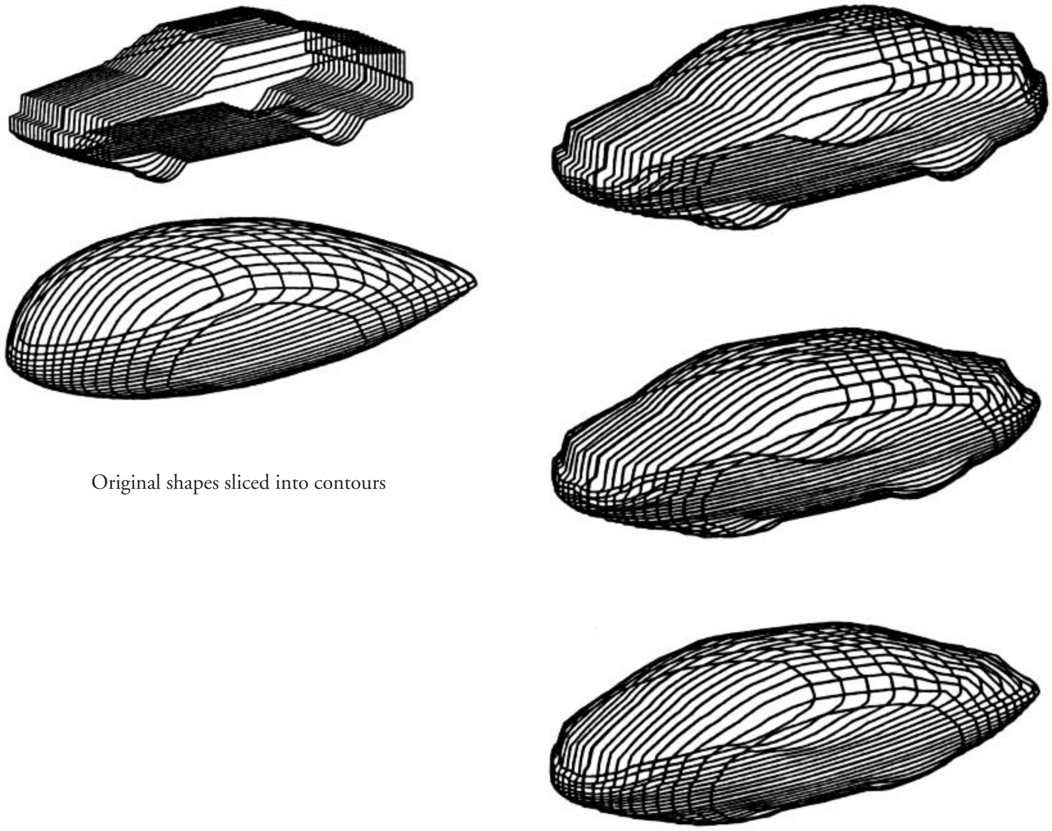
Slice from Object 2 showing reconstructed polygon

Superimposed slices showing interpolated points

Slice reconstructed from interpolated points

**Figure 3.92** Interpolating slices taken from two objects along each respective central axis

can be successfully mapped to a sphere. The most obvious way is to project each vertex and edge of the object away from a center point of the object onto the sphere. This, of course, works fine for star-shaped polyhedra but fails for others.



Original shapes sliced into contours

Interpolated shapes

**Figure 3.93** 3D shape interpolation from multiple 2D slices [11]

Another approach fixes key vertices to the surface of the sphere. These vertices are either selected by the user or automatically picked by being topmost, bottommost, leftmost, and so on. A spring-damper model is then used to force the remaining vertices to the surface of the sphere while minimizing edge length.

If both objects are successfully mapped to the sphere's surface (i.e., no overlap), the projected edges are intersected and merged into one topology. The new vertices and edges are a superset of both object topologies. They are then projected back onto both object surfaces. This produces two new object definitions, identical in shape to the original objects but now having the same vertex-edge topology, allowing for a vertex-by-vertex interpolation to transform one object into the other.

Once the vertices of both objects have been mapped onto a unit sphere, edges map to circular arcs on the surface of the sphere. The following description of the algorithm to merge the topologies follows that found in Kent, Carlson, and Parent [22]. It assumes that the faces of the models have been triangulated prior to projection onto the sphere and that degenerate cases, such as the vertex of one object projecting onto the vertex or edge of the other object, do not occur; these can be handled by relatively simple extensions to the algorithm.

To efficiently intersect each edge of one object with edges of the second object, it is necessary to avoid a brute force edge-edge comparison for all pairs of edges. While this approach would work theoretically, it would, as noted by Kent, be very time-consuming and subject to numerical inaccuracies that may result in intersection points being erroneously ordered along an edge. Correct ordering of intersections along an edge is required by the algorithm.

In the following discussion on merging the topologies of the two objects, all references to vertices, edges, and faces of the two objects refer to their projection on the unit sphere. The two objects are referred to as Object *A* and Object *B.* Subscripts on vertex, edge, and face labels indicate which object they come from. Each edge will have several lists associated with it: an intersection list, a face list, and an intersection-candidate list.

The algorithm starts by considering one vertex, $V_A$, of Object *A* and finding the face, $F_B$, of Object *B* that contains vertex $V_A$. See Figure 3.94. Taking into account that it is operating within the two-dimensional space of the surface of the unit sphere, the algorithm can achieve this result quite easily and quickly.

The edges emanating from $V_A$ are added to the work list. Face $F_B$ becomes the current face, and all edges of face $F_B$ are put on each edge's intersection-candidate list. This phase of the algorithm has finished when the work list has been emptied of all edges.

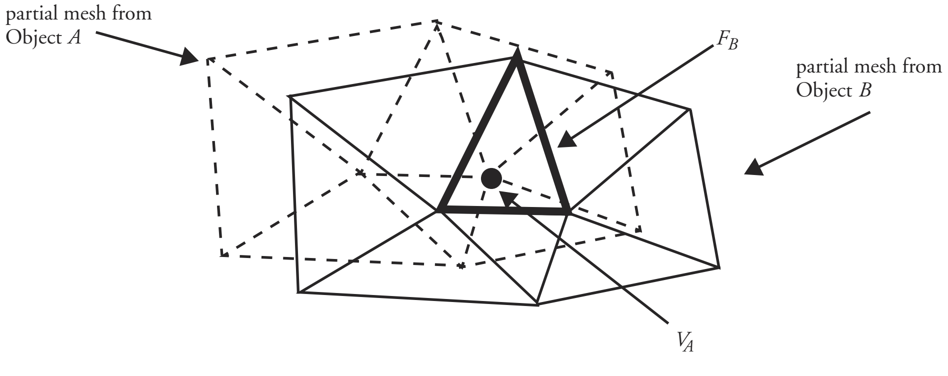


**Figure 3.94** Locating initial vertex of Object *A* in the face of Object *B*

An edge, $E_A$, and its associated intersection-candidate list are taken from the work list. The edge $E_A$ is tested for any intersection with the edges on its intersection-candidate list. If no intersections are found, intersection processing for edge $E_A$ is complete and the algorithm proceeds to the intersection-ordering phase. If an intersection, *I,* is found with one of the edges, $E_B$, then the following steps are done: *I* is added to the final model; I is added to both edge $E_A$'s intersection list and edge $E_B$'s intersection list; the face, $G_B$, on the other side of edge $E_B$ becomes the current face; and the other edges of face $G_B$ (the edges not involved in the intersection) replace the edges in edge $E_A$'s intersection-candidate list. In addition, to facilitate the ordering of intersections along an edge, pointers to the two faces from Object *A* that share $E_A$ are associated with *I.* This phase of the algorithm then repeats by considering the edges on the intersection-candidate list for possible intersections and, if any are found, processes subsequent intersections. When this phase is complete all edge-edge intersections have been found. See Figure 3.95.

For Object *A,* the intersections have been generated in sorted order along the edge. However, for Object *B,* the intersections have been associated with the edges but have been recorded in essentially a random order along the edge. The intersection-ordering phase uses the faces associated with intersection points and the list of intersections that have been associated with an edge of Object *B* to sort the intersection points along the edge. The face information is used because numerical inaccuracies can result in erroneous orderings if the numeric values of only parameters or intersection coordinates are used to order the intersections along the edge.

One of the defining vertices, $V_B$, for an edge, $E_B$, from Object *B,* is located within a face, $F_A$, of Object *A*. Initially, face $F_A$ is referred to as the *current face.* As a result of the first phase, the edge already has all of its intersections recorded on its intersection list. Associated with each intersection point are the faces of Object *A* that share the edge $E_B$ intersected to produce the point. The intersection points are searched to find the one that lists the current face, $F_A$, as one of its faces; one and only one intersection point will list $F_A$. This intersection point is the first intersec-

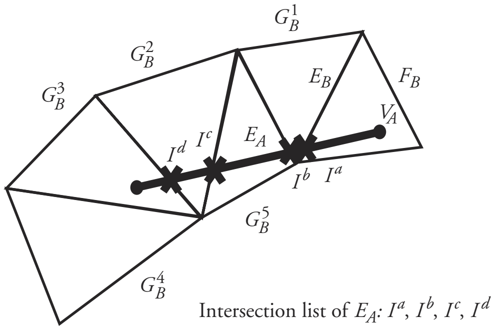


**Figure 3.95** The intersection list for edge $E_A$

tion along the edge, and the other face associated with the intersection point will become the current face. The remaining intersection points are searched to see which lists this new current face; it becomes the next intersection point, and the other face associated with it becomes the new current face. This process continues until all intersection points have been ordered along edge $E_B$.

All of the information necessary for the combined models has been generated. It needs to be mapped back to each original object, and new polyhedral definitions need to be generated. The intersections along edges, kept as parametric values, can be used on the original models. The vertices of Object *A,* mapped to the surface of the sphere, need to be mapped onto the original Object *B* and vice versa for vertices of Object *B.* This can be done by computing the barycentric coordinates of the vertex with respect to the triangle that contains it (see Appendix B for details). These barycentric coordinates can be used to locate the point on the original object surface.

Now all of the vertices, edges, and intersection points from both objects have been mapped back onto each object. New face definitions need to be constructed for each object. Because both models started out as triangulated meshes, there are only a limited number of configurations possible when one considers the retriangulation required from the combined models (Figure 3.96). Processing can proceed by considering all of the original triangles from one of the models. For each triangle, use the intersection points along its edges and the vertices of the other object that are contained in it (along with any corresponding edge definitions) and construct a new triangulation of the triangle. When this is finished, repeat the process with the next triangle from the object.

The process of retriangulating a triangle proceeds as follows. First, output any complete triangles from the other object contained in this object. Second, retriangulate the triangle fragments that include an edge or edge segment of the original triangle. Start at one of the vertices of the triangle and go to the first intersection encountered along the boundary of the triangle. The procedure progresses around the boundary of the triangle and ends when it returns to this intersection point. The next intersection along the boundary is also obtained. These two intersections are considered and the configuration identified by noting whether zero, one, or two original vertices are contained in the boundary between the two intersection points. The element inside the triangle that connects the two vertices/intersections is either a vertex or an edge of the other object (along with the two edge segments involved in the intersections). See Figure 3.96. Once the configuration is determined, retriangulating the region is a simple task. The procedure then continues with the succeeding vertices or intersections in the order that they appear around the boundary of the original triangle.

This completes the retriangulation of the combined topologies on the sphere. The resulting mesh can then be mapped back onto the surfaces of both objects,

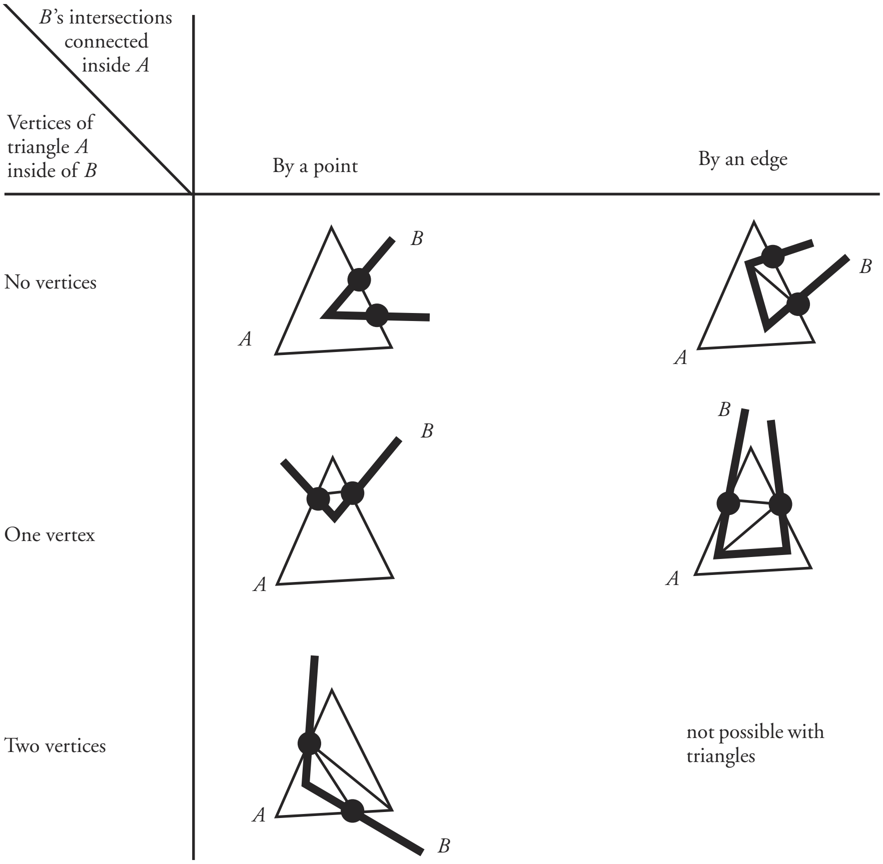

*B*'s intersections connected inside *A*
Vertices of triangle *A* inside of *B*
By a point
By an edge
No vertices
One vertex
Two vertices
not possible with triangles
A
B

**Figure 3.96** Configurations possible with overlapping triangles and possible triangulations

which establishes new definitions of the original objects but with the same vertex-edge connectivity. Notice that geometric information is used in the various mapping procedures in an attempt to map similarly oriented regions of the objects onto one another, thus giving the user some control over how corresponding parts of the objects map to one another.

### 3.9.5 Recursive Subdivision

The main problem with the procedure above is that many new edges are created as a result of the merging operation. There is no attempt to map existing edges into one another. To avoid a plethora of new edges, a recursive approach can be taken in which each object is reduced to two-dimensional polygonal meshes [31]. Meshes from each object are matched by associating the boundary vertices and adding new ones when necessary. The meshes are then similarly split and the procedure is recursively applied until everything has been reduced to triangles. During the splitting process, existing edges are used whenever possible to reduce the number of new edges created. Edges and faces are added during subdivision to maintain topological equivalence. A data structure must be used that supports a closed, oriented, path of edges along the surface of an object. Each mesh is defined by being on a particular side (e.g., right side) of such a path, and each section of a path will be shared by two and only two meshes.

The initial objects are divided into an initial number of polygonal meshes. Each mesh is associated with a mesh from the other object so that adjacency relationships are maintained by the mapping. The simplest way to do this is merely to break each object into two meshes—a front mesh and a back mesh. A front and back mesh can be constructed by searching for the shortest paths between the topmost, bottomost, leftmost, and rightmost vertices of the object and then appending these paths (Figure 3.97). On particularly simple objects, care must be taken so that these paths do not touch except at the extreme points.

This is the only place where geometric information is used. If the user wants certain areas of the objects to map to each other during the transformation process, then those areas should be the initial meshes associated with each other, providing the adjacency relationships are maintained by the associations.

When a mesh is associated with another mesh, a one-to-one mapping must be established between the vertices on the boundary of the two meshes. If one of the

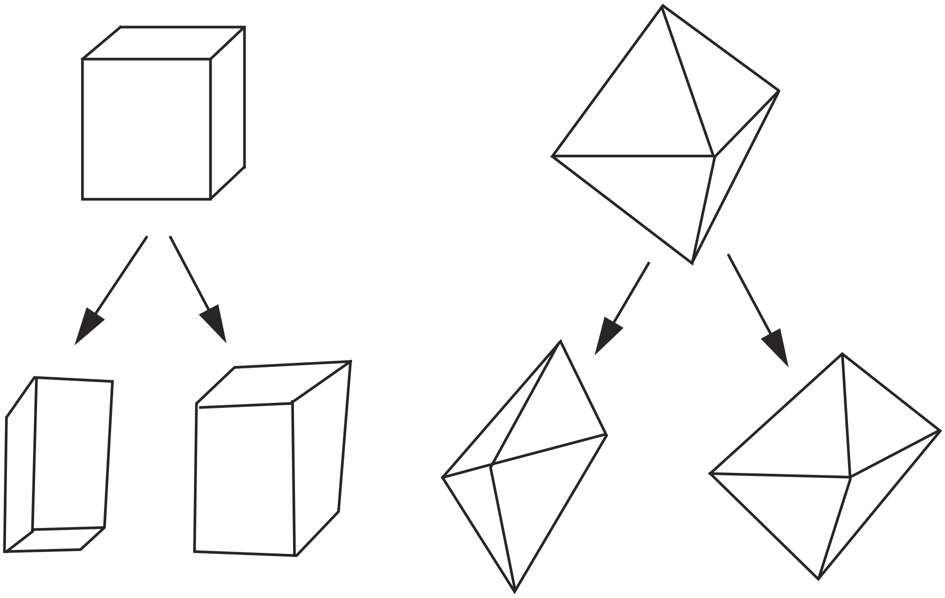
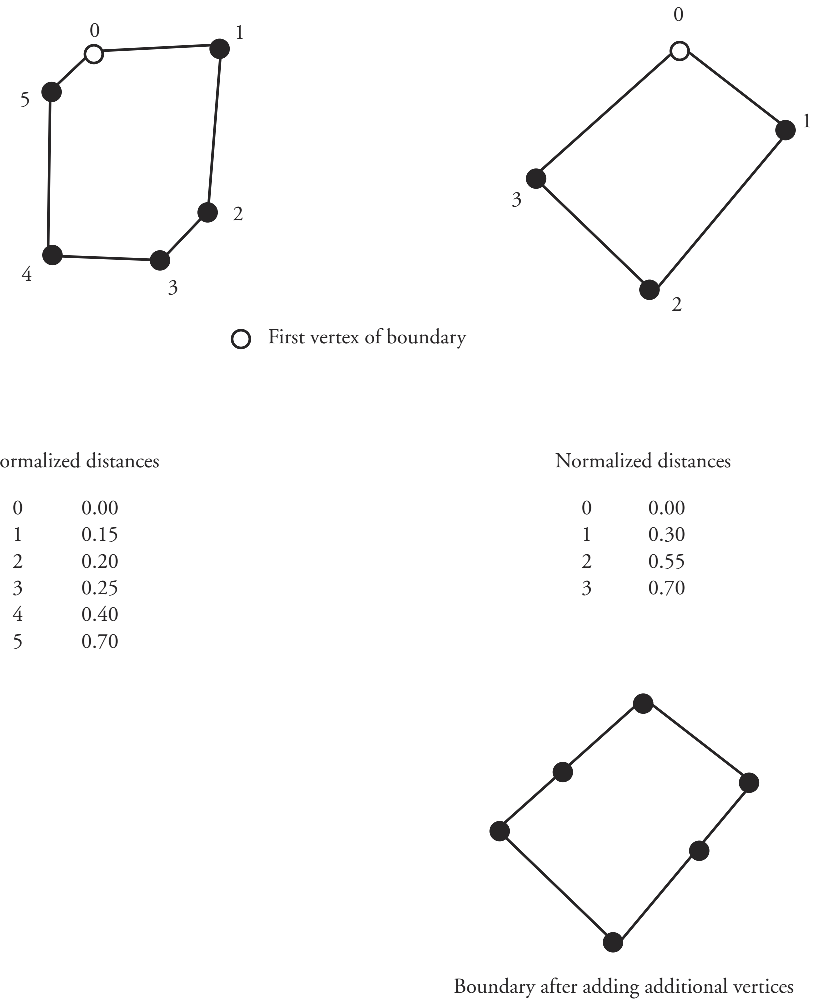


**Figure 3.97** Splitting objects into initial front and back meshes

meshes has fewer boundary vertices than the other, then new vertices must be introduced along its boundary to make up for the difference. There are various ways to do this, and the success of the algorithm is not dependent on the method. A suggested method is to compute the normalized distance of each vertex along the boundary as measured from the first vertex of the boundary (the topmost vertex can be used as the first vertex of the boundaries of the initial meshes). For the boundary with fewer vertices, new vertices can be added one at a time by searching for the largest gap in normalized distances for successive vertices in the boundary (Figure 3.98). These vertices must be added to the original object definition. When the boundaries have the same number of vertices, a vertex on one boundary



Normalized distances

| | |
|---|---|
| 0 | 0.00 |
| 1 | 0.15 |
| 2 | 0.20 |
| 3 | 0.25 |
| 4 | 0.40 |
| 5 | 0.70 |

Normalized distances

| | |
|---|---|
| 0 | 0.00 |
| 1 | 0.30 |
| 2 | 0.55 |
| 3 | 0.70 |

Boundary after adding additional vertices

**Figure 3.98** Associating vertices of boundaries

is said to be associated with the vertex on the other boundary at the same relative location.

Once the meshes have been associated, each mesh is recursively divided. One mesh is chosen for division, and a path of edges is found across it. Again, there are various ways to do this and the procedure is not dependent on the method chosen for its success. However, the results will have a slightly different quality depending on the method used. One good approach is to choose two vertices across the boundary from each other and try to find an existing path of edges between them. An iterative procedure can be easily implemented that tries all pairs halfway around and then tries all pairs one less than halfway around, then two less, and so on. There will be no path only if the "mesh" is a single triangle—in which case the other mesh must be tried. There will be some path that exists on one of the two meshes unless both meshes are a single triangle, which is the terminating criterion for the recursion (and would have been previously tested for).

Once a path has been found across one mesh, then a path across the mesh it is associated with must be established between corresponding vertices. This may require creating new vertices and new edges (and, therefore, new faces) and is the trickiest part of the implementation because minimizing the number of new edges will help reduce the complexity of the resulting topologies. When these paths (one on each mesh) have been created, the meshes can be divided along these paths, creating two pairs of new meshes. The boundary association, finding a path of edges, and mesh splitting are recursively applied to each new mesh until all of the meshes have been reduced to single triangles. At this point the new vertices and new edges have been added to one or both objects so that both objects have the same topology. Vertex-to-vertex interpolation of vertices can take place at this point in order to carry out the object interpolation.

### 3.9.6 Summary

3D shape interpolation remains a difficult problem to implement robustly and efficiently. However, several useful techniques have been developed that provide promising results. Two of these are presented above and should give some idea of the possibilities.

## 3.10 Chapter Summary

Interpolation is fundamental to most animation, and the ability to understand and control the interpolation process is very important in computer animation programming. Interpolation of values takes many forms, including arc length, image pixel color, and shape parameters. The control of the interpolation process can be key framed, scripted, or analytically determined. But in any case, interpolation forms the foundation upon which most computer animation takes place, including those animations that use advanced algorithms.

## References

1. Alias/Wavefront, *MEL for Artists,* Alias/Wavefront, a Silicon Graphics Company, 1998.
2. A. Barr, "Global and Local Deformations of Solid Primitives," *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 21–30 (July 1984, Minneapolis, Minn.).
3. T. Beier and S. Neely, "Feature Based Image Metamorphosis," *Computer Graphics* (Proceedings of SIGGRAPH 92), 26 (2), pp. 253–254 (July 1992, Chicago, Ill.). Edited by Edwin E. Catmull. ISBN 0-201-51585-7.
4. R. Burden, J. Faire, J. Douglas, and A. Reynolds, *Numerical Analysis,* Prindle, Weber & Schmidt, Boston, Mass., 1981.
5. N. Burtnyk and M. Wein, "Computer Generated Key Frame Animation," *Journal of the Society of Motion Picture and Television Engineers,* 8 (3), 1971, pp. 149–153.
6. N. Burtnyk and M. Wein, "Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation," *Communications of the ACM,* 19 (10), October 1976, pp. 564–569.
7. E. Catmull, "The Problems of Computer-Assisted Animation," *Computer Graphics* (Proceedings of SIGGRAPH 78), 12 (3), pp. 348–353 (August 1978, Atlanta, Ga.).
8. J. Chadwick, *Bragger,* Ohio State University, Columbus, 1989.
9. J. Chadwick, *City Savvy,* Ohio State University, Columbus, 1988.
10. J. Chadwick, D. Haumann, and R. Parent, "Layered Construction for Deformable Animated Characters," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 243–252, (August 1989, Boston, Mass.).
11. E. Chen and R. Parent, "Shape Averaging and Its Application to Industrial Design," *IEEE Computer Graphics and Applications,* 9 (1), January 1989, pp. 47–54.
12. H. N. Christiansen and T. W. Sederberg, "Conversion of Complex Contour Line Definitions into Polygonal Element Mosaics," *Computer Graphics* (Proceedings of SIGGRAPH 78), 12 (3), pp. 187–192 (August 1978, Atlanta, Ga.).
13. S. Coquillart, "Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 187–196 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
14. S. Coquillart and P. Jancene, "Animated Free-Form Deformation: An Interactive Animation Technique," *Computer Graphics* (Proceedings of SIGGRAPH 91), 25 (4), pp. 41–50 (July 1991, Las Vegas, Nev.). Edited by Thomas W. Sederberg. ISBN 0-201-56291-X.
15. T. Defiant, "The Digital Component of the Circle Graphics Habitat," *Proceedings of the National Computer Conference '76,* pp. 195–203 (June 7–10, 1976, New York), AFIPS Press.
16. S. Ganapathy and T. G. Denny, "A New General Triangulation Method for Planar Contours," *Computer Graphics* (Proceedings of SIGGRAPH 82), 16 (3), pp. 69–75 (July 1982, Boston, Mass.).

17. B. Guenter and R. Parent, "Computing the Arc Length of Parametric Curves," *IEEE Computer Graphics and Applications,* 10 (3), May 1990, pp. 72–78.
18. R. Hackathorn, "ANIMA II: A 3D Color Animation System," *Computer Graphics* (Proceedings of SIGGRAPH 77), 11 (2), pp. 54–64 (July 1977, San Jose, Calif.). Edited by James George.
19. R. Hackathorn, R. Parent, B. Marshall, and M. Howard, "An Interactive Micro-Computer Based 3D Animation System," *Proceedings of Canadian Man-Computer Communications Society Conference '81,* pp. 181–191 (June 10–12, 1981, Waterloo, Ontario).
20. E. Hofer, "A Sculpting Based Solution for Three-Dimensional Computer Character Facial Animation," Master's thesis, Ohio State University, 1993.
21. C. Judd and P. Hartley, "Parametrization and Shape of B-Spline Curves for CAD," *Computer Aided Design* 12 (5), September 1980, pp. 235–238.
22. J. Kent, W. Carlson, and R. Parent, "Shape Transformation for Polyhedral Objects," *Computer Graphics* (Proceedings of SIGGRAPH 92), 26 (2), pp. 47–54 (July 1992, Chicago, Ill.). Edited by Edwin E. Catmull. ISBN 0-201-51585-7.
23. M. Levoy, "A Color Animation System Based on the Mulitplane Technique," *Computer Graphics* (Proceedings of SIGGRAPH 77), 11 (2), pp. 65–71 (July 1977, San Jose, Calif.). Edited by James George.
24. P. Litwinowicz, "Inkwell: A 2½-D Animation System," *Computer Graphics* (Proceedings of SIGGRAPH 91), 25 (4), pp. 113–122 (July 1991, Las Vegas, Nev.). Edited by Thomas W. Sederberg.
25. P. Litwinowicz and L. Williams, "Animating Images with Drawings," Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, pp. 409–412 (July 1994, Orlando, Fla.), ACM Press. Edited by Andrew Glassner. ISBN 0-89791-667-0.
26. N. Magnenat-Thalmann and D. Thalmann, *Computer Animation: Theory and Practice,* Springer-Verlag, New York, 1985.
27. N. Magnenat-Thalmann, D. Thalmann, and M. Fortin, "Miranim: An Extensible Director-Oriented System for the Animation of Realistic Images," *IEEE Computer Graphics and Applications,* 5 (3), pp. 61–73, March 1985.
28. S. May, "Encapsulated Models: Procedural Representations for Computer Animation," Ph.D. dissertation, Ohio State University, 1998.
29. L. Mezei and A. Zivian, "ARTA: An Interactive Animation System," *Proceedings of Information Processing 71,* pp. 429–434 (North-Holland, Amsterdam, 1971).
30. M. Mortenson, *Geometric Modeling,* John Wiley & Sons, New York, 1985.
31. R. Parent, "Shape Transformation by Boundary Representation Interpolation: A Recursive Approach to Establishing Face Correspondences," OSU Technical Report OSU-CISRC-2-91-TR7, *Journal of Visualization and Computer Animation* 3, January 1992, pp. 219–239.
32. R. Parent, "A System for Generating Three-Dimensional Data for Computer Graphics," Ph.D. dissertation, Ohio State University, 1977.
33. W. Reeves, "In-betweening for Computer Animation Utilizing Moving Point Constraints," *Computer Graphics* (Proceedings of SIGGRAPH 81), 15 (3), pp. 263–270 (August 1981, Dallas, Tex.).
34. C. Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics* (Proceedings of SIGGRAPH 82), 16 (3), pp. 289–296 (July 1982, Boston, Mass.).
35. D. Rogers and J. Adams, *Mathematical Elements for Computer Graphics,* McGraw-Hill, New York, 1976.

36. T. Sederberg, “Free-Form Deformation of Solid Geometric Models,” *Computer Graphics* (Proceedings of SIGGRAPH 86), 20 (4), pp. 151–160 (August 1986, Dallas, Tex.). Edited by David C. Evans and Russell J. Athay.
37. K. Shoemake, “Animating Rotation with Quaternion Curves,” *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 143–152 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
38. Side Effects Software, Toronto, *Houdini 2.0: User Guide,* September 1997.
39. P. Talbot, J. Carr III, R. Coulter, Jr., and R. Hwang, “Animator: An On-line Two-Dimensional Film Animation System,” *Communications of the ACM,* 14 (4), pp. 251–259.
40. K. Weiler. “Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments,” *IEEE Computer Graphics and Applications,* 5 (1), pp. 21–40, January 1985.
41. G. Wolberg, *Digital Image Warping,* IEEE Computer Society Press, Los Alamitos, Calif., 1988.

CHAPTER 4

# Advanced Algorithms

Animators are often more concerned with the general quality of the motion than with precisely controlling the position and orientation of each object in each frame. Such is the case with physical simulations; when dealing with a large number of objects; when animating objects whose motion is constrained in one way or another; or when dealing with objects in the background whose precise motion is not of great importance to the animation. This chapter is concerned with the algorithms that employ some kind of structured model approach to producing motion. The structure of the model automatically enforces certain qualities or constraints on the motion to be generated. The use of a model eliminates the need for the animator to be constantly concerned with specifying details of the motion. Instead, those details are filled in by the model. Of course, by using these models, the animator typically loses some fine control over the motion of the objects. The model can take various forms, such as enforcing relative placement of geometric elements, enforcing nonpenetration constraints, calculating reaction to gravity and other forces, enforcing volume preservation, or following rules of behavior. Motion is produced by the combination of the constraints and rules of the model with additional control information from the user.

This chapter discusses both kinematic and dynamic models. *Kinematic control* refers to the movement of objects irrespective of the forces involved in producing the movement. For example, the interpolation techniques covered in the previous chapter are concerned with kinematic control. Several of the algorithms given here are also kinematic in nature, such as those having to do with the control of linked armatures. *Dynamic control* is concerned with computing the underlying forces that are then used to produce movement. Among the dynamic control algorithms are those that are physically based.

In the discussion that follows, kinematic models are covered first, then hierarchical models and the associated use of forward and inverse kinematic control. Rigid body dynamics and the use of constraints, which are primarily concerned with dynamic control, are then discussed, followed by techniques to control groups of objects. The chapter concludes with a discussion of animating implicit surfaces.

## 4.1 Automatic Camera Control

One simple example of using a procedure or algorithm for control is the specification of the position and orientation of the camera. Various guidelines are employed in the art of filmmaking for positioning the camera so as to capture conversation, follow action, and emphasize spatial qualities such as vastness or intimacy [6]. These guidelines primarily address aesthetic concerns; the discussion here focuses on basic computational models for calculating camera motion based on the geometric configuration of the elements in the scene.

Often, the animator is more concerned with effectively showing an action taking place than with getting a particular camera angle. When using high-level algorithms to control the motion of objects, the animator may not be able to anticipate the exact, or even general, position of objects during the animation sequence. As a consequence, it is difficult for the animator to know exactly how to position and orient the camera so that the important action will be captured in the image. In such cases, it is often useful to have the camera position and center-of-interest location automatically generated for each frame of the animation. There are several ways to automatically set up camera control; the choice depends on the effect desired in the animation.

A common way to automatically control the camera is to place the camera position and/or center of interest relative to the positions of one or more objects in the animation. (A simplifying assumption, which will be used for now, is that the camera will maintain a head-up orientation during the motion.) A static camera can be used to track an object by attaching the center of interest to the object's center point. If a group of objects move together, then the average of their locations can be used as the camera's center of interest. This works as long as other moving objects do not get in the way and the object (or group of objects) of interest does not move too far away.

The camera can closely follow a widely roaming object by locating the position of the camera relative to the moving object. For example, a constant global offset vector can be used to position the camera relative to the center of interest; the offset vector can also be relative to the tracked object's local coordinate system. Sometimes it is useful to constrain the camera's position to a predefined plane or along a line segment or curve. The closest point on the constraining element to the center of interest can be calculated and used as the camera location. For example, in some situations it might make sense to keep the camera at a specified altitude (constrained to be located on a plane parallel with the ground plane) so that it can capture the action below. Other constraints can also be used, such as distance between the camera position and the center of interest and/or the angle made by the view vector with the ground plane.

Such precise calculations of a camera location can sometimes result in movements that are too jerky if the objects of interest change position too rapidly. If the camera motion can be precomputed (the animation is not being generated interactively or in real time), then smoothing the curve by averaging each point with some number of adjacent points will smooth out the curve. Attaching the camera or center of interest with a spring and damper, instead of rigidly, can help to smooth out the motion. For example, to track a flock of birds, the center of interest can be attached to the center of the flock and the camera can be attached by a spring-damper combination (see Section 4.3 for a discussion of modeling dynamics and Appendix B for the basic equations of motion) to a location that is to the back and side of this position.

It is useful during animation development to define cameras whose only purpose is to inspect the motion of other objects. A camera can be attached to the front or "over the shoulder" of an object moving in an environment, or it can be attached to a point directly above an object of interest with the center of interest coinciding with the object. To check facial expressions and eye movements, a camera can be positioned directly in front of an object with the center of interest positioned on the figure's face.

Although automatic control of the camera is a useful tool, as with most automated techniques, the animator trades off control for ease of use. Efficient use of automatic control requires experience so that the quality of the results can be anticipated beforehand.

## 4.2 Hierarchical Kinematic Modeling

*Hierarchical modeling* is the enforcement of connectivity (or relative placement) constraints among objects organized in a treelike structure. Planetary systems are one type of hierarchical model. In planetary systems, moons rotate around planets, which rotate around a sun, which moves in a galaxy. A common type of hierarchical model used in graphics has objects that are connected end to end to form multibody jointed chains. Such hierarchies are useful for modeling animals and humans so that the joints of the limbs are manipulated to produce a figure with moving appendages. Such a figure is often referred to as *articulated*. The movement of an appendage by changing the configuration of a joint is referred to as *articulation*. Because the connectivity of the figure is built into the structure of the model, the animator does not need to make sure that the objects making up the limbs stay attached to one another.

Much of the material concerning the animation of hierarchies in computer graphics comes directly from the field of robotics (e.g., [7]). The robotics literature discusses the modeling of *manipulators,* a sequence of objects connected in a chain by *joints*. The rigid objects forming the connections between the joints are called *links,* and the free end of the chain of alternating joints and links is called the *end effector*. The local coordinate system associated with each joint is referred to as the *frame*.

Robotics is concerned with all types of joints in which two links move relative to one another. Graphics, on the other hand, is concerned primarily with *revolute* joints, in which one link rotates about a fixed point of the other link. The links are usually considered to be pinned together at this point, and the link farther down the chain rotates while the other one remains fixed—at least as far as this joint is concerned. The other type of joint used in computer animation is the *prismatic* joint, in which one link translates relative to another. See Figure 4.1.

The joints of Figure 4.1 allow motion in one direction and are said to have one *degree of freedom* (DOF). Structures in which more than one degree of freedom are coincident are called *complex joints*. Complex joints include the planar joint and the ball-and-socket joint. Planar joints are those in which one link slides on the

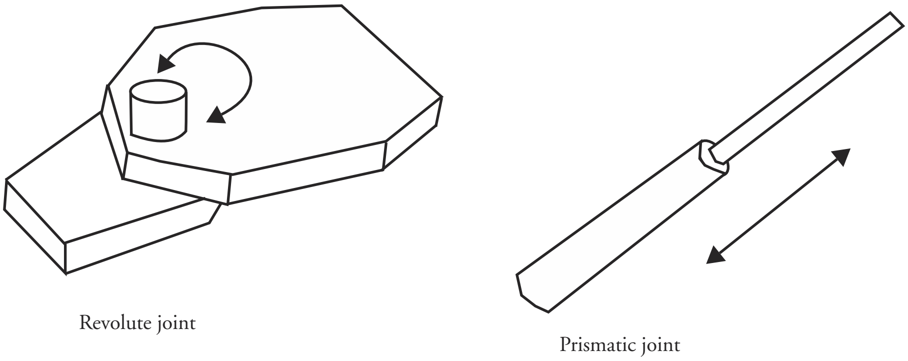
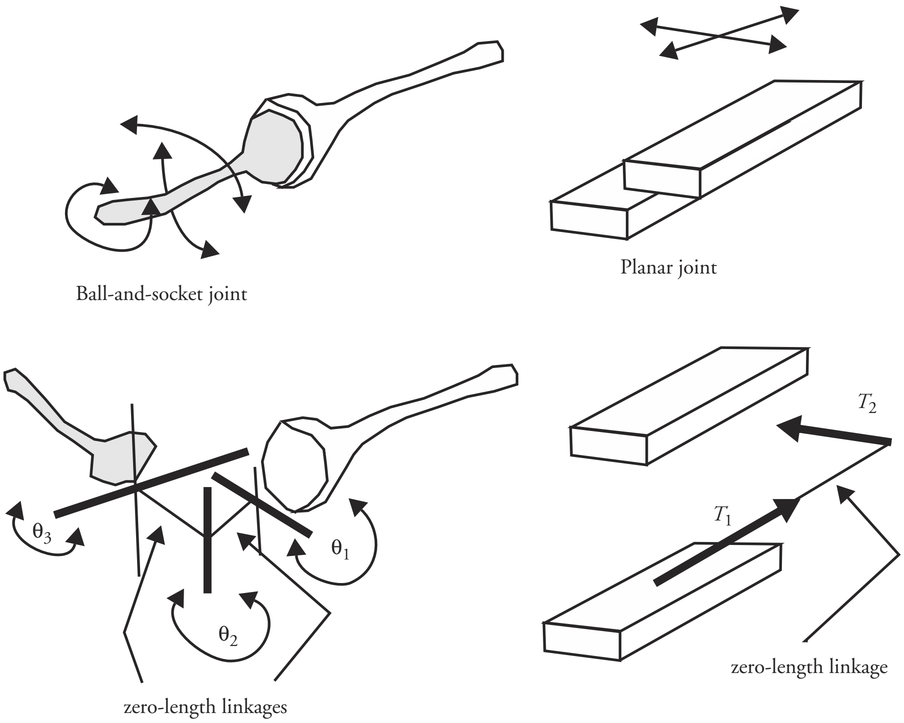


**Figure 4.1** Typical joints used in computer animation



Ball-and-socket joint modeled as 3 one-degree joints with zero-length links

Planar joint modeled as 2 one-degree prismatic joints with zero-length links

**Figure 4.2** Modeling complex joints

planar surface of another. Typically, when a joint has more than one ($n > 1$) degree of freedom, such as a ball-and-socket joint, it is modeled as a set of $n$ one-degree-of-freedom joints connected by $n - 1$ links of zero length (see Figure 4.2).

### 4.2.1 Representing Hierarchical Models

Human figures and animals are conveniently modeled as hierarchical linkages. Such linkages can be represented by a tree structure of *nodes* connected by *arcs.*[1] The highest node of the tree is the *root node,* which corresponds to the root object of the hierarchy whose position is known in the global coordinate system. The

[1] The connections between nodes of a tree structure are sometimes referred to as links; however, the robotics literature refers to the objects between the joints as links. To avoid overloading the term *links, arcs* is used here to refer to the connections between nodes in a tree.

position of all other nodes of the hierarchy will be located relative to the root node. A node from which no arcs extend downward is referred to as a *leaf node*. “Higher up in the hierarchy” refers to a node that is closer to the root node. When discussing two nodes of the tree connected by an arc, the one higher up the hierarchy is referred to as the *parent node,* and the one farther down the hierarchy is referred to as the *child node*.

The mapping between the hierarchy and tree structure relates a node of the tree to information about the object part (the link) and relates an arc of the tree (the joint) to the transformation to apply to all of the nodes below it in the hierarchy. Relating a tree arc to a figure joint may seem counterintuitive, but it is convenient because a node of the tree can have several arcs emanating from it, just as an object part may have several joints attached to it. In a discussion of a hierarchical model presented by a specific tree structure, the terms *node, object part,* and *link* are used interchangeably since all refer to the geometry to be articulated. Similarly, the terms *joint* and *arc* are used interchangeably.

In the tree structure, there is a root arc that represents a global transformation to apply to the root node (and, therefore, indirectly to all of the nodes of the tree). Changing this transformation will rigidly reposition the entire structure in the global coordinate system. See Figure 4.3.

A node of the tree structure contains the information necessary to define the object part in a position ready to be articulated. In the case of rotational joints, this means that the point of rotation on the object part is made to coincide with

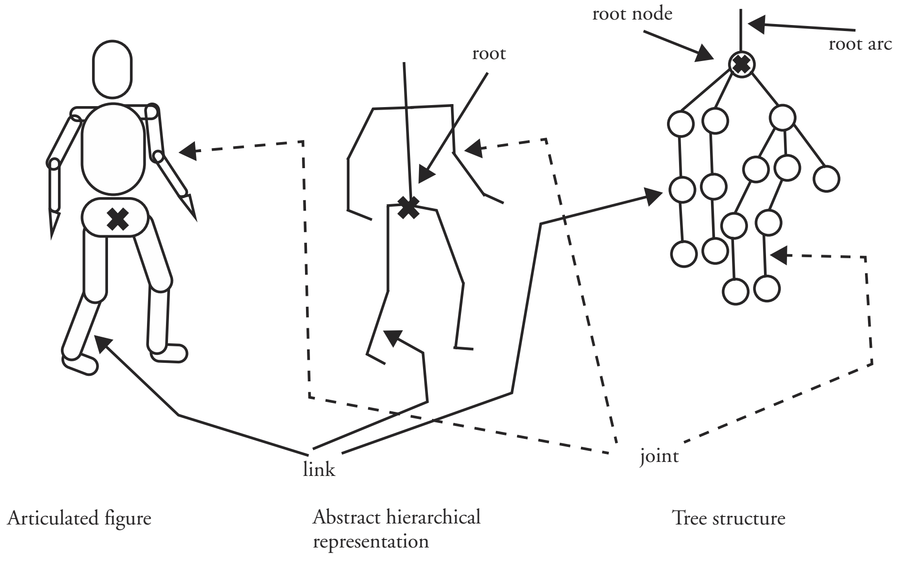


**Figure 4.3** Example of a tree structure representing a hierarchical structure

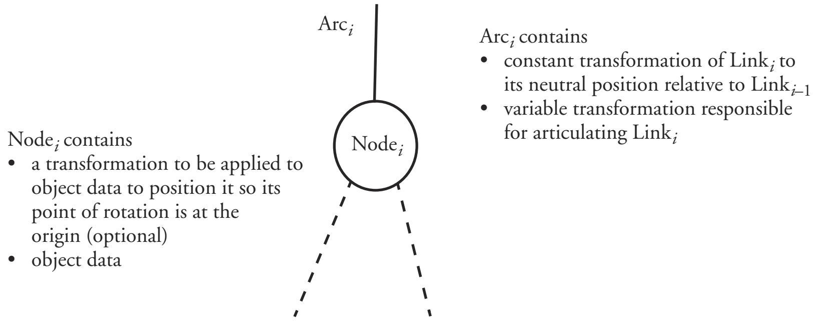

Arc$_i$
Arc$_i$ contains
• constant transformation of Link$_i$ to its neutral position relative to Link$_{i–1}$
• variable transformation responsible for articulating Link$_i$
Node$_i$
Node$_i$ contains
• a transformation to be applied to object data to position it so its point of rotation is at the origin (optional)
• object data

**Figure 4.4** Arc and node definition

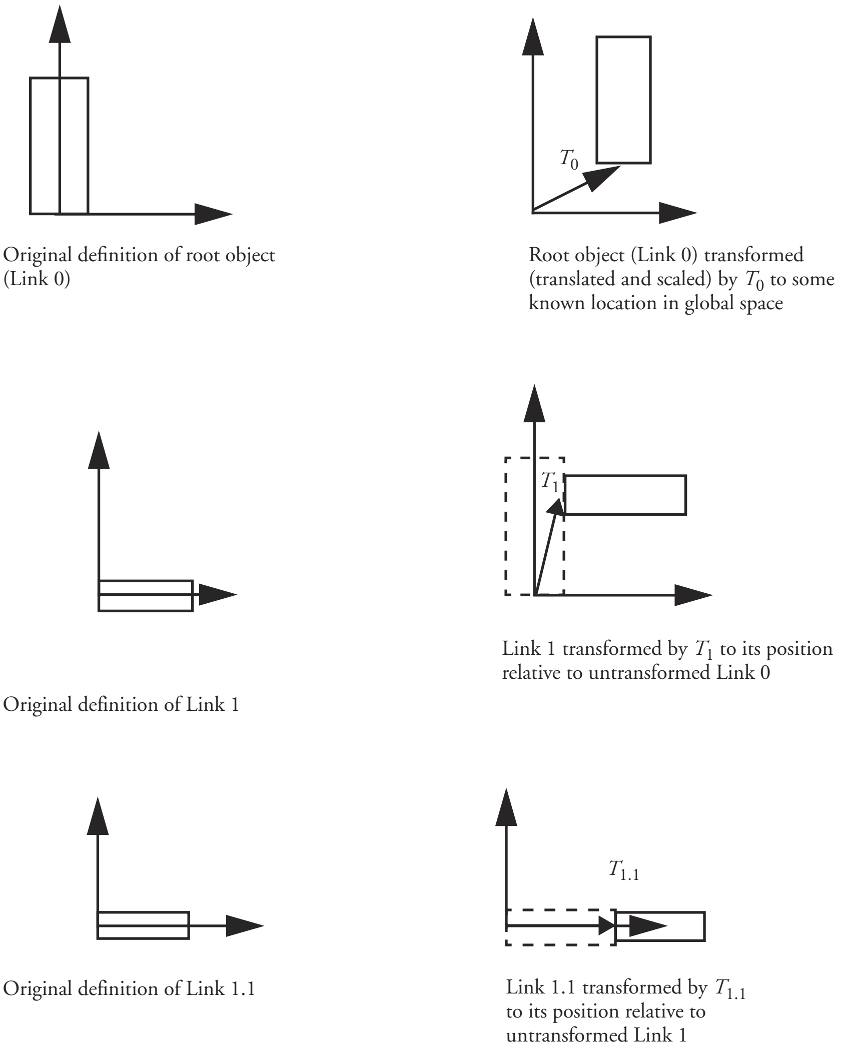
the origin. The object data may be defined in such a position, or there may be a transformation matrix contained in the node that, when applied to the object data, positions it so. In either case, all of the information necessary to prepare the object data for articulation is contained at the node. The node represents the transformation of the object data into a link of the hierarchical model.

Two types of transformations are associated with an arc leading to a node. One transformation rotates and translates the object into its position of attachment relative to the link one position up in the hierarchy. This defines the link's neutral position relative to its parent. The other transformation is the variable information responsible for the actual joint articulation. See Figure 4.4.

### A Simple Example

Consider the simple, two-dimensional, three-link example of Figure 4.5. In this example, there is assumed to be no transformation at any of the nodes; the data are defined in a position ready for articulation. Link 0, the root object, is transformed to its orientation and position in global space by $T_0$. Because all of the other parts of the hierarchy will be defined relative to this part, this transformation affects the entire assemblage of parts and thus will transform the position and orientation of the entire structure. This transformation can be changed over time in order to animate the position and orientation of the rigid structure. Link 1 is defined relative to the untransformed root object by transformation $T_1$. Similarly, Link 1.1 is defined relative to the untransformed Link 1 by transformation $T_{1.1}$. These relationships can be represented in a tree structure by associating the links with nodes and the transformations with arcs. In the example shown in Figure 4.6, the articulation transformations are not yet included in the model.

An arc in the tree representation contains a transformation that applies to the object represented by the node to which the arc immediately connects. This transformation is also applied to the rest of the linkage farther down the hierarchy. The



Original definition of root object (Link 0)

Root object (Link 0) transformed (translated and scaled) by $T_0$ to some known location in global space

Original definition of Link 1

Link 1 transformed by $T_1$ to its position relative to untransformed Link 0

Original definition of Link 1.1

Link 1.1 transformed by $T_{1.1}$ to its position relative to untransformed Link 1

**Figure 4.5** Example of a hierarchical model

vertices of a particular object can be transformed to their final positions by concatenating the transformations higher up the tree and applying the composite transformation matrix to the vertices. A vertex of the root object, Link 0, is located in the world coordinate system by applying the rigid transformation that affects the entire structure; see Equation 4.1. A vertex of the Link 1 object is located in the world coordinate system by transforming it first to its location relative to Link 0 and then relocating it (conceptually along with Link 0) to world space by Equation 4.2. A vertex of the Link 1.1 object is similarly located in world space by

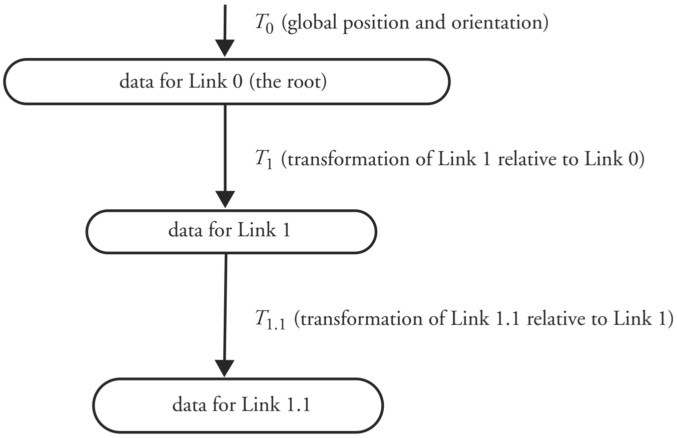

$T_0$ (global position and orientation)
data for Link 0 (the root)
$T_1$ (transformation of Link 1 relative to Link 0)
data for Link 1
$T_{1.1}$ (transformation of Link 1.1 relative to Link 1)
data for Link 1.1

**Figure 4.6** Example of a tree structure

Equation 4.3. Notice that as the tree is traversed farther down one of its branches, a newly encountered arc transformation is concatenated with the transformations previously encountered.

$$V_0' = T_0 \cdot V_0 \tag{Eq. 4.1}$$

$$V_1' = T_0 \cdot T_1 \cdot V_1 \tag{Eq. 4.2}$$

$$V_{1.1}' = T_0 \cdot T_1 \cdot T_{1.1} \cdot V_{1.1} \tag{Eq. 4.3}$$

As previously discussed, when one constructs the static position of the assembly, each arc of the tree has an associated transformation that rotates and translates the link associated with the child node relative to the link associated with the parent node. To easily animate a revolute joint, there is also a parameterized (variable) transformation that controls the rotation at the specified joint. See Figure 4.7. In the tree representation that implements a revolute joint, a rotation transformation is associated with the arc that precedes the node representing the link to be rotated. See Figure 4.8. The rotational transformation is applied to the link before the arc's constant transformation. If a transformation is present at the node (for preparing the data for articulation), then the rotational transformation is applied after the node transformation but before the arc's constant transformation.

To locate a vertex of Link 1 in world space, one must first transform it via the joint rotation matrix. Once that is complete, then the rest of the transformations up the hierarchy are applied. See Equation 4.4. A vertex of Link 1.1 is transformed similarly by compositing all of the transformations up the hierarchy to the root, as in Equation 4.5. In the case of multiple appendages, the tree structure would

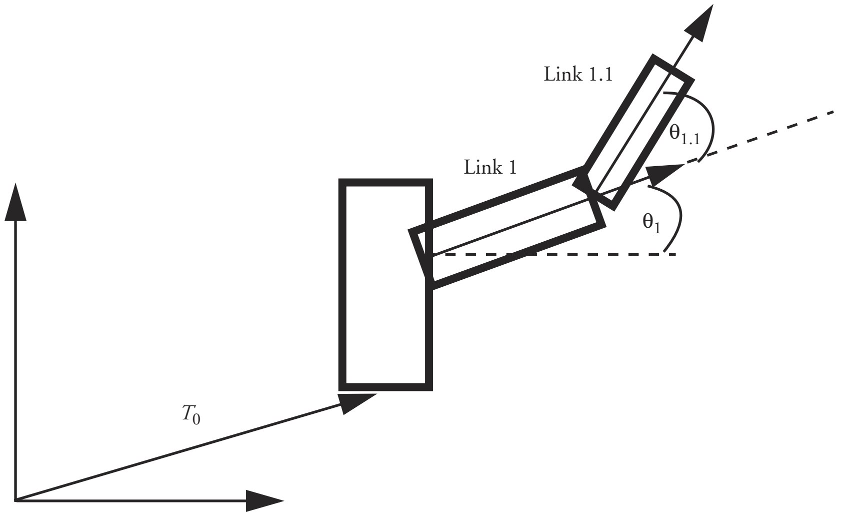


**Figure 4.7** Variable rotations at the joints

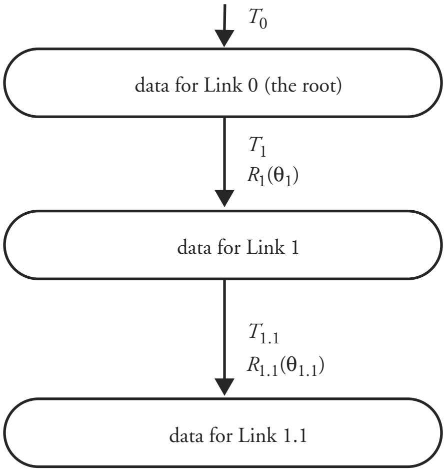


**Figure 4.8** Hierarchy showing joint rotations

reflect the bifurcations (or multiple branches if more than two). Adding another arm to our simple example results in Figure 4.9.

$$V_1' = T_0 \cdot T_1 \cdot R_1(\theta_1) \cdot V_1 \quad \text{(Eq. 4.4)}$$

$$V_{1.1}' = T_0 \cdot T_1 \cdot R_1(\theta_1) \cdot T_{1.1} \cdot R_{1.1}(\theta_{1.1}) \cdot V_{1.1} \quad \text{(Eq. 4.5)}$$

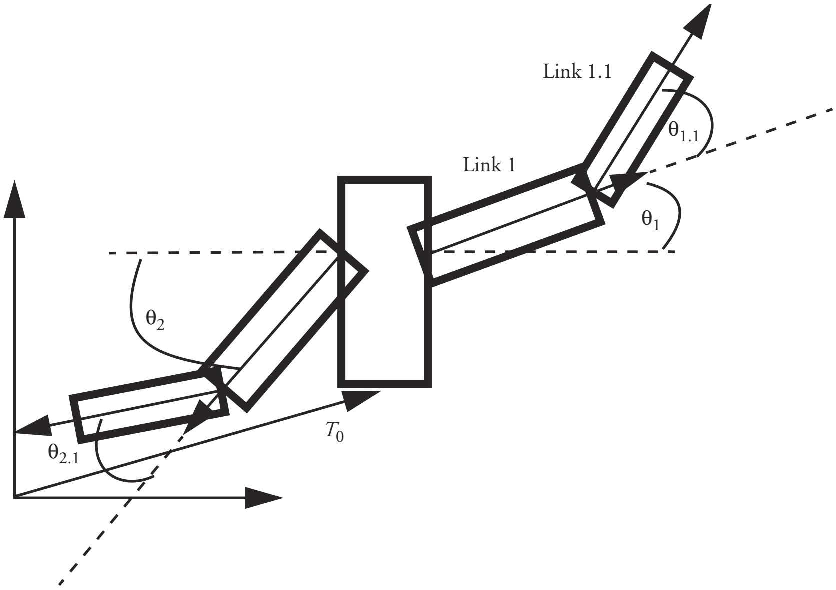


**Figure 4.9** Hierarchy with two appendages

The corresponding tree structure would have two arcs emanating from the root node, as in Figure 4.10. Branching in the tree occurs whenever multiple appendages emanate from the same object. For example, in a simplified human figure, the root hip area (see Figure 4.3) might branch into the torso and two legs. If prismatic joints are used, the strategy is the same, the only difference being that the rotation transformation of the joint (arc) is replaced by a translation.

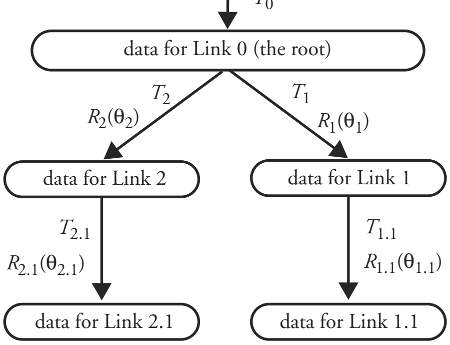


**Figure 4.10** Tree structure corresponding to hierarchy with two appendages

### 4.2.2 Forward Kinematics

Evaluation of a hierarchy by traversing the corresponding tree produces the figure in a position that reflects the setting of the joint parameters. Traversal follows a depth-first pattern from root to leaf node. The traversal then backtracks up the tree until an unexplored downward arc is encountered. The downward arc is then traversed, followed by backtracking up to find the next unexplored arc. This traversal continues until all nodes and arcs have been visited. Whenever an arc is followed down the tree hierarchy, its transformations are concatenated to the transformations of its parent node. Whenever an arc is traversed back up the tree to a node, the transformation of that node must be restored before traversal continues downward.

A stack of transformations is a conceptually simple way to implement the saving and restoring of transformations as arcs are followed down and then back up the tree. Immediately before an arc is traversed downward, the current composite matrix is pushed onto the stack. The arc transformation is then concatenated with the current transformation by premultiplying it with the composite matrix. Whenever an arc is traversed upward, the top of the stack is popped off of the stack and becomes the current composite transformation. (If node transformations, which prepare the data for transformation, are present, they must not be included in a matrix that gets pushed onto the stack.)

```
/* the node structure */
typedef struct node_struct {
    object       *obj;         /*  pointer to object data structure */
    struct arc   **arc_array;  /* array of pointer to arcs emanating
                                  downward from node */
    int          num_arc;      /* number of arcs in array */
} node_td;

/* the arc structure */
typedef struct arc_struct {
    trans_mat    rot;          /* joint rotation matrix */
    trans_mat    m;            /* orientation and position matrix */
    node_td      *nptr;        /* pointer to node below arc */
} arc_td;

/* the highest structure of the tree is the root arc holding the global
   transforms */
/* the high-level routine simply calls for the (recursive) evaluation of
   the root node */
eval_tree(struct arc rootArc)
{
    eval_node(rootArc->m,rootArc->node);  /* recursively evaluate the
                                             root node */
}

/* the recursive evaluation routine */
eval_node(trans_mat m,node_struct node);
{
    trans_mat    temp_m;              /* temporary transformation */

    concat_tm(node->m,m,&temp_m);     /* concatenate current and node
                                         transform */
    transf_obj(obj,temp_m,&temp_obj); /* transform object */
    display_obj(temp_obj);            /* display transformed object */

    /* loop over each arc emanating from node and recursively evaluate
    /* the attached node */
    for (l=0; l<node->num_arc; l++) {
        premul_tm(node->arc_array[l]->m,temp_m);
        premul_tm(node->arc_array[l]->rot,temp_m);
        eval_node(temp_m,node->arc_array[l]->node);
    }
}
```

To animate the linkage, the rotation parameters at the joints (the changeable rotation matrices associated with the tree arcs and parameterized by joint angle) are manipulated. A completely specified set of rotation parameters, which results in positioning the hierarchical figure, is called a *pose*. A pose is specified by a vector (the *pose vector*) consisting of one angle for each joint.

In a simple animation, a user may determine a key position interactively then interpolate joint rotations between key positions. Positioning a figure by specifying all of the joint angles is called *forward kinematics*. Unfortunately, getting the figure to a final desired position by specifying joint angles can be tedious for the user. Often, it is a trial-and-error process. To avoid the difficulties in having to specify all of the joint angles, *inverse kinematics* (IK) is sometimes used, in which the desired position and orientation of the end effector are given and the internal joint angles are calculated automatically.

### 4.2.3 Local Coordinate Frames

In setting up complex hierarchies and in applying sophisticated procedures such as inverse kinematics, it is convenient to be able to define points in the local coordinate system (frame) associated with a joint and to have a well-defined method for converting the coordinates of a point from one frame to another. A common use for this method is to convert points defined in the frame of a joint to the global coordinate system for display purposes. In the example above, a transformation matrix is associated with each arc to represent the transformation of a point from the local coordinate space of a child node to the local coordinate space of the parent node. The inverse of the transformation matrix can be used to transform a point from the parent's frame to the child's frame. Successively applying the inverses of matrices farther up the hierarchy can transform a point from any position in the tree into world coordinates for display. In the three-dimensional case, 4x4 transformation matrices can be used to describe the relation of one coordinate frame to the next. However, robotics has adopted a more concise and more meaningful parameterization: the Denavit-Hartenberg notation.

### Denavit-Hartenberg Notation

The Denavit-Hartenberg (DH) notation is a particular way of describing the relationship of a parent coordinate frame to a child coordinate frame. This convention is commonly used in robotics and often adopted for use in computer animation. Each frame is described relative to an adjacent frame by four parameters that describe the position and orientation of a child frame in relation to its parent's frame.

For revolute joints, the *z*-axis of the joint's frame corresponds to the axis of rotation (prismatic joints are discussed below). The link associated with the joint extends down the *x*-axis of the frame. First consider a simple configuration in which the joints and the axes of rotation are coplanar. The distance down the *x*-axis from one joint to the next is the *link length,* $a_i$. The *joint angle*, $\theta_{i+1}$, is specified by the rotation of the $i + 1$ joint's *x*-axis, $x_{i+1}$, about its *z*-axis relative to the *i*th frame's *x*-axis direction, $x_i$. See Figure 4.11.

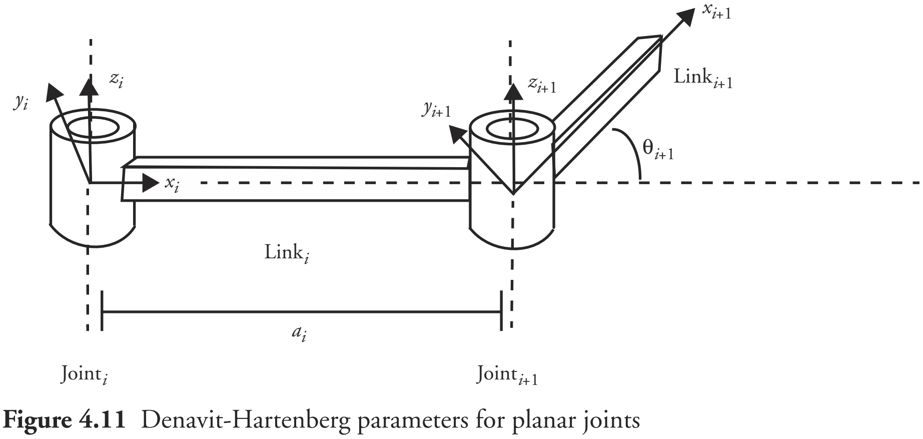


**Figure 4.11** Denavit-Hartenberg parameters for planar joints

Nonplanar configurations can be represented by including the two other DH parameters. For this general case, the *x*-axis of the *i*th joint is defined as the line segment perpendicular to the *z*-axes of the *i*th and $i + 1$ frames. The *link twist* parameter, $\alpha_i$, describes the rotation of the $i + 1$ frame's *z*-axis about this perpendicular relative to the *z*-axis of the *i*th frame. The *link offset* parameter, $d_{i+1}$, specifies the distance along the *z*-axis (rotated by $\alpha_i$) of the $i + 1$ frame from the *i*th *x*-axis to the $i + 1$ *x*-axis. See Figure 4.12.

Notice that the parameters associated with the *i*th joint do not all relate the *i*th frame to the $i + 1$ frame. The link length and link twist relate the *i*th and $i + 1$ frames; the link offset and joint rotation relate the $i - 1$ and *i*th frames. See Table 4.1.

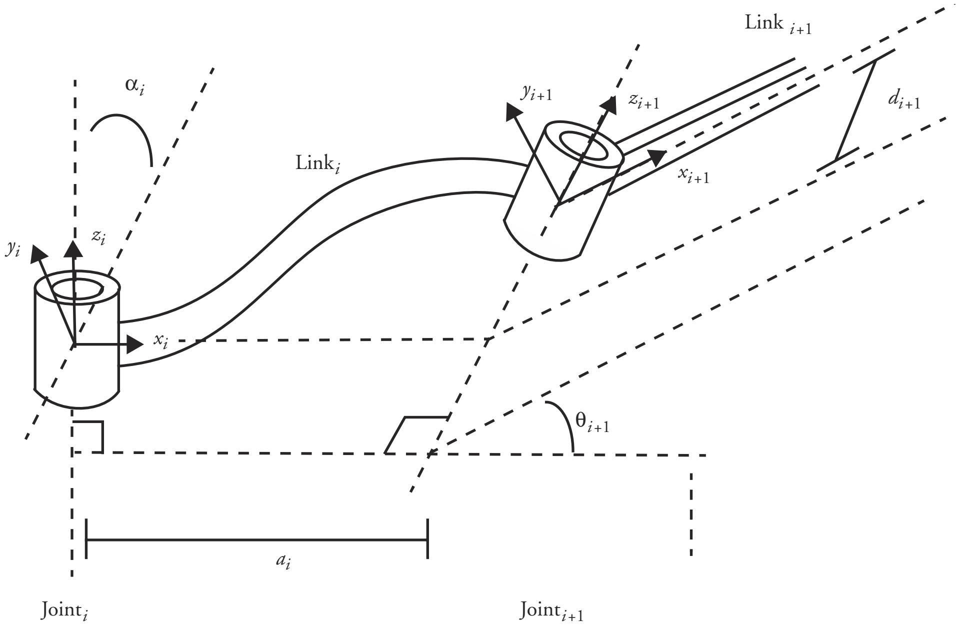


**Figure 4.12** Denavit-Hartenberg parameters

**Table 4.1** Denavit-Hartenberg Joint Parameters for Joint *i*

| Name | Symbol | Description |
|---|---|---|
| Link offset | $d_i$ | distance from $x_{i-1}$ to $x_i$ along $z_i$ |
| Joint angle | $\theta_i$ | angle between $x_{i-1}$ and $x_i$ about $z_i$ |
| Link length | $a_i$ | distance from $z_i$ to $z_{i+1}$ along $x_i$ |
| Link twist | $\alpha_i$ | angle between $z_i$ and $z_{i+1}$ about $x_i$ |

Stated another way, the parameters that describe the relationship of the $i + 1$ frame to the $i$th frame are a combination of $i$th joint parameters and $i + 1$ joint parameters. The parameters can be paired off to define two *screw transformations,* each of which consists of a translation and rotation relative to a single axis. The offset ($d_{i+1}$) and angle ($\theta_{i+1}$) are the translation and rotation of the $i + 1$ joint relative to the $i$th joint with respect to the $i$th joint's $z$-axis. The length ($a_i$) and twist ($\alpha_i$) are the translation and rotation of the $i + 1$ joint with respect to the $i$th joint's $x$-axis. See Table 4.2. The transformation of the $i + 1$ joint's frame from the $i$th frame can be constructed from a series of transformations, each of which corresponds to one of the DH parameters. As an example, consider a point, $V_{i+1}$, whose coordinates are given in the coordinate system of joint $i + 1$. To determine the point's coordinates in terms of the coordinate system of joint $i$, the transformation shown in Equation 4.6 is applied.

In Equation 4.6, $T$ and $R$ represent translation and rotation transformation matrices respectively; the parameter specifies the amount of rotation or translation, and the subscript specifies the axis involved. The matrix $M$ maps a point defined in the $i + 1$ frame into a point in the $i$th frame. By forming the $M$ matrix and its inverse associated with each pair of joints, one can convert points from one frame to another, up and down the hierarchy.

### A Simple Example

Consider the simple three-joint manipulator of Figure 4.13. The DH parameters are given in Table 4.3. The linkage is planar, so there are no displacement parameters and no twist parameters. Each successive frame is described by the joint angle and the length of the link.

**Table 4.2** Parameters That Relate the $i$th Frame and the $i + 1$ Frame

| Name | Symbol | Description | Screw Transformation |
|---|---|---|---|
| Link offset | $d_{i+1}$ | distance from $x_i$ to $x_{i+1}$ along $z_{i+1}$ | relative to $z_{i+1}$ |
| Joint angle | $\theta_{i+1}$ | angle between $x_i$ and $x_{i+1}$ about $z_{i+1}$ | relative to $z_{i+1}$ |
| Link length | $a_i$ | distance from $z_i$ to $z_{i+1}$ along $x_i$ | relative to $x_i$ |
| Link twist | $\alpha_i$ | angle between $z_i$ and $z_{i+1}$ about $x_i$ | relative to $x_i$ |

$$V_i = T_X(a_i) R_X(\alpha_i) T_Z(d_{i+1}) R_Z(\theta_{i+1}) V_{i+1}$$

$$R_Z(\theta_{i+1}) = \begin{bmatrix} \cos(\theta_{i+1}) & (-\sin(\theta_{i+1})) & 0 & 0 \\ \sin(\theta_{i+1}) & \cos(\theta_{i+1}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_Z(d_{i+1}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_{i+1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_X(\alpha_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_X(a_i) = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$V_i = M_i^{i+1} V_{i+1}$$

$$M_i^{i+1} = \begin{bmatrix} \cos(\theta_{i+1}) & -\sin(\theta_{i+1}) & 0 & a_i \\ \cos(\alpha_i) \cdot \sin(\theta_{i+1}) & \cos(\alpha_i) \cdot \cos(\theta_{i+1}) & -\sin(\alpha_i) & -d_{i+1} \cdot \sin(\alpha_i) \\ \sin(\alpha_i) \cdot \sin(\theta_{i+1}) & \sin(\alpha_i) \cdot \cos(\theta_{i+1}) & \cos(\alpha_i) & d_{i+1} \cdot \cos(\alpha_i) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**(Eq. 4.6)**

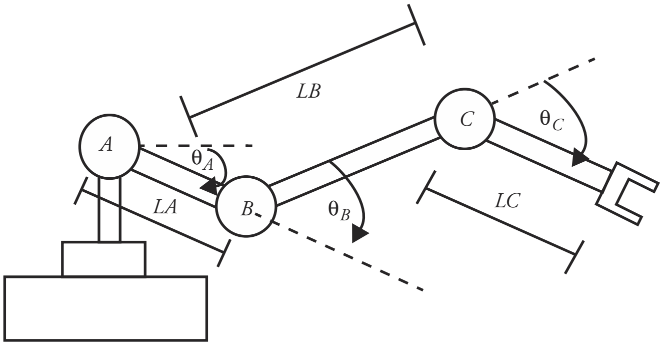


**Figure 4.13** Simple manipulator using three revolute joints

**Table 4.3** Parameters for Three-Revolute Joint Armature

| Joint/Parameter | Link Displacement | Joint Angle | Link Length | Link Twist |
|---|---|---|---|---|
| *A* | 0 | $\theta_A$ | 0 | 0 |
| *B* | 0 | $\theta_B$ | *LA* | 0 |
| *C* | 0 | $\theta_C$ | *LB* | 0 |

## Including a Ball-and-Socket Joint

Some human joints are conveniently modeled using a ball-and-socket joint. Consider an armature with a hinge joint, followed by a ball-and-socket joint followed by another hinge joint, as shown in Figure 4.14.

The DH notation can represent the ball-and-socket joint by three single DOF joints with zero-length links between them. See Figure 4.15.

Notice that in a default configuration with joint angles set to zero, the DH model of the ball-and-socket joint is in a gimbal lock position (incrementally changing two of the parameters results in rotation about the same axis). The first

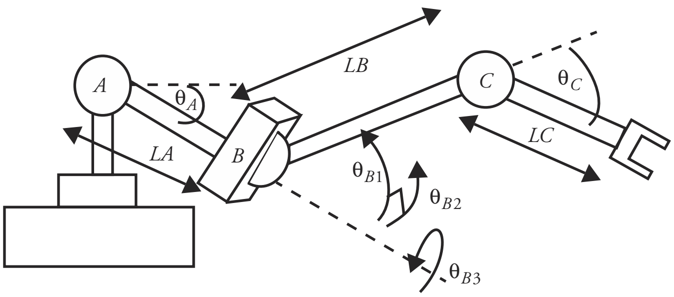


**Figure 4.14** Incorporating a ball-and-socket joint

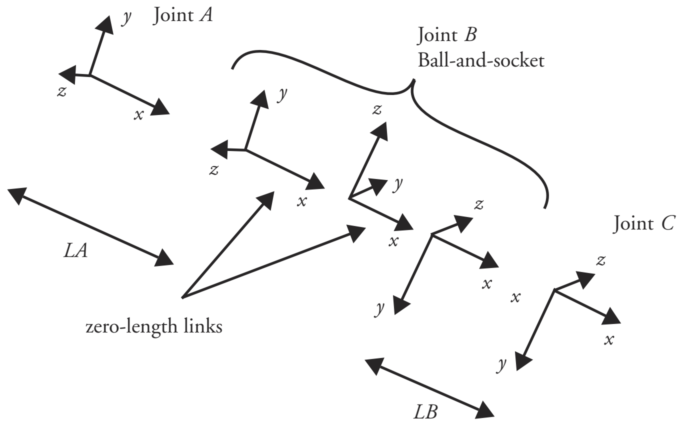


**Figure 4.15** Coordinate axes induced by the DH representation of a ball-and-socket joint

and third DOFs of that joint are aligned. The *z*-axes of these joints are colinear because the links between them are zero length and the two link twist parameters relating them are 90 degrees. This results in a total of 180 degrees and thus aligns the axes. As a consequence, the representation of the ball-and-socket joint is usually initialized with the middle of the three joint angles set to 90 degrees. See Table 4.4.

### Constructing the Frame Description

Because each frame's displacement and joint angle are defined relative to the previous frame, a Frame 0 is defined so that the Frame 1 displacement and angle can be defined relative to it. Frame 0 is typically defined so that it coincides with Frame 1 with zero displacement and zero joint angle. Similarly, because the link of the last frame does not connect to anything, the *x*-axis of the last frame is chosen so that it coincides with the *x*-axis of the previous frame when the joint angle is zero; the origin of the *n*th frame is chosen as the intersection of the *x*-axis of the previous frame and the joint axis when the displacement is zero.

**Table 4.4** Joint Parameters for Ball-and-Socket Joint

| Joint/Parameter | Link Displacement | Joint Angle | Link Length | Link Twist |
|---|---|---|---|---|
| *A* | 0 | $\theta_A$ | 0 | 0 |
| *B*1 | 0 | $\theta_{B1}$ | *LA* | 90 |
| *B*2 | 0 | $90 + \theta_{B2}$ | 0 | 90 |
| *B*3 | 0 | $\theta_{B3}$ | 0 | 0 |
| *C* | 0 | $\theta_C$ | *LB* | 0 |

The following procedure can be used to construct the frames for intermediate joints.

1. For each joint, identify the axis of rotation for revolute joints and the axis of displacement for prismatic joints. Refer to this axis as the *z*-axis of the joint's frame.
2. For each adjacent pair of joints, the *i*th – 1 and *i*th for *i* from 1 to *n,* construct the common perpendicular between the *z*-axes or, if they intersect, the perpendicular to the plane that contains them. Refer to the intersection of the perpendicular and the *i*th frame's *z*-axis (or the point of intersection of the two axes) as the origin of the *i*th frame. Refer to the perpendicular as the *x*-axis of the *i*th frame. See Figure 4.16.
3. Construct the *y*-axis of each frame to be consistent with the right-hand rule (assuming right-hand space).

### 4.2.4 Inverse Kinematics

In inverse kinematics, the desired position and possibly orientation of the end effector are given by the user, and the joint angles required to attain that configuration are calculated. The problem can have zero, one, or more solutions. If there are so many constraints on the configuration that no solution exists, the system is called *overconstrained.* If there are relatively few constraints on the system and there are many solutions to the problem posed, then it is *underconstrained.* The *reachable workspace* is that volume which the end effector can reach. The *dextrous workspace* is the volume that the end effector can reach in any orientation.

If the mechanism is simple enough, then the joint angles (the pose vector) required to produce the final, desired configuration can be calculated analytically. Given an initial pose vector and the final pose vector, intermediate configurations can be formed by interpolation of the values in the pose vectors, thus animating the mechanism from its initial configuration to the final one. However, if the

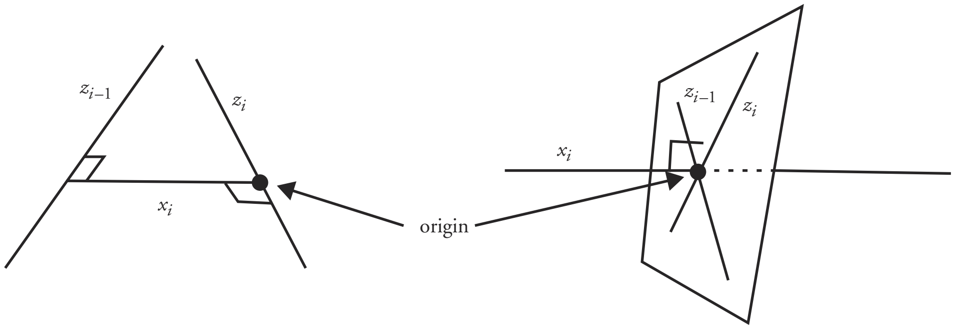


**Figure 4.16** Determining the origin and *x*-axis of the *i*th frame

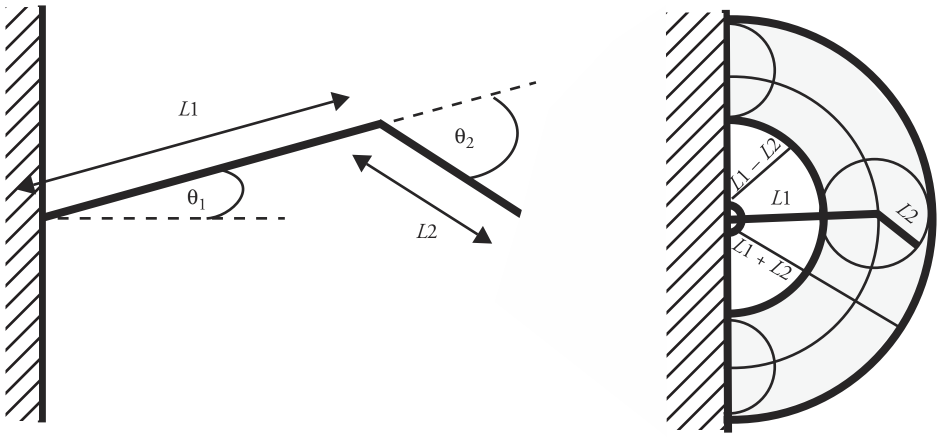
mechanism is too complicated for analytic solutions, then an incremental approach can be used that employs a matrix of values (the *Jacobian*) that relates changes in the joint angles to changes in the end effector position and orientation. The end effector is iteratively nudged until the final configuration is attained within a given tolerance.

### Solving a Simple System by Analysis

For sufficiently simple mechanisms, the joint angles of a final desired position can be determined analytically by inspecting the geometry of the linkage. Consider a simple two-link arm in two-dimensional space. Link lengths are $L1$ and $L2$ for the first and second link respectively. If a position is fixed for the base of the arm at the first joint, any position beyond $|L1 - L2|$ units from the base of the link and within $L1 + L2$ of the base can be reached. See Figure 4.17.

Assume for now (without loss of generality) that the base is at the origin. In a simple inverse kinematics problem, the user gives the $(X, Y)$ coordinate of the desired position for the end effector. The joint angles, $\theta 1$ and $\theta 2$, can be solved for by computing the distance from the base to the goal and using the law of cosines to compute the interior angles. Once the interior angles are computed, the rotation angles for the two links can be computed. See Figure 4.18. Of course, the first step is to make sure that the position of the goal is within the reach of the end effector; that is, $L1 - L2 \le \sqrt{X^2 + Y^2} \le L1 + L2$.

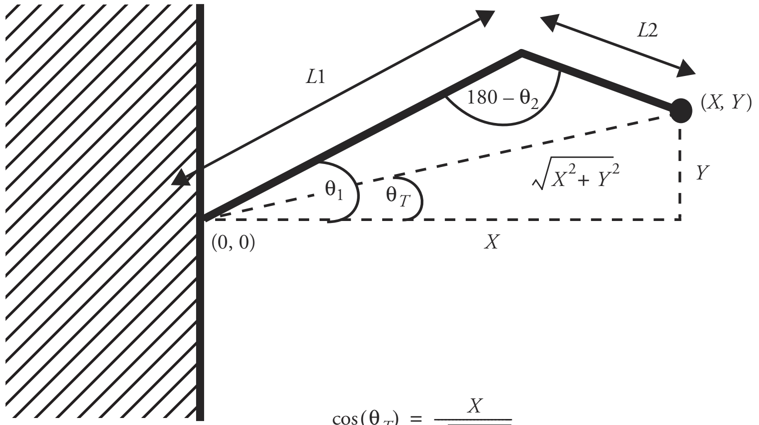
In this simple scenario, there are only two solutions that will give the correct answer; the configurations are symmetric with respect to the line from (0, 0) to $(X, Y)$. This is reflected in the equation in Figure 4.18 because the arccosine is



Configuration

Reachable workspace

**Figure 4.17** Simple linkage



$$\cos(\theta_T) = \frac{X}{\sqrt{X^2 + Y^2}}$$

$$\theta_T = \operatorname{acos}\left(\frac{X}{\sqrt{X^2 + Y^2}}\right)$$

$$\cos(\theta_1 - \theta_T) = \frac{L1^2 + X^2 + Y^2 - L2^2}{2 \cdot L1 \cdot \sqrt{X^2 + Y^2}} \qquad \text{(cosine rule)}$$

$$\theta_1 = \operatorname{acos}\left(\frac{L1^2 + X^2 + Y^2 - L2^2}{2 \cdot L1 \cdot \sqrt{X^2 + Y^2}}\right) + \theta_T$$

$$\cos(180 - \theta_2) = \frac{L1^2 + L2^2 - (X^2 + Y^2)}{2 \cdot L1 \cdot L2} \qquad \text{(cosine rule)}$$

$$\theta_2 = \operatorname{acos}\left(\left(\frac{L1^2 + L2^2 - (X^2 + Y^2)}{2 \cdot L1 \cdot L2}\right)\right)$$

**Figure 4.18** Equations used in solving simple inverse kinematic problem

two-valued in both plus and minus theta (θ). However, for more complicated armatures, there may be infinitely many solutions that will give the desired end effector location.

The joint angles for relatively simple linkages can be solved by algebraic manipulation of the equations that describe the relationship of the end effector to the base frame. Most linkages used in robotic applications are designed to be simple enough for this analysis. However, for many cases that arise in computer animation, analytic solutions are not tractable. In such cases, iterative numeric solutions must be relied on.

## The Jacobian

Most mechanisms of interest to computer animation are too complex to allow an analytic solution. For these, the motion can be incrementally constructed. At each time step, a computation is performed that determines the best way to change each joint angle in order to direct the current position and orientation of the end effector toward the desired configuration. The computation forms the matrix of partial derivatives called the *Jacobian*.

To explain the Jacobian from a strictly mathematical point of view, consider the six arbitrary functions of Equation 4.7, each of which is a function of six independent variables. Given specific values for the input variables, $x_i$, each of the output variables, $y_i$, can be computed by its respective function.

$$\begin{aligned} y_1 &= f_1(x_1, x_2, x_3, x_4, x_5, x_6) \\ y_2 &= f_2(x_1, x_2, x_3, x_4, x_5, x_6) \\ y_3 &= f_3(x_1, x_2, x_3, x_4, x_5, x_6) \\ y_4 &= f_4(x_1, x_2, x_3, x_4, x_5, x_6) \\ y_5 &= f_5(x_1, x_2, x_3, x_4, x_5, x_6) \\ y_6 &= f_6(x_1, x_2, x_3, x_4, x_5, x_6) \end{aligned} \qquad \text{(Eq. 4.7)}$$

These equations can also be used to describe the change in the output variables relative to the change in the input variables. The differentials of $y_i$ can be written in terms of the differentials of $x_i$ using the chain rule. This generates Equation 4.8. Equation 4.7 and Equation 4.8 can be put in vector notation, producing Equation 4.9 and Equation 4.10 respectively.

$$\begin{aligned} \delta y_i &= \frac{\delta f_i}{\partial x_1} \cdot \delta x_1 + \frac{\delta f_i}{\partial x_2} \cdot \delta x_2 + \frac{\delta f_i}{\partial x_3} \cdot \delta x_3 + \frac{\delta f_i}{\partial x_4} \cdot \delta x_4 \\ &\quad + \frac{\delta f_i}{\partial x_5} \cdot \delta x_5 + \frac{\delta f_i}{\partial x_6} \cdot \delta x_6 \end{aligned} \qquad \text{(Eq. 4.8)}$$

$$Y = F(X) \qquad \text{(Eq. 4.9)}$$

$$\delta Y = \frac{\partial F}{\partial X} \cdot \partial X \qquad \text{(Eq. 4.10)}$$

The 6x6 matrix of partial derivatives, $\partial F/\partial X$, is called the *Jacobian* and is a function of the current values of $x_i$. The Jacobian can be thought of as mapping the velocities of $X$ to the velocities of $Y$ (Equation 4.11). At any point in time, the Jacobian is a linear function of $x_i$. At the next instant of time, $X$ has changed and so has the linear transformation represented by the Jacobian.

$$\dot{Y} = J(X) \cdot \dot{X} \qquad \text{(Eq. 4.11)}$$

When one applies the Jacobian to a linked appendage, the input variables, $x_i$, become the joint angles and the output variables, $y_i$, become the end effector position and orientation. In this case, the Jacobian relates the velocities of the joint angles to the velocities of the end effector position and orientation (Equation 4.12).

$$V = J(\theta)\dot{\theta} \tag{Eq. 4.12}$$

$V$ is the vector of linear and rotational velocities and represents the desired change in the end effector. The desired change will be based on the difference between its current position/orientation to that specified by the goal configuration. These velocities are vectors in three-space, so each has an $x$, $y$, and $z$ component (Equation 4.13). $\dot{\theta}$ is a vector of joint angle velocities which are the unknowns of the equation (Equation 4.14). $J$, the Jacobian, is a matrix that relates the two and is a function of the current pose (Equation 4.15).

$$V = [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]^T \tag{Eq. 4.13}$$

$$\dot{\theta} = [\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \ldots, \dot{\theta}_n]^T \tag{Eq. 4.14}$$

$$J = \begin{bmatrix} \frac{\partial v_x}{\partial \theta_1} & \frac{\partial v_x}{\partial \theta_2} & \cdots & \frac{\partial v_x}{\partial \theta_n} \\ \frac{\partial v_y}{\partial \theta_1} & \frac{\partial v_y}{\partial \theta_2} & \cdots & \frac{\partial v_y}{\partial \theta_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial \omega_z}{\partial \theta_1} & \frac{\partial \omega_z}{\partial \theta_2} & \cdots & \frac{\partial \omega_z}{\partial \theta_n} \end{bmatrix} \tag{Eq. 4.15}$$

Each term of the Jacobian relates the change of a specific joint to a specific change in the end effector. The rotational change in the end effector, $\omega$, is merely the velocity of the joint angle about the axis of revolution at the joint under consideration. The linear change in the end effector is the cross product of the axis of revolution and a vector from the joint to the end effector. The rotation at the joint induces an instantaneous linear direction of travel at the end effector. See Figure 4.19.

The desired angular and linear velocities are computed by finding the difference between the current configuration of the end effector and the desired configuration. The angular and linear velocities of the end effector induced by the rotation of a specific joint axis are determined by the computations shown in Figure 4.19.

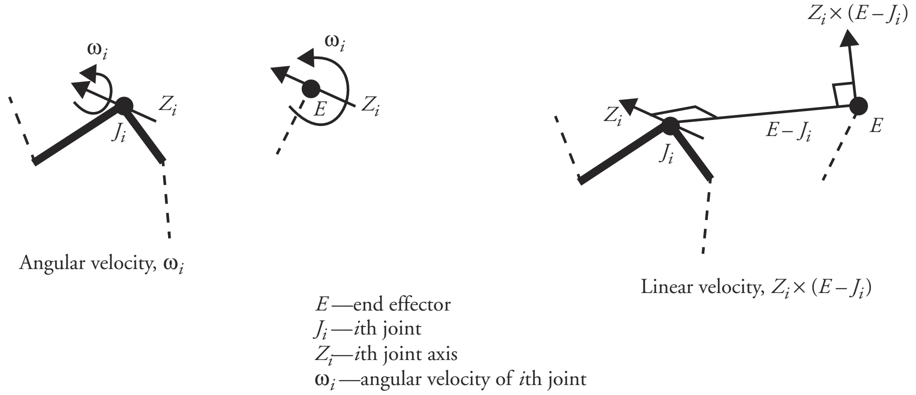


**Figure 4.19** Angular and linear velocities induced by joint axis rotation

The problem is to determine the best linear combination of velocities induced by the various joints that would result in the desired velocities of the end effector. The Jacobian is formed by posing the problem in matrix form.

When one assembles the Jacobian, it is important to make sure that all of the coordinate values are in the same coordinate system. It is often the case that joint-specific information is given in the coordinate system local to that joint. In forming the Jacobian matrix, this information must be converted into some common coordinate system such as the global inertial coordinate system or the end effector coordinate system. Various methods have been developed for computing the Jacobian based on attaining maximum computational efficiency given the required information in local coordinate systems, but all methods produce the derivative matrix in a common coordinate system.

### A Simple Example

Consider the simple three-revolute-joint, planar manipulator of Figure 4.20. In this example, the objective is to move the end effector, *E,* to the goal position, *G.* The orientation of the end effector is of no concern in this example. The axis of rotation of each joint is perpendicular to the figure, coming out of the paper. The effect of an incremental rotation, $g_i$, of each joint can be determined by the cross product of the joint axis and the vector from the joint to the end effector, $V_i$ (Figure 4.21). Notice that the magnitude of each $g_i$ is a function of the distance between the locations of the joint and the end effector.

The desired change to the end effector is the difference between the current position of the end effector and the goal position. A vector of the desired change in values is set equal to the Jacobian matrix multiplied by a vector of the unknown values, which are the changes to the joint angles (Equation 4.16).

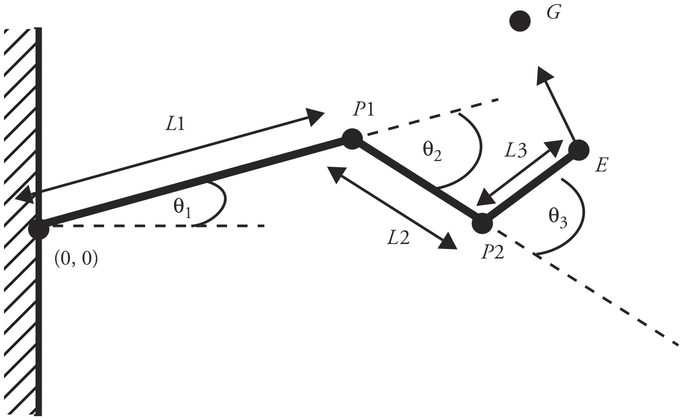


**Figure 4.20** Planar, three-joint manipulator

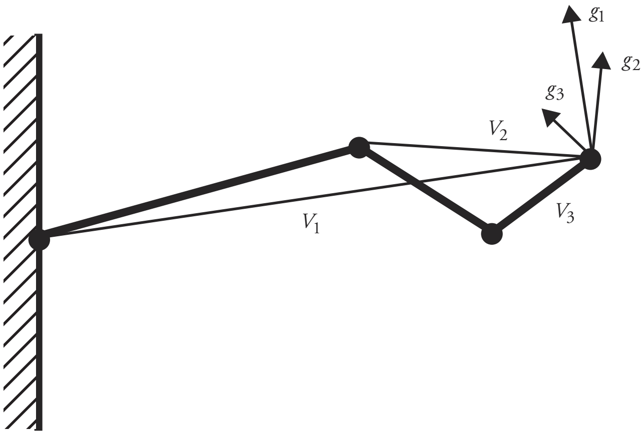


**Figure 4.21** Instantaneous changes in position induced by joint angle rotations

$$\begin{bmatrix} (G-E)_x \\ (G-E)_y \\ (G-E)_z \end{bmatrix}$$

$$= \begin{bmatrix} ((0,0,1)\times E)_x & (0,0,1)\times(E-P_1)_x & (0,0,1)\times(E-P_2)_x \\ ((0,0,1)\times E)_y & (0,0,1)\times(E-P_1)_y & (0,0,1)\times(E-P_2)_y \\ ((0,0,1)\times E)_z & (0,0,1)\times(E-P_1)_z & (0,0,1)\times(E-P_2)_z \end{bmatrix} \tag{Eq. 4.16}$$

$$\cdot \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{bmatrix}$$

### Solution Using the Inverse Jacobian

Once the Jacobian has been computed, an equation in the form of Equation 4.17 is to be solved. In the case that $J$ is a square matrix, the inverse of the Jacobian, $J^{-1}$, is used to compute the joint angle velocities given the end effector velocities (Equation 4.18).

$$V = J\dot{\theta} \tag{Eq. 4.17}$$

$$J^{-\dot{1}}V = \dot{\theta} \tag{Eq. 4.18)}$$

If the inverse of the Jacobian ($J^{-1}$) does not exist, then the system is said to be singular for the given joint angles. A singularity occurs when a linear combination of the joint angle velocities cannot be formed to produce the desired end effector velocities. As a simple example of such a situation, consider a fully extended, planar arm with a goal position somewhere on the forearm (see Figure 4.22). In such a case, a change in each joint angle would produce a vector perpendicular to the desired direction. Obviously, no linear combination of these vectors could produce the desired motion vector. Unfortunately, all of the singularities of a system cannot be determined simply by visually inspecting the possible geometric configurations of the linkage.

Problems with singularities can be reduced if the manipulator is redundant—when there are more degrees of freedom than there are constraints to be satisfied.

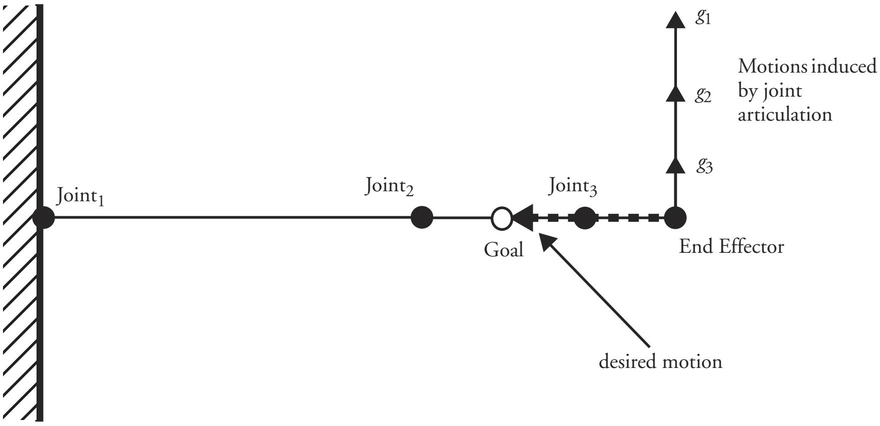


**Figure 4.22** Simple example of a singular configuration

In this case, the Jacobian is not a square matrix and there are an infinite number of solutions to the inverse kinematics problem. Because the Jacobian is not square, a conventional inverse does not exist. Instead, the *pseudo inverse,* $J^+$, can be used (Equation 4.19). Equation 4.19 works because a matrix multiplied by its own transpose will be a square matrix.

$$V = J\dot{\theta}$$
$$J^T V = J^T J\dot{\theta}$$
$$(J^T J)^{-1} J^T V = (J^T J)^{-1} J^T J\dot{\theta}$$
$$J^+ V = \dot{\theta} \qquad \text{(Eq. 4.19)}$$

$J^+ = (J^T J)^{-1} J^T = J^T (JJ^T)^{-1}$ is called the *pseudo inverse* of *J*. It maps the desired velocities of the end effector to the required velocities of the joint angles. After making the substitutions shown in Equation 4.20, LU decomposition can be used to solve Equation 4.21 for $\beta$. This can then be substituted into Equation 4.22 to solve for $\dot{\theta}$.

$$J^+ V = \dot{\theta}$$
$$J^T (JJ^T)^{-1} V = \dot{\theta}$$
$$\beta = (JJ^T)^{-1} V \qquad \text{(Eq. 4.20)}$$

$$(JJ^T)\beta = V \qquad \text{(Eq. 4.21)}$$

$$J^T\beta = \dot{\theta} \tag{Eq. 4.22}$$

It is important to remember that the Jacobian is only valid for the instantaneous configuration for which it is formed. That is, as soon as the configuration of the linkage changes, the Jacobian ceases to accurately describe the relationship between changes in joint angles and changes in end effector position and orientation. This means that if too big a step is taken in joint angle space, the end effector may not appear to travel in the direction of the goal. If this appears to happen during an animation sequence, then taking smaller steps in joint angle space and thus recalculating the Jacobian more often may be in order.

### Adding More Control

The pseudo inverse computes one of many possible solutions. It minimizes joint angle rates. The configurations produced, however, do not necessarily correspond to what might be considered natural poses. To better control the kinematic model, a control expression can be added to the pseudo inverse Jacobian solution. The control expression is used to solve for control angle rates with certain attributes. The added control expression, because of its form, contributes nothing to the desired end effector motion. The form for the control expression is shown in Equation 4.23. In Equation 4.24 it is shown that this form of the control expression does not add anything to the velocities. As a consequence, the control expression can be combined with the pseudo inverse Jacobian solution so that the given velocities are still satisfied [27].

$$\dot{\theta} = (J^+J - I)z \tag{Eq. 4.23}$$

$$\begin{aligned} V &= J\dot{\theta} \\ V &= J(J^+J - I)z \\ V &= (JJ^+J - J)z \\ V &= (J - J)z \\ V &= 0 \cdot z \\ V &= 0 \end{aligned} \tag{Eq. 4.24}$$

To bias the solution toward specific joint angles, such as the middle angle between joint limits, $H$ is defined as in Equation 4.25, where $\theta_i$ are the current joint angles, $\theta_{ci}$ are the desired joint angles, $\alpha_i$ are the desired angle gains, and $\psi$ is the $\psi$th norm (for $\psi$ even). Variable $z$ is equal to the gradient of $H$, $\nabla H$ (Equation 4.26). This does not enforce joint limits as hard constraints, but the solution can be biased toward the middle values so that violating the joint limits is less probable.

$$H = \sum_{i=1}^{n} \alpha_i \cdot (\theta_i - \theta_{ci})^{\psi} \qquad \text{(Eq. 4.25)}$$

$$z = \nabla_{\theta} H = \frac{dH}{d\theta} = \psi \sum_{i=1}^{n} \alpha_i \cdot (\theta_i - \theta_{ci})^{\psi - 1} \qquad \text{(Eq. 4.26)}$$

The desired angles and gains are input parameters. The gain indicates the relative importance of the associated desired angle; the higher the gain, the stiffer the joint.[2] If the gain for a particular joint is high, then the solution will be such that the joint angle quickly approaches the desired joint angle. The control expression is added to the solution indicated by the conventional pseudo inverse of the Jacobian (Equation 4.27). If all gains are zero, then the solution will reduce to the conventional pseudo inverse of the Jacobian. Equation 4.27 can be solved by rearranging terms as shown in Equation 4.28.

$$\dot{\theta} = J^{+}V + (J^{+}J - I)\nabla_{\theta}H \qquad \text{(Eq. 4.27)}$$

$$\begin{aligned}
\dot{\theta} &= J^{+}V + (J^{+}J - I)\nabla_{\theta}H \\
\dot{\theta} &= J^{+}V + J^{+}J\nabla_{\theta}H - I\nabla_{\theta}H \\
\dot{\theta} &= J^{+}(V + J\nabla_{\theta}H) - \nabla_{\theta}H \\
\dot{\theta} &= J^{T}(JJ^{T})^{-1}(V + J\nabla_{\theta}H) - \nabla_{\theta}H \\
\dot{\theta} &= J^{T}[(JJ^{T})^{-1}(V + J\nabla_{\theta}H)] - \nabla_{\theta}H \qquad \text{(Eq. 4.28)}
\end{aligned}$$

To solve Equation 4.28, set $\beta = (JJ^{T})^{-1}(V + J\nabla_{\theta}H)$ so that Equation 4.29 results. Use LU decomposition to solve for $\beta$ in Equation 4.30. Substitute the solution for $\beta$ in Equation 4.29 to solve for $\dot{\theta}$.

$$\dot{\theta} = J^{T}\beta - \nabla_{\theta}H \qquad \text{(Eq. 4.29)}$$

$$V + J\nabla_{\theta}H = (JJ^{T})\beta \qquad \text{(Eq. 4.30)}$$

[2] *Stiffness* refers to how much something reacts to being perturbed. A stiff spring is a strong spring. A stiff joint, as used here, is a joint that has a higher resistance to being pulled away from its desired value.

Simple Euler integration can be used at this point to update the joint angles. The Jacobian has changed at the next time step, so the computation must be performed again and another step taken. This process repeats until the end effector reaches the goal configuration within some acceptable (i.e., user-defined) tolerance.

### 4.2.5 Summary

Hierarchical models are extremely useful for enforcing certain relationships among the elements so that the animator can concentrate on just the degrees of freedom remaining. Forward kinematics gives the animator explicit control over each degree of freedom but can become cumbersome when the animation is trying to attain a specific position or orientation of an element at the end of a hierarchical chain. Inverse kinematics, using the inverse or pseudo inverse of the Jacobian, allows the animator to concentrate only on the conditions at the end of such a chain but might produce undesirable configurations. Additional control expressions can be added to the pseudo inverse Jacobian solution to express a preference for solutions of a certain character. However, these are all kinematic techniques. Often, more realistic motion is desired and physically based simulations are needed. These approaches are discussed below.

## 4.3 Rigid Body Simulation

A common objective in computer animation is to create realistic-looking motion. A major component of realistic motion is the physically based reaction of rigid bodies to commonly encountered forces such as gravity, viscosity, friction, and those resulting from collisions. Creating realistic motion with key-frame techniques can be a daunting task. However, the equations of motion can be incorporated into an animation system to automatically calculate these reactions. This can eliminate considerable tedium—if the animator is willing to relinquish precise control over the motion of some objects.

In rigid body simulation, various forces to be simulated are modeled in the system. These forces may arise due to relative positioning of objects (e.g., gravity, collisions), object velocity (e.g., viscosity), or the absolute position of objects in user-specified vector fields (e.g., wind). When applied to objects, these forces induce linear and angular accelerations based on the mass of the object (in the linear case) and mass distribution of the object (in the angular case). These accelerations, which are the time derivative of velocities, are integrated over a delta time step to produce changes in object velocities (linear and angular). These velocities, in turn integrated over a delta time step, produce changes in object positions and orienta-

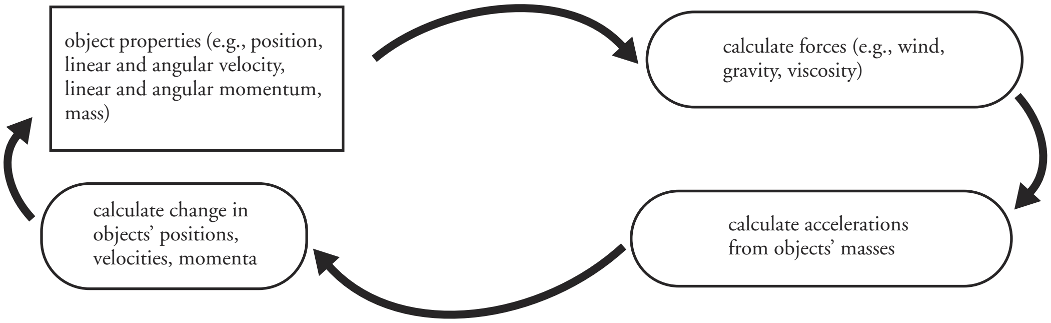

object properties (e.g., position, linear and angular velocity, linear and angular momentum, mass)
calculate forces (e.g., wind, gravity, viscosity)
calculate accelerations from objects' masses
calculate change in objects' positions, velocities, momenta

**Figure 4.23** Rigid body simulation update cycle

tions (Figure 4.23). The new positions and velocities of the objects give rise to new forces, and the process repeats for the next time step.

The free flight of objects through space is a simple type of rigid body simulation. The simulation of rigid body physics becomes more complex as objects collide, roll and slide over one another, and come to rest in complex arrangements. In addition, a persistent issue in rigid body simulation, as with most animation, is the modeling of a continuous process (such as physics) with discrete time steps. The trade-off of accuracy for computational efficiency is an ever-present consideration.

It should be noted that the difference between material commonly taught in standard physics texts and that used in computer animation is in how the equations of motion are used. In standard physics, the emphasis is usually in analyzing the equations of motion for times at which significant events happen, such as an object hitting the ground. In computer animation, the concern is with modeling the motion of objects at discrete time steps [16] as well as the significant events and their aftermath. While the fundamental principles and basic equations are the same, the discrete time sampling creates numerical issues that must be dealt with carefully. See Appendix B for equations of motion from basic physics.

## 4.3.1 Bodies in Free Fall

To understand the basics in modeling physically based motion, the motion of a point in space will be considered first. The position of the point at discrete time steps is desired, where the interval between these time steps is some uniform $\Delta t$. To update the position of the point over time, its position, velocity, and acceleration are used, which are modeled as functions of time, $x(t)$, $v(t)$, $a(t)$, respectively.

If there are no forces applied to a point, then a point's acceleration is zero and its velocity remains constant (possibly nonzero). In the absence of acceleration, the point's position, $x(t)$, is updated by its velocity, $v(t)$, as in Equation 4.31. A point's velocity, $v(t)$, is updated by its acceleration, $a(t)$. Acceleration arises from forces applied to an object over time. To simplify the computation, a point's acceleration is usually assumed to be constant over the time period $\Delta t$. See Equation 4.32. A point's position is updated by the average velocity during the time period $\Delta t$. Under the constant-acceleration assumption, the average velocity during a time period is the average of its beginning velocity and ending velocity, as in Equation 4.33. By substituting Equation 4.32 into Equation 4.33, one defines the updated position in terms of the starting position, velocity, and acceleration (Equation 4.34).

$$x(t+\Delta t) = x(t) + v(t)\cdot\Delta t \tag{Eq. 4.31}$$

$$v(t+\Delta t) = v(t) + a(t)\cdot\Delta t \tag{Eq. 4.32}$$

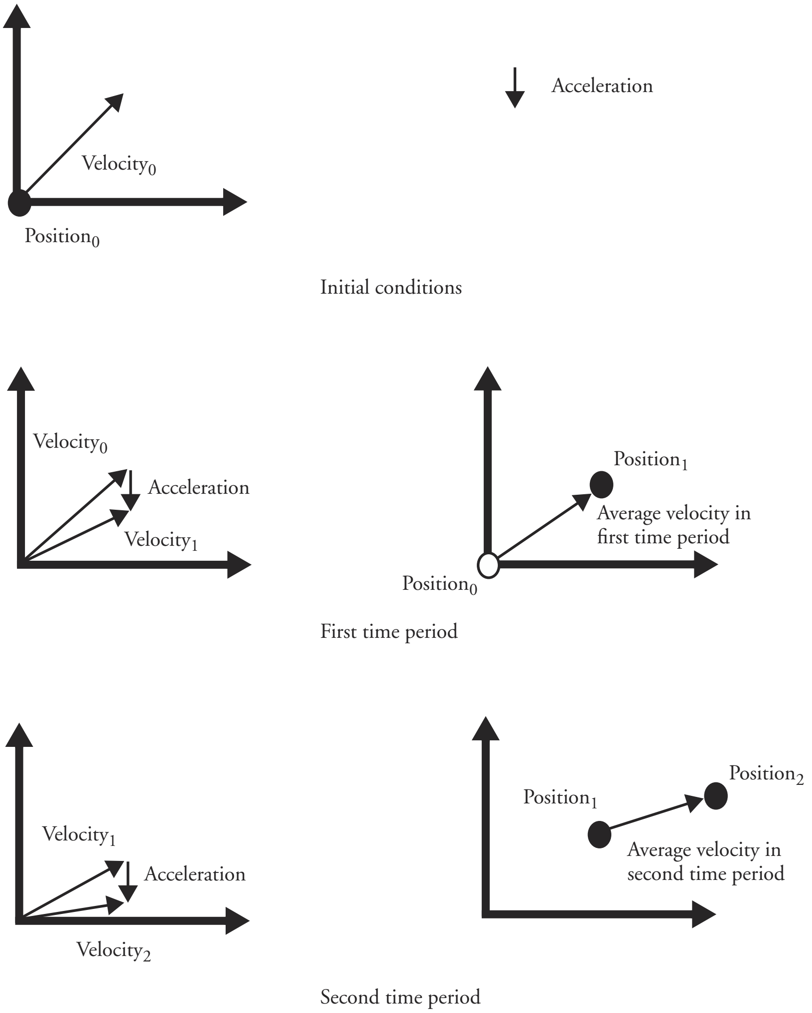
$$x(t+\Delta t) = x(t) + ((v(t) + v(t+\Delta t))/2)\cdot\Delta t \tag{Eq. 4.33}$$

$$x(t+\Delta t) = x(t) + v(t)\cdot\Delta t + \frac{1}{2}\cdot a(t)\cdot\Delta t^2 \tag{Eq. 4.34}$$

### A Simple Example

Using a standard physics example, consider a point with an initial position of (0, 0), with an initial velocity of (100, 100) feet per second, and under the force of gravity resulting in a uniform acceleration of (0, –32) feet per second per second. Assume a delta time interval of one-thirtieth of a second (corresponding roughly to the frame interval in NTSC video). In this example, the acceleration is uniformly applied throughout the sequence, and the velocity is modified at each time step by the downward acceleration. For each time interval, the average of the beginning and ending velocities is used to update the position of the point. This process is repeated for each step in time. See Equation 4.35 and Figures 4.24 and 4.25.

$$
\begin{aligned}
v_{\frac{1}{30}} &= \{100, 100\} + \{0, -32\}\cdot\frac{1}{30} = \left\{100, \frac{1484}{15}\right\}\\
x_{\frac{1}{30}} &= \{0, 0\} + \frac{1}{2}\cdot\left(\left\{100, \frac{1484}{15}\right\} + \{100, 100\}\right)\cdot\frac{1}{30} = \left\{\frac{10}{3}, \frac{746}{225}\right\}\\
v_{\frac{1}{15}} &= \left\{100, \frac{1484}{15}\right\} + \{0, -32\}\cdot\frac{1}{30} = \left\{100, \frac{1468}{15}\right\}\\
x_{\frac{1}{15}} &= \left\{\frac{10}{3}, \frac{746}{225}\right\} + \frac{1}{2}\cdot\left(\left\{100, \frac{1468}{15}\right\} + \left\{100, \frac{1484}{15}\right\}\right)\cdot\frac{1}{30} = \left\{\frac{20}{3}, \frac{1484}{225}\right\}\\
v_{\frac{1}{10}} &= \left\{100, \frac{1468}{15}\right\} + \{0, -32\}\cdot\frac{1}{30} = \left\{100, \frac{484}{5}\right\}\\
x_{\frac{1}{10}} &= \left\{\frac{20}{3}, \frac{1484}{225}\right\} + \frac{1}{2}\cdot\left(\left\{100, \frac{484}{5}\right\} + \left\{100, \frac{1468}{15}\right\}\right)\cdot\frac{1}{30} = \left\{10, \frac{246}{25}\right\}\\
&\text{etc.}
\end{aligned}
\tag{Eq. 4.35}
$$



Initial conditions

First time period

Second time period

**Figure 4.24** Modeling of a point's position at discrete time intervals (vector lengths are for illustrative purposes and are not accurate)

### A Note about Numeric Approximation

The assumption that acceleration remains constant over the delta time step is incorrect in most rigid body simulations; many forces continually vary as the object changes its position and velocity over time. This means that the acceleration actually varies over the time step, and it often varies in nonlinear ways. Sampling the force at the beginning of the time step and using that to determine the acceleration throughout the time step is not the best approach. The multiplication of acceleration at the beginning of the time step by $\Delta t$ to step to the next function value (velocity) is an example of using the Euler integration method (Figure 4.26).

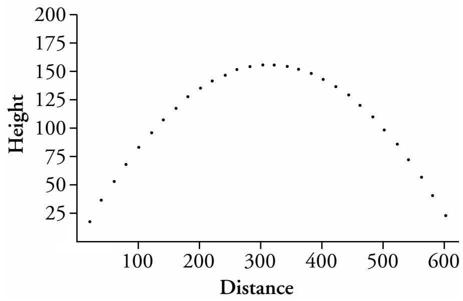


**Figure 4.25** Path of a particle in the simple example from the text

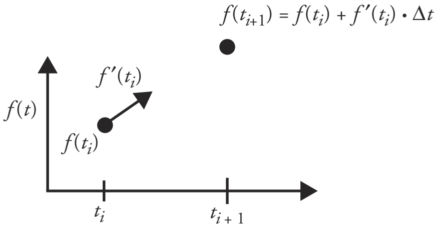
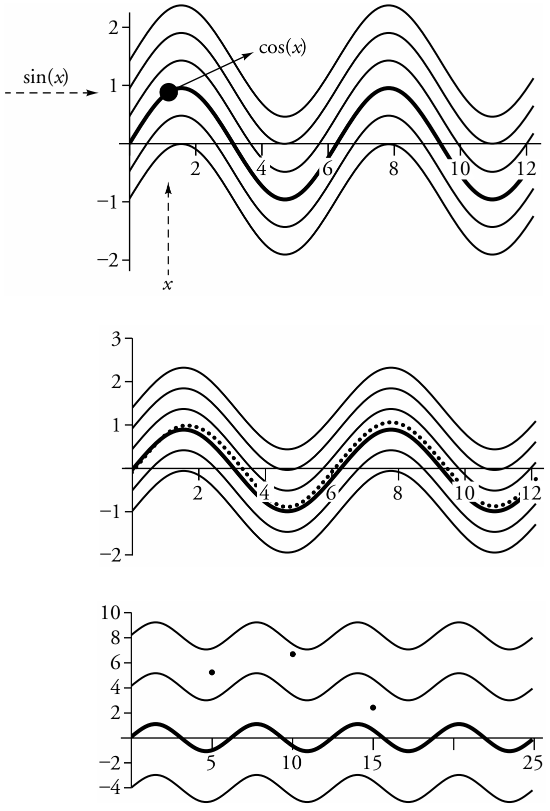


**Figure 4.26** Euler integration

It is important to understand the shortcomings of this approach and the available options for improving the accuracy of the simulation. Many books have been written about the subject; *Numerical Recipes: The Art of Scientific Computing,* by Press et al. [22], is a good place to start. This short section is intended merely to demonstrate that better options exist and that, at the very least, a Runge-Kutta method should be considered.

It is easy to see how the size of the time step affects accuracy (Figure 4.27). By taking time steps that are too large, the numerically approximated path deviates dramatically from the ideal continuous path. Accuracy can be increased by taking smaller steps, but this can prove to be computationally expensive.

Accuracy can also be increased by using better methods of integration. *Runge-Kutta* is a particularly useful one. Figure 4.28 shows the advantage of using the *second-order Runge-Kutta,* or *midpoint,* method, which uses derivative information from the midpoint of the stepping interval. *Second-order* refers to the magnitude of the error term. Higher-order Runge-Kutta methods are more accurate and therefore allow larger time steps to be taken, but they are more computationally expensive per time step. Generally, it is better to use a fourth- or fifth-order Runge-Kutta method.



a) In this example, the sine function is the underlying (unknown) function. The objective is to reconstruct it based on an initial point and knowledge about the derivative function (cosine). Start out at any ($x$, $y$) location and, if the reconstruction is completely accurate, the sinusoidal curve that passes through that point should be followed. In the examples below, the initial point is (0, 0) so the thicker sine curve should be followed.

b) Updating the function values by taking small enough steps along the direction indicated by the derivative generates a good approximation to the function. In this example, $\Delta x = 0.2$.

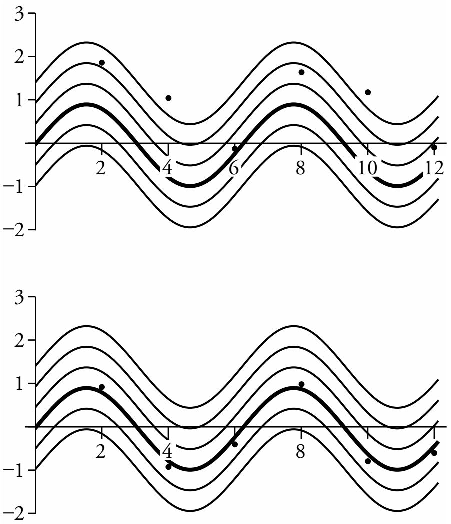
c) However, if the step size becomes too large, then the function reconstructed from the sample points can deviate widely from the underlying function. In this example, $\Delta x = 5$.

**Figure 4.27** Approximating the sine curve by stepping in the direction of its derivative

While computer animation is concerned primarily with visual effects and not numeric accuracy, it is still important to keep an eye on the numerics that underlie any simulation responsible for producing the motion. Visual realism can be compromised if the numeric calculation is allowed to become too sloppy. One should have a basic understanding of the strengths and weaknesses of the numeric techniques used, and, in most cases, employing the Euler method should be done with caution.

## Equations of Motion for a Rigid Body

To develop the equations of motion for a rigid body, several concepts from physics are presented first [16]. The rotational equivalent of linear force, or *torque,* needs to be considered when a force is applied to an object not directly in line with its *center of mass.* To uniquely solve for the resulting motions of interacting objects, *linear momentum* and *angular momentum* have to be conserved. And, finally, to calculate the angular momentum, the distribution of an object's mass in space



Using step sizes of 2 with the Euler method

Using step sizes of 2 with the midpoint method

**Figure 4.28** Euler method and midpoint method

must be characterized by its *inertia tensor*. These concepts are discussed below and are followed by the equations of motion.

### Orientation and Rotational Movement

Similar to linear attributes of position, velocity, and acceleration, three-dimensional objects have rotational attributes of orientation, angular velocity, and angular acceleration as functions of time. If an individual point in space is modeled, such as in a particle system, then its rotational information can be ignored. Otherwise the physical extent of the mass of an object needs to be taken into consideration in realistic physical simulations.

For current purposes, consider an object's orientation to be represented by a rotation matrix, $R(t)$. *Angular velocity* is the rate at which the object is rotating irrespective of its linear velocity. It is represented by a vector, $\omega(t)$. The direction of the vector indicates the orientation of the axis about which the object is rotating; the magnitude of the angular velocity vector gives the speed of the rotation in revolutions per unit of time. For a given number of rotations in a given time period, the angular velocity of an object is the same whether the object is rotating about its own axis or rotating about an axis some distance away. If an object is rotating about its own axis at the rate of two revolutions per minute, it will have the same angular velocity as when it is rotating two revolutions per minute about an axis ten miles away. In the latter case, the rotation will also induce an instantaneous linear velocity (which constantly changes). But in both cases the object is still rotating at two revolutions per minute (Figure 4.29).

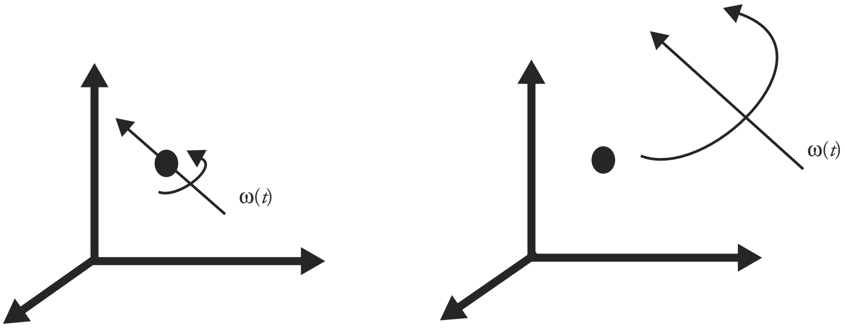


**Figure 4.29** For a given number of rotations per unit of time, the angular velocity is the same whether the axis of rotation is near or far away

Consider a point, *a,* whose position in space is defined relative to a point, $b = x(t)$; *a*'s position relative to *b* is defined by $r(t)$. The point *a* is rotating and the axis of rotation passes through the point *b* (Figure 4.30). The change in $r(t)$ is computed by taking the cross product of $r(t)$ and $\omega(t)$ (Equation 4.36). Notice that the change in $r(t)$ is perpendicular to the plane formed by $\omega(t)$ and $r(t)$ and that the magnitude of the change is dependent on the perpendicular distance between $\omega(t)$ and $r(t)$, as well as the magnitudes of $\omega(t)$ and $r(t)$.

$$\begin{aligned} \dot{r}(t) &= \omega(t) \times r(t) \\ \left|\dot{r}(t)\right| &= |\omega(t)||r(t)|\sin\theta \end{aligned} \qquad \text{(Eq. 4.36)}$$

Now consider an object that has an extent (distribution of mass) in space. The orientation of an object, represented by a rotation matrix, can be viewed as a transformed version of the object's local unit coordinate system. As such, its columns can be viewed as vectors defining relative positions in the object. Thus, the change in the rotation matrix can be computed by taking the cross product of $\omega(t)$ with each of the columns of $R(t)$ (Equation 4.37). By defining a special

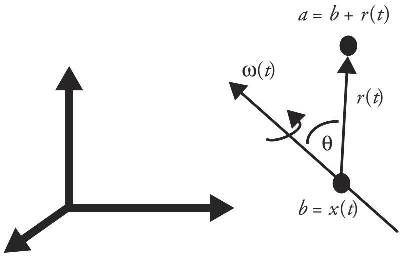


**Figure 4.30** A point rotating about an axis

matrix to represent cross products (Equation 4.38), one can represent Equation 4.37 by matrix multiplication (Equation 4.39).

$$R(t) = \left\lfloor R_1(t) \quad R_2(t) \quad R_3(t) \right\rfloor$$

$$\dot{R}(t) = \left\lfloor \omega(t) \times R_1(t) \quad \omega(t) \times R_2(t) \quad \omega(t) \times R_3(t) \right\rfloor \qquad \text{(Eq. 4.37)}$$

$$A \times B = \begin{bmatrix} A_y \cdot B_z - A_z \cdot B_y \\ -(A_z \cdot B_x) + A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{bmatrix} = \begin{bmatrix} 0 & -A_x & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} = A^* B \qquad \text{(Eq. 4.38)}$$

$$\dot{R}(t) = \omega(t)^* R(t) \qquad \text{(Eq. 4.39)}$$

Consider a point, *Q*, on a rigid object. Its position in the local coordinate space of the object is *q; q* does not change. Its position in world space, $q(t)$, is given by Equation 4.40. The position of the body in space is given by $x(t)$, and the orientation of the body is given by $R(t)$. The velocity of the particle is given by differentiating Equation 4.40. The relative position of the particle in the rigid body is constant. The change in orientation is given by Equation 4.39, while the change in position is represented by a velocity vector. These are combined to produce Equation 4.41. The world space position of the point *Q* in the object, taking into consideration the object's orientation, is given by rearranging Equation 4.40 to give $R(t)q = q(t) - x(t)$. Substituting this into Equation 4.41 and distributing the cross product produces Equation 4.42

$$q(t) = R(t)q + x(t) \qquad \text{(Eq. 4.40)}$$

$$\dot{q}(t) = \omega(t)^* R(t)q + v(t) \qquad \text{(Eq. 4.41)}$$

$$\dot{q}(t) = \omega(t) \times (q(t) - x(t)) + v(t) \qquad \text{(Eq. 4.42)}$$

## Center of Mass

The *center of mass* of an object is defined by the integration of the differential mass times its position in the object. In computer graphics, the mass distribution of an object is typically modeled by individual points, which is usually implemented by assigning a mass value to each of the object's vertices. If the individual masses are given by $m_i$, then the total mass of the object is represented by Equation 4.43. Using an object coordinate system that is located at the center of mass is often useful when modeling the dynamics of a rigid object. For the current discussion, it is assumed that $x(t)$ is the center of mass of an object. If the location of each mass point in world space is given by $q_i(t)$, then the center of mass is represented by Equation 4.44.

$$M = \Sigma m_i \qquad \text{(Eq. 4.43)}$$

$$x(t) = \frac{\Sigma m_i q_i(t)}{M} \qquad \text{(Eq. 4.44)}$$

## Forces

A linear force (a force along a straight line), *F*, applied to a mass, *m,* gives rise to a linear acceleration, *a,* by means of the relationship shown in Equation 4.45 and Equation 4.46. This fact provides a way to calculate acceleration from the application of forces. Examples of such forces are gravity, viscosity, friction, impulse forces due to collisions, and forces due to spring attachments. See Appendix B for the basic equations from physics that give rise to such forces.

$$F = m \cdot a \qquad \text{(Eq. 4.45)}$$

$$a = F/m \qquad \text{(Eq. 4.46)}$$

The various forces acting on a point can be summed to form the total external force, $F(t)$ (Equation 4.47). Given the mass of the point, the acceleration due to the total external force can be calculated and then used to modify the velocity of the point. This can be done at each time step. If the point is assumed to be part of a rigid object, then the point's location on the object must be taken into consideration, and the effect of the force on the point has an impact on the object as a whole. The rotational equivalent of linear force is *torque.* The torque that arises from the application of forces acting on a point of an object is given by Equation 4.48.

$$F(t) = \Sigma f_i(t) \qquad \text{(Eq. 4.47)}$$

$$\tau_i(t) = (q(t) - x(t)) \times f_i(t)$$
$$\tau(t) = \Sigma \tau_i(t) \qquad \text{(Eq. 4.48)}$$

## Momentum

As with force, the momentum (mass times velocity) of an object is decomposed into a linear component and an angular component. The object's local coordinate system is assumed to be located at its center of mass. The linear component acts on this center of mass, and the angular component is with respect to this center. *Linear momentum* and *angular momentum* need to be updated for interacting objects because these values are conserved in a closed system. Saying that the linear momentum is conserved in a closed system, for example, means that the sum of the linear momentum does not change if there are no outside influences on the system. The case is similar for angular momentum. That they are conserved means they can be used to solve for unknown values in the system, such as linear velocity and angular velocity.

*Linear momentum* is equal to velocity times mass (Equation 4.49). The total linear momentum P(t) of a rigid body is the sum of the linear momentums of each particle (Equation 4.50). For a coordinate system whose origin coincides with the center of mass, Equation 4.50 simplifies to the mass of the object times its velocity (Equation 4.51). Further, since the mass of the object remains constant, taking the derivative of momentum with respect to time establishes a relationship between linear momentum and linear force (Equation 4.52). This states that the force acting on a body is equal to the change in momentum. Interactions composed of equal but opposite forces result in no change in momentum (i.e., momentum is conserved).

$$p = m \cdot v \tag{Eq. 4.49}$$

$$P(t) = \Sigma m_i \dot{q}_i(t) \tag{Eq. 4.50}$$

$$P(t) = M \cdot v(t) \tag{Eq. 4.51}$$

$$\dot{P}(t) = M \cdot \dot{v}(t) = F(t) \tag{Eq. 4.52}$$

*Angular momentum* is a measure of the rotating mass weighted by the mass's distance from the axis of rotation. For a mass point in an object, the angular momentum is computed by taking the cross product of a vector to the mass and a velocity vector of that mass point times the mass of the point. These vectors are relative to the center of mass of the object. The total angular momentum of a rigid body is computed by integrating this equation over the entire object. For the purposes of computer animation, the computation is usually summed over mass points that make up the object (Equation 4.53). Notice that angular momentum is not directly dependent on the linear components of the object's motion.

$$\begin{aligned} L(t) &= \Sigma((q(t) - x(t)) \times m_i \cdot (\dot{q}(t) - v(t))) \\ &= \Sigma(R(t)q \times m_i \cdot (\omega(t) \times (q(t) - x(t)))) \\ &= \Sigma(m_i \cdot (R(t)q \times (\omega(t) \times R(t)q))) \end{aligned} \tag{Eq. 4.53}$$

In a manner similar to the relation between linear force and the change in linear momentum, torque equals the change in angular momentum (Equation 4.54). If no torque is acting on an object, then angular momentum is constant. However, the angular velocity of an object does not necessarily remain constant even in the case of no torque. The angular velocity can change if the distribution of mass of an object changes, such as when an ice skater spinning on his skates pulls his arms in toward his body to spin faster. This action brings the mass of the skater closer to the center of mass. Angular momentum is a function of angular velocity, mass, and the distance the mass is from the center of mass. To maintain a constant angular momentum, the angular velocity must increase if the distance of the mass decreases.

$$\dot{L}(t) = \tau(t) \tag{Eq. 4.54}$$

### Inertia Tensor

Angular momentum is related to angular velocity in much the same way that linear momentum is related to linear velocity, $P(t) = M \cdot v(t)$. See Equation 4.55. However, in the case of angular momentum, a matrix is needed to describe the distribution of mass of the object in space, the *inertia tensor*, $I(t)$. The inertia tensor is a symmetric 3x3 matrix. The initial inertia tensor defined for the untransformed object is denoted as $I_{\text{object}}$ (Equation 4.56). Terms of the matrix are calculated by integrating over the object (e.g., Equation 4.57). In Equation 4.57, the density of an object point, $q = (q_x, q_y, q_z)$, is $\rho$. For the discrete case, the equations are given by Equation 4.58. In a center-of-mass-centered object space, the inertia tensor for a transformed object depends on the orientation, $R(t)$, of the object but not on its position in space. Thus it is dependent on time. It can be transformed according to the transformation of the object by $I(t) = R(t) I_{\text{object}} R(t)^T$. Its inverse is transformed in the same way.

$$L(t) = I(t) \cdot \omega(t) \tag{Eq. 4.55}$$

$$I_{\text{object}} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \tag{Eq. 4.56}$$

$$I_{xx} = \iiint \rho(q) \cdot (q_y^2 + q_z^2) \mathrm{dxdydz} \tag{Eq. 4.57}$$

$$\begin{aligned} I_{xx} &= \Sigma m_i \cdot (y_i^2 + z_i^2) & I_{xy} &= \Sigma m_i \cdot x_i \cdot y_i \\ I_{yy} &= \Sigma m_i \cdot (x_i^2 + z_i^2) & I_{xz} &= \Sigma m_i \cdot x_i \cdot z_i \\ I_{zz} &= \Sigma m_i \cdot (x_i^2 + y_i^2) & I_{yz} &= \Sigma m_i \cdot y_i \cdot z_i \end{aligned} \tag{Eq. 4.58}$$

## The Equations

The state of an object can be kept in a vector, $S(t)$, consisting of its position, orientation, linear momentum, and angular momentum (Equation 4.59). Object attributes, which do not change over time, include its mass, $M$, and its object-space inertia tensor, $I_{\text{object}}$. At any time, an object's time-varying inertia tensor, angular velocity, and linear velocity can be computed (Equation 4.60–Equation 4.62). The time derivative of the object's state vector can now be formed (Equation 4.63).

$$S(t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} \tag{Eq. 4.59}$$

$$I(t) = R(t) I_{\text{object}} R(t)^T \tag{Eq. 4.60}$$

$$\omega(t) = I(t)^{-1} L(t) \tag{Eq. 4.61}$$

$$v(t) = \frac{P(t)}{M} \tag{Eq. 4.62}$$

$$\frac{d}{dt} S(t) = \frac{d}{dt} \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ \omega(t)^* R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \tag{Eq. 4.63}$$

This is enough information to run a simulation. Once the ability to compute the derivative information is available, then a differential equation solver can be used to update the state vector. In the simplest implementation, Euler's method can be used to update the values of the state array. The values of the state array are updated by multiplying their time derivatives by the length of the time step. In practice, Runge-Kutta methods (especially fourth-order) have found popularity because of their trade-off in speed, accuracy, and ease of implementation.

Care must be taken in updating the orientation of an object. If the derivative information above is used to update the orientation rotation matrix, then the columns of this matrix can quickly become nonorthogonal and not of unit length. At the very least, the column lengths should be renormalized after being updated. A better method is to update the orientation matrix by applying to its columns the axis-angle rotation implied by the angular velocity vector, $\omega(t)$. The magnitude of the angular velocity vector represents the angle of rotation about the axis along the angular velocity vector (see Appendix B for the axis-angle calculation). Alternatively, quaternions can be used to represent both the object's orientation and the orientation derivative, and its use here is functionally equivalent to the axis-angle approach.

## 4.3.2 Bodies in Contact

When an object starts to move in any kind of environment other than a complete void, chances are that sooner or later it will bump into something. If nothing is done about this in a computer animation, the object will penetrate and then pass through other objects. Other types of contact include objects sliding against and resting on each other. All of these types of contact require the calculation of forces in order to accurately simulate the reaction of one object to another.

### Colliding Bodies

As objects move relative to one another, there are two issues that must be addressed: (1) detecting the occurrence of collision and (2) computing the appropriate response to those collisions. The former is strictly a kinematic issue in that it has to do with the positions and orientations of objects and how they change over time. The latter is usually a dynamic issue in that forces that are a result of the collision are computed and used to produce new motions for the objects involved.

Collision detection considers the movement of one object relative to another. In its most basic form, testing for a collision amounts to determining whether there is intersection in the static position of two objects at a specific instance in time. In a more sophisticated form, the movement of one object relative to the other object during a finite time interval is tested for overlap. These computations can become quite involved when dealing with complex geometries.

Collision response is a consideration in physically based simulation. The geometric extent of the object is not of concern but rather the distribution of its mass. Localized forces at specific points on the object impart linear and rotational forces onto the other objects involved.

In dealing with the time of collision, there are two options. The first is to proceed as best one can from this point in time by calculating an appropriate reaction to the current situation by the particle involved in the collision (the penalty method). This option allows penetration of the particle before the collision reaction takes place. Of course, if the particle is moving rapidly, this penetration might be visually significant. If multiple collisions occur in a time interval, they are treated as occurring simultaneously even though handling them in their correct sequential order may have produced different results. While more inaccurate than the second option, this is simpler to implement and often gives acceptable results.

The second option is to back up time $t_i$ to the first instant that a collision occurred and determine the appropriate response at the time of collision. If multiple collisions occur in a time interval, then time is backed up to the point at which the first collision took place. In complex environments in which collisions happen at a high rate, this constant backing up of time and recomputing the motion of objects can become quite time-consuming.

There are three common options for collision response: a strictly kinematic response, the penalty method, and the calculation of an impulse force. The kinematic response is quick and easy. It produces good visual results for particles and spherically shaped objects. The penalty method introduces a temporary, nonphysically based force in order to restore nonpenetration. It is typically used when response to the collision occurs at the time step when penetration is detected (as opposed to backing up time). The advantage of this technique is that it is easy to compute and the force is easily incorporated into the computational mechanism that simulates rigid body movement. Calculating the impulse force is a more precise way of introducing a force into the system and is typically used when time is backed up to the point of first contact. Detecting collisions and reacting to them are discussed next.

### Particle-Plane Collision and Kinematic Response

One of the simplest illustrative situations to consider for collision detection and response is that of a particle traveling at a constant velocity toward a stationary plane at an arbitrary angle (see Figure 4.31). The task is to detect when the particle collides with the plane and have it bounce off the plane. Because a simple plane is involved, its planar equation can be used (Equation 4.64). $E(p)$ is the planar equation, $a,b,c,d$ are coefficients of the planar equation, and $p$ is a particle that is only a position in space and has no physical extent of its own. Points on the plane satisfy the planar equation, that is, $E(p) = 0$. The planar equation is formed so that for

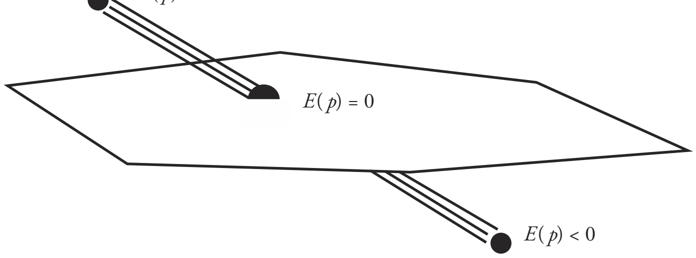


**Figure 4.31** Point-plane collision

points in front of the plane the planar equation evaluates to a positive value, $E(p) > 0$; for points behind the plane, $E(p) < 0$.

$$E(p) = a \cdot x + b \cdot y + c \cdot z + d \qquad \text{(Eq. 4.64)}$$

The particle travels toward the plane as its position is updated according to its average velocity over the time interval, as in Equation 4.65. At each time step $t_i$, the particle is tested to see if it is still in front of the plane, $E(p(t_i)) > 0$. As long as this evaluates to a positive value, there is no collision. The first time $t$ at which $E(p(t_i)) \leq 0$ indicates that the particle has collided with the plane at some time between $t_{i-1}$ and $t_i$. What to do now? The collision has already occurred and something has to be done about it.

$$p(t_i) = p(t_{i-1}) + \partial t \cdot v_{\text{ave}}(t) \qquad \text{(Eq. 4.65)}$$

In the kinematic response, when penetration is detected, the component of the velocity vector of the particle that is parallel to the surface normal is simply negated by subtracting it out of the velocity vector and then subtracting it out again. To reduce the height of each successive bounce, a damping factor, $0 < k < 1$, can be applied when subtracting it out the second time (Equation 4.66). This bounces the particle off a surface at a reduced velocity (Figure 4.32). This approach is not physically based, but it produces reasonable visuals, especially for particles and spherically shaped objects. Taking the example from Section 4.3.1 and incorporating kinematic response to collisions with the ground produces Figure 4.33.

$$\begin{aligned} v(t_{i+1}) &= v(t_i) - v(t_i) \cdot N - k \cdot v(t_i) \cdot N \\ &= v(t_i) - (1 + k) \cdot v(t_i) \cdot N \end{aligned} \qquad \text{(Eq. 4.66)}$$

### The Penalty Method

When objects penetrate due to temporal sampling, a simple method of constructing a reaction to the implied collision is the *penalty method.* As the name suggests, a point is penalized for penetrating another object. In this case, a spring, with a zero rest length, is momentarily attached from the offending point to the surface it penetrated in such a way so as to impart a restoring force on the offending point. For now, it is assumed that the surface it penetrates is immovable and thus does not have to be considered as far as collision response is concerned. The closest point on the penetrated surface to the penetrating point is used as the point of attachment. See Figure 4.34. The spring, therefore, imparts a force on the point in the direction of the penetrated surface normal and with a magnitude according to Hooke's law ($F = -k \cdot d$). A mass assigned to the point is used to compute a resultant acceleration ($a = F/m$), which contributes an upward velocity to the point. When this upward velocity is combined with the point's original motion, the

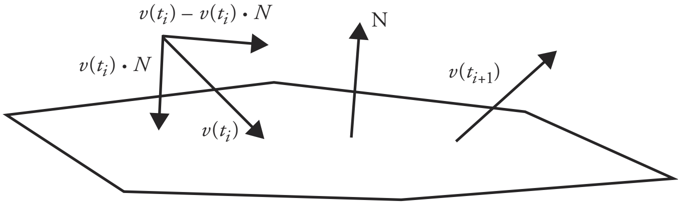


**Figure 4.32** Kinematic solution for collision reaction

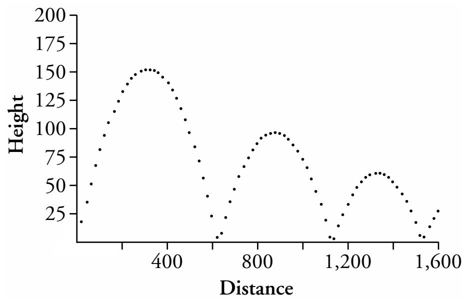


**Figure 4.33** Kinematic response to collisions with ground using 0.8 as the damping factor for the example from Section 4.3.1

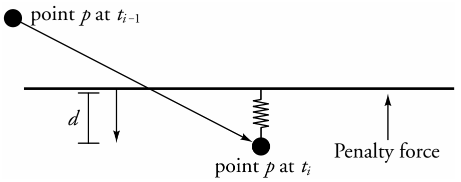


**Figure 4.34** Penalty spring

point's downward motion will be stopped and reversed by the spring, while any component of its motion tangent to the surface will be unaffected. While easy to implement, this approach is not ideal. An arbitrary mass ($m$) must be assigned to the point, and an arbitrary constant ($k$) must be determined for the spring. It is difficult to control because if the spring constant is too weak or if the mass is too large, then the collision will not be corrected immediately. If the spring constant is too strong or the mass is too small, then the colliding surfaces will be thrown apart in an unrealistic manner. If the point is moving fast relative to the surface it penetrated, then it may take a few time steps for the spring to take effect and restore nonpenetration.

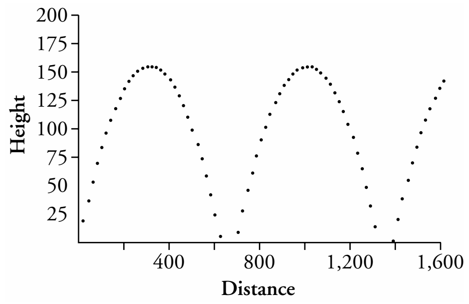


**Figure 4.35** Penalty method with a spring constant of 250 and a point mass of 10 for example from Section 4.3.1

Using the example from Section 4.3.1 and implementing the penalty method produces the motion traced in Figure 4.35. In this implementation, a temporary spring is introduced into the system whenever both the $y$ component of the position and the $y$ component of the velocity vector are negative; a spring constant of 250 and a point mass of 10 are used. While the spring force is easy to incorporate with other system forces such as gravity, the penalty method requires the user to specify fairly arbitrary control parameters that typically require a trial-and-error process to determine appropriate values.

When used with polyhedra objects, the penalty force can give rise to torque when not acting in line with the center of mass of an object. When two polyhedra collide, the spring is attached to both objects and imparts an equal but opposite force on the two to restore nonpenetration. Detecting collisions among polyhedra is discussed next, followed by a more accurate method of computing the impulse forces due to such collisions.

### Testing Planar Polyhedra

In environments in which objects are modeled as planar polyhedra, at each time step each polyhedron is tested for possible penetration against every other polyhedron. A collision is implied whenever the environment transitions from a nonpenetration state to a state in which a penetration is detected. A test for penetration at each time step can miss some collisions because of the discrete temporal sampling, but it is sufficient for many applications.

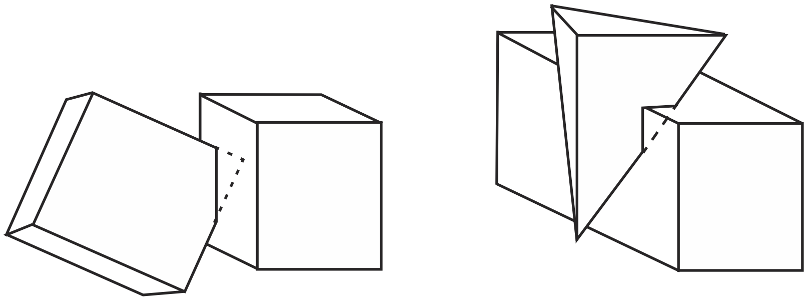
Various tests can be used to determine whether an overlap condition exists between polyhedra and the tests should be chosen according to computational efficiency and generality. Bounding box tests, also known as min-max tests, can be used to quickly determine if there is any chance for an intersection. A bounding box can easily be constructed by searching for the minimum and maximum values in $x$, $y$, and $z$. Bounding boxes can be tested for overlap. If there is no overlap of the bounding boxes, then there can be no overlap of the objects. If the object's shape does not match a rectangle well, then a bounding sphere or bounding slabs (pairs of parallel planes at various orientations that bound the object) can be used. A hierarchy of bounding shapes can be used on object groups, individual objects, and individual faces. If these bounding tests fail to conclusively establish that a nonoverlap situation exists, then more elaborate tests must be conducted.

Most collisions can be detected by testing each of the vertices of one object to see if any are inside another polyhedron and then testing each of the vertices of the other object to see if any are inside the first. To test whether a point is inside a convex polyhedron, each of the planar equations for the faces of the object is evaluated using the point's coordinates. If the planes are oriented so that a point to the outside of the object evaluates to a positive value and a point to the inside of the object evaluates to a negative value, then the point under consideration has to evaluate to a negative value in all of the planar equations in order to be declared inside the polyhedron.

Testing a point to determine if it is inside a concave polyhedron is more difficult. One way to do it is to construct a semi-infinite ray emanating from the point under consideration in any particular direction. For example, $y = P_y$, $z = P_z$, $x > P_x$ defines a line parallel to the *x*-axis going in the positive direction from the point *P.* This semi-infinite ray is used to test for intersection with all of the faces of the object, and the intersections are counted; an odd number of intersections means that the point is inside the object, and an even number means that the point is outside. To intersect a ray with a face, the ray is intersected with the planar equation of the face and then the point of intersection is tested to see if it is inside the polygonal face.

Care must be taken in correctly counting intersections if this semi-infinite ray intersects the object exactly at an edge or vertex or if the ray is colinear with an edge of the object. (These cases are similar to the situations that occur when scan-converting concave polygons.) While this does not occur very often, it should be dealt with robustly. Because the number of intersections can be difficult to resolve in such situations, it can be computationally more efficient to simply choose a semi-infinite ray in a different direction and start over with the test.

One can miss some intersection situations if performing only these tests. The edges of each object must be tested for intersection with the faces of the other object and vice versa to capture other overlap situations. The edge-face intersection test consists of two steps. In the first step, the edge is tested for intersection with the plane of the face (i.e., the edge vertices are on opposite sides of the plane of the face) and the point of intersection is computed if it exists. In the second step, the point of intersection is tested for 2D containment inside the face. The edge-face intersection tests would actually capture all of the penetration situations, but because it is the more expensive of the two tests, it is usually better to perform the vertex tests first. See Figure 4.36.



Vertex inside a polyhedron

Object penetration without a vertex of one object contained in the other

**Figure 4.36** Detecting polyhedra intersections

Often, a normal that defines the plane of intersection is used in the collision response calculations. For collisions that are detected by the penetration tests, one of two collision situations can usually be established by looking at the relative motion of the objects. Either a vertex has just penetrated a face, or an edge from one object has just intersected an edge from the other object. In either case, the plane of intersection can be established. When a vertex has penetrated a face, a normal to the face is used. When there is an edge-edge intersection, the normal is defined as the cross product of the two edges.

In certain limited cases, a more precise determination of collision can be performed. For example, if the movement of one object with respect to another is a linear path, then the volume swept by one object can be determined and intersected with the other object, whose position is assumed to be static relative to the moving object. If the direction of travel is indicated by a vector, *V,* then front faces and back faces with respect to the direction of travel can be determined. The boundary edges that share a front face and a back face can be identified. The volume swept by the object is defined by the original positions of the back faces, the translated positions of the front faces, and faces that are the boundary edges extruded in the direction of travel (Figure 4.37). This volume is intersected with the static object.

## Impulse Force of Collision

To allow for more accurate handling of the moment of collision, time can be "backed up" to the point of impact, the reaction can be computed, and the time can be moved forward again. While more accurate in its reconstruction of events, this approach can become very computationally intense in complex environments if many closely spaced collisions occur. However, for a point-plane collision, the impulse can be simplified because the angular velocity can be ignored.

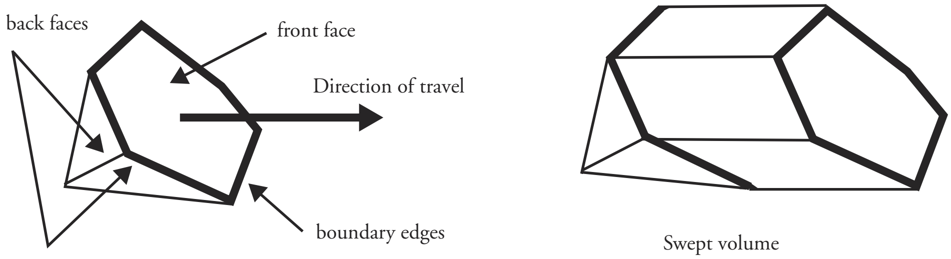


**Figure 4.37** Swept volume

The actual time of collision within a time interval can be searched for by using a binary search strategy. If it is known that a collision occurred between $t_{i-1}$ and $t_i$, these times are used to initialize the lower ($L = t_{i-1}$) and upper ($U = t_i$) bounds on the time interval to be tested. At each iteration of the search, the point's position is tested at the middle of the bounded interval. If the test indicates that the point has not collided yet, then the upper bound is replaced with the middle of the time interval. Otherwise, the lower bound is replaced with the middle of the time interval. This test repeats with the middle of the new time interval and continues until some desired tolerance is attained. The tolerance can be based either on the length of the time interval or on the range of distances the point could be on either side of the surface.

Alternatively, a linear-path, constant-velocity approximation can be used over the time interval to simplify the calculations. The position of the point before penetration and the position after penetration can be used to define a linear path. The point of intersection along the line with a planar surface can be calculated analytically. The relative position of the intersection point along this line can be used to estimate the precise moment of impact (Figure 4.38).

At the time of impact, the normal component of the point's velocity can be modified as the response to the collision. A scalar, the coefficient of restitution,

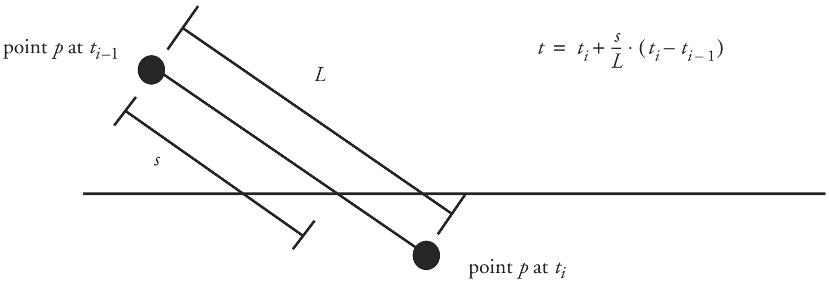
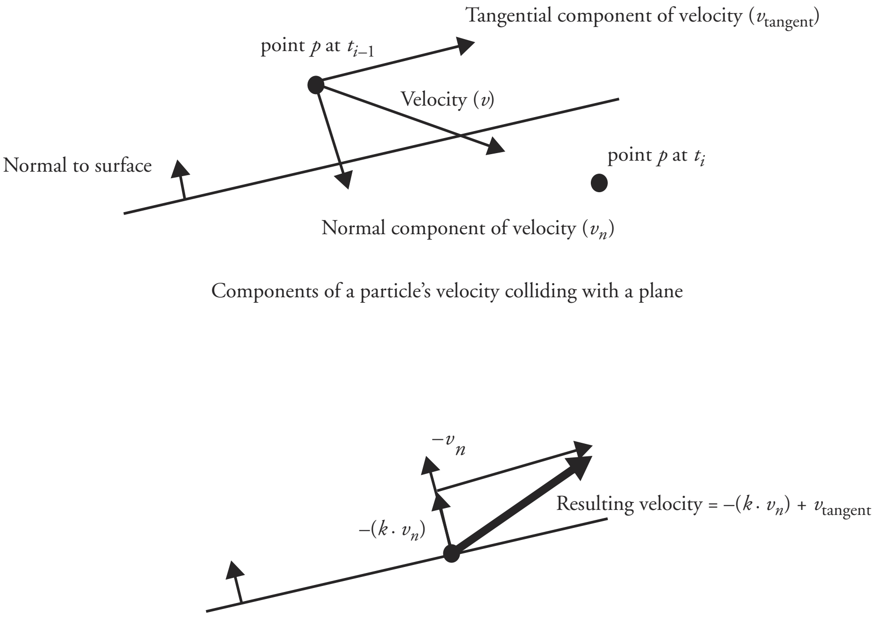


**Figure 4.38** Linearly estimating time of impact, $t$


Tangential component of velocity ($v_{\text{tangent}}$)
point $p$ at $t_{i-1}$
Velocity ($v$)
Normal to surface
point $p$ at $t_i$
Normal component of velocity ($v_n$)
$-v_n$
Resulting velocity = $-(k \cdot v_n) + v_{\text{tangent}}$
$-(k \cdot v_n)$

Components of a particle's velocity colliding with a plane

Computing velocity resulting from a collision ($k$ is the coefficient of restitution)

**Figure 4.39** Impact response of a point with a plane

between zero and one can be applied to the resulting velocity to model the degree of elasticity of the collision (Figure 4.39).

## Computing Impulse Forces

Once a collision is detected and the simulation has been backed up to the point of intersection, the reaction to the collision can be calculated. By working back from the desired change in velocity due to a collision, an equation that computes the required change in momentum is formed. This equation uses a new term, called *impulse,* expressed in units of momentum. The impulse, *J,* can be viewed as a large force acting over a short time interval (Equation 4.67). The change in momentum *P,* and therefore the change in velocity, can be computed once the impulse is known (Equation 4.68).

$$J = F \cdot \Delta t \tag{Eq. 4.67}$$

$$J = F \cdot \Delta t = M \cdot a \cdot \Delta t = M \cdot \Delta v = \Delta(M \cdot v) = \Delta P \tag{Eq. 4.68}$$

To characterize the elasticity of the collision response, the user selects the coefficient of restitution, $0 \leq \varepsilon \leq 1$, which relates the relative velocity before the collision to the relative velocity after the collision in the direction normal to the plane of intersection (Equation 4.69). To compute *J,* the equations of how the velocities must change based on the equations of motion before and after the collision are used. These equations use the impulse, *j,* and can then be used to solve for its value.

$$v_{\mathrm{rel}}^{+} = -\varepsilon \cdot v_{\mathrm{rel}}^{-} \tag{Eq. 4.69}$$

Assume that the collision of two objects, *A* and *B,* has been detected at time *t.* Each object has a position for its center of mass ($x_A(t)$, $x_B(t)$), linear velocity ($v_A(t)$, $v_B(t)$), and angular velocity ($\omega_A(t)$, $\omega_B(t)$). The points of collision ($p_A$, $p_B$) have been identified on each object. See Figure 4.40.

At the point of intersection, the normal to the surface of contact, *n,* is determined depending on whether it is a vertex-face contact or an edge-edge contact. The relative positions of the contact points with respect to the center of masses are $r_A$ and $r_B$ respectively (Equation 4.70). The relative velocity of the contact points of the two objects in the direction of the normal to the surface is computed by Equation 4.71. The velocities of the points of contact are computed as in Equation 4.72.

$$\begin{aligned} r_A &= p_A - x_A(t) \\ r_B &= p_B - x_B(t) \end{aligned} \tag{Eq. 4.70}$$

$$v_{\mathrm{rel}} = (\dot{p}_A(t) - \dot{p}_A(t)) \cdot n \tag{Eq. 4.71}$$

$$\begin{aligned} \dot{p}_A(t) &= v_A(t) + \omega_A(t) \times r_A \\ \dot{p}_B(t) &= v_B(t) + \omega_B(t) \times r_B \end{aligned} \tag{Eq. 4.72}$$

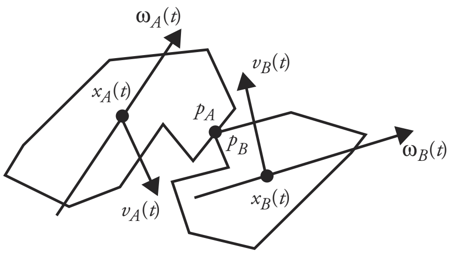


**Figure 4.40** Configuration of colliding objects

The linear and angular velocities of the objects before the collision ($v_A^-$, $v_B^-$) are updated by the impulse to form the linear and angular velocities after the collision ($v_A^+$, $v_B^+$). Impulse is a vector quantity in the direction of the normal to the surface of contact, $J = j \cdot n$. The linear velocities are updated by adding in the effect of the impulse scaled by one over the mass (Equation 4.73). The angular velocities are updated by computing the effect of the impulse on the angular velocity of the objects (Equation 4.74).

$$\begin{aligned} v_A^+ &= v_A^- + \frac{j \cdot n}{M_A} \\ v_B^+ &= v_B^- + \frac{j \cdot n}{M_B} \end{aligned} \qquad \text{(Eq. 4.73)}$$

$$\begin{aligned} \omega_A^+ &= \omega_A^- + I_A^{-1}(t) \cdot (r_A \times j \cdot n) \\ \omega_B^+ &= \omega_B^- + I_B^{-1}(t) \cdot (r_B \times j \cdot n) \end{aligned} \qquad \text{(Eq. 4.74)}$$

To solve for the impulse, form the difference between the velocities of the contact points after collision in the direction of the normal to the surface of collision (Equation 4.75). The version of Equation 4.72 for velocities after collision is substituted into Equation 4.75. Equation 4.73 and Equation 4.74 are then substituted into that to produce Equation 4.76. Finally, substituting into Equation 4.69 and solving for $j$ produces Equation 4.77.

$$v_{\text{rel}}^+ = n \cdot (\dot{p}_A^+(t) - \dot{p}_B^+(t)) \qquad \text{(Eq. 4.75)}$$

$$v_{\text{rel}}^+ = n \cdot (v_A^+(t) + \omega_A^+(t) \times r_A - v_B^+(t) + \omega_B^+(t) \times r_B) \qquad \text{(Eq. 4.76)}$$

$$j = \frac{-((1 + \varepsilon) \cdot v_{\text{rel}}^-)}{\frac{1}{M_A} + \frac{1}{M_B} + n \cdot (I_A^{-1}(t)(r_A \times n)) \times r_A + (I_B^{-1}(t)(r_B \times n)) \times r_B} \qquad \text{(Eq. 4.77)}$$

Contact between two objects is defined by the point on each object involved in the contact and the normal to the surface of contact. A point of contact is tested to see if an actual collision is taking place. When collision is tested for, the velocity of the contact point on each object is calculated. A collision is detected if the component of the relative velocity of the two points in the direction of the normal indicates the contact points are approaching each other.

If there is a collision, then Equation 4.77 is used to compute the magnitude of the impulse. The impulse is used to scale the contact normal, which can then be used to update the linear and angular momenta of each object.

```
Compute VA, VB                      ;EQ 121
Compute Vrelative = dot(N,(VA-VB)   ;relative velocity of two contact
                                     points
if Vrelative > threshold
              compute j             ;EQ 126
              J = j*n
              PA += J               ;update linear momentum
              PB -= J               ;update linear momentum
              LA += rA x J          ;update angular momentum
              LB -= rB x J          ;update angular momentum
else if Vrelative < -threshold
              resting contact
else
              objects are moving away from each other
```

If there is more than one point of contact between two objects, then each must be tested for collision. Each time a collision point is identified, it is processed for updating the momentum as above. If any collision is processed in the list of contact points, then after the momenta have been updated, the contact list must be traversed again to see if there exist any collision points with the new object momenta. Each time one or more collisions are detected and processed in the list, the list must be traversed again; this is repeated until it can be traversed and no collisions are detected.

### Friction

An object resting on the surface of another object (the *supporting object*) is referred to as being in a state of *resting contact* with respect to that supporting object. In such a case, any force acting on the first object is decomposed into a component perpendicular to the normal of the contact surface and a component parallel to the normal of the contact surface (the *normal force*) (Figure 4.41). If the direction of the parallel component is toward the supporting object, and if the supporting object is considered immovable, such as the ground or a table, the parallel component is immediately and completely canceled by a force equal and opposite in direction that is supplied by the supporting object. If the supporting object is movable, then the parallel component is applied transitively to the supporting object and added to the forces being applied to the supporting object. If the perpendicular component of force is directed away from the supporting object, then it is simply used to move the object up and away from the supporting object.

The parallel component of the force applied to the object is responsible for sliding (or rolling) the object along the surface of the supporting object. If the object is initially stationary with respect to the supporting object, then there is typically some threshold force, due to *static friction,* that must be exceeded before the object

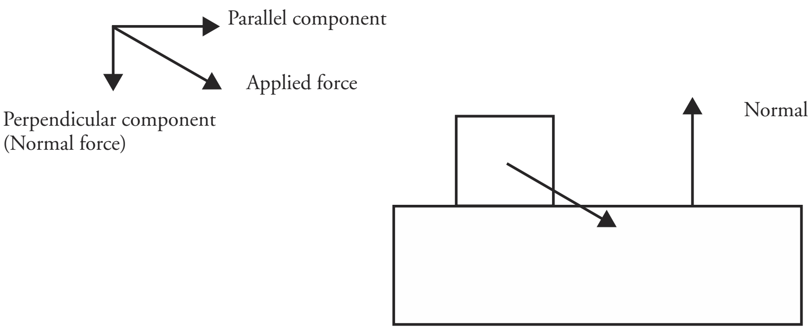


**Figure 4.41** Horizontal and vertical components of force

will start to move. This static friction force, $F_s$, is a function of the normal force, $F_N$ (Equation 4.78). The constant $\mu_s$ is the coefficient of static friction and is a function of the two surfaces in contact.

$$F_s = \mu_s \cdot F_N \tag{Eq. 4.78}$$

Once the object is moving along the surface of the supporting object, there is a *kinetic friction* that works opposite to the direction of travel. This friction creates a force, opposite to the direction of travel, which is a linear function of the normal force, *N,* on the object (Equation 4.79). The constant $\mu_k$ is the coefficient of kinetic friction and is a function of the two surfaces in contact.

$$F_k = \mu_k \cdot F_N \tag{Eq. 4.79}$$

## Resting Contact

Computing the forces involved in resting contact is one of the more difficult dynamics problems for computer animation. The exposition here follows that found in Baraff's work [1]. The solution requires access to quadratic programming, the implementation of which is beyond the scope of this book. An example situation in which several objects are resting on one another is shown in Figure 4.42.

For each contact point, there is a force normal to the surface of contact, just as in colliding contact. The objective is to find the magnitude of that force for a given configuration of objects. These forces (1) have to be strong enough to prevent interpenetration; (2) must only push objects apart, not together; and (3) have to go to zero at the point of contact at the moment that objects begin to separate.

To analyze what is happening at a point of contact, use a distance function, $d_i(t)$, that evaluates to the distance between the objects at the *i*th point of contact. Assuming objects *A* and *B* are involved in the *i*th contact and the points involved in the *i*th contact are $p_A$ and $p_B$ from the respective objects, then the distance function is given by Equation 4.80.

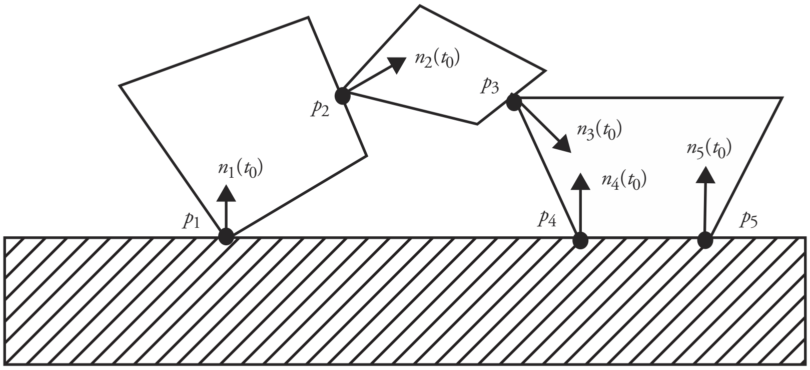


**Figure 4.42** Multiple-object resting contacts

$$d_i(t) = n_i(t) \cdot (p_A(t) - p_B(t)) \quad \textbf{(Eq. 4.80)}$$

If $d_i(t)$ is zero, then the objects involved are still in contact. Whenever $d_i(t) > 0$, the objects are separating. One of the objectives is to avoid $d_i(t) < 0$, which would indicate penetration.

At $t_0$, the distance between the objects is zero. To prevent object penetration from time $t_0$ onward, the relative velocity of the two objects must be greater or equal to zero, $\dot{d}_i(t) \geq 0$. The equation for relative velocity is produced by differentiating Equation 4.80 and is shown in Equation 4.81. At time $t_0$, the objects are touching, so $p_A(t_0) = p_B(t_0)$. This means that $\dot{d}_i(t_0) = n_i(t_0) \cdot (\dot{p}_A(t_0) - \dot{p}_B(t_0))$. In addition, for resting contact, $\dot{d}_i(t_0) = 0$.

$$\dot{d}_i(t) = \dot{n}_i(t) \cdot (p_A(t) - p_B(t)) + n_i(t) \cdot (\dot{p}_A(t) - \dot{p}_B(t)) \quad \textbf{(Eq. 4.81)}$$

Since $d_i(t_0) = \dot{d}_i(t_0) = 0$, penetration will be avoided if the second derivative is greater than or equal to zero, $\ddot{d}_i(t) \geq 0$. The second derivative is produced by differentiating Equation 4.81 as shown in Equation 4.82. At $t_0$, remembering that $p_A(t_0) = p_B(t_0)$, one finds that the second derivative simplifies as shown in Equation 4.83. Notice that Equation 4.83 further simplifies if the normal to the surface of contact does not change ($\dot{n}_i(t_0) = 0$).

$$\begin{aligned}\ddot{d}_i(t) = {} & \ddot{n}_i(t) \cdot (p_A(t) - p_B(t)) + \dot{n}_i(t) \cdot (\dot{p}_A(t) - \dot{p}_B(t)) \\ & + \dot{n}_i(t) \cdot (\dot{p}_A(t) - \dot{p}_B(t)) + n_i(t) \cdot (\ddot{p}_A(t) - \ddot{p}_B(t)) \quad \textbf{(Eq. 4.82)}\end{aligned}$$

$$\ddot{d}_i(t_0) = n_i(t_0) \cdot (\ddot{p}_A(t_0) - \ddot{p}_B(t_0)) + 2 \cdot \dot{n}_i(t_0) \cdot (\dot{p}_A(t_0) - \dot{p}_B(t_0)) \quad \textbf{(Eq. 4.83)}$$

The constraints on the forces as itemized at the beginning of this section can now be written as equations: the forces must prevent penetration (Equation 4.84);

the forces must push objects apart, not together (Equation 4.85); and either the objects are not separating or, if the objects are separating, then the contact force is zero (Equation 4.86).

$$\ddot{d}_i(t) \geq 0 \tag{Eq. 4.84}$$

$$f_i \geq 0 \tag{Eq. 4.85}$$

$$\ddot{d}_i(t) \cdot f_i = 0 \tag{Eq. 4.86}$$

The relative acceleration of the two objects at the $i$th contact point, $\ddot{d}_i(t_0)$, is written as a linear combination of all of the unknown $f_{ij}$s (Equation 4.87). For a given contact $j$, its effect on the relative acceleration of the bodies involved in the $i$th contact point needs to be determined. Referring to Equation 4.83, the component of the relative acceleration that is dependent on the velocities of the points, $2 \cdot \dot{n}_i(t_0) \cdot (\dot{p}_A(t_0) - \dot{p}_B(t_0))$, is not dependent on the force $f_j$ and is therefore part of the $b_i$ term. The component of the relative acceleration dependent on the accelerations of the points involved in the contact, $n_i(t_0) \cdot (\ddot{p}_A(t_0) - \ddot{p}_B(t_0))$, is dependent on $f_j$ if object $A$ or object $B$ is involved in the $j$th contact. The acceleration at a point on an object arising from a force can be determined by differentiating Equation 4.72, producing Equation 4.88 where $r_A(t) = p_A(t) - x_A(t)$.

$$\ddot{d}_i(t_0) = b_i + \sum_{j=1}^{n} (a_{ij} \cdot f_j) \tag{Eq. 4.87}$$

$$\ddot{p}_A(t) = \dot{v}_A(t) + \dot{\omega}_A(t) \times r_A + \omega_A(t) \times (\omega_A(t) \times r_A) \tag{Eq. 4.88}$$

In Equation 4.88, $\dot{v}_A(t)$ is the linear acceleration as a result of the total force acting on object $A$ divided by its mass. A force $f_j$ acting in direction $n_j(t_0)$ produces $f_j/m_A \cdot n_j(t_0)$. Angular acceleration, $\dot{\omega}_A(t)$, is formed by differentiating Equation 4.61 and produces Equation 4.89, in which the first term contains torque and therefore relates the force $f_j$ to $\ddot{p}_A(t)$ while the second term is independent of the force $f_j$ and so is incorporated into a constant term, $b_i$.

$$\dot{\omega}_A(t) = I_A^{-1}(t) \cdot \tau_A(t) + I_A^{-1}(t) \cdot (L_A(t) \times \omega_A(t)) \tag{Eq. 4.89}$$

The torque from force $f_j$ is $(p_j - x_A(t_0)) \times f_j \cdot n_j(t_0)$. The total dependence of $\ddot{p}_A(t)$ on $f_j$ is shown in Equation 4.90. Similarly, the dependence of $\ddot{p}_B(t)$ on $f_j$ can be computed. The results are combined as indicated by Equation 4.83 to form the term $a_{ij}$ as it appears in Equation 4.87.

$$f_j \cdot \left( \frac{n_j(t_0)}{m_A} + (I_A^{-1}(t_0) \cdot (p_j - x_A(t_0)) \times n_j(t_0)) \times r_A \right) \tag{Eq. 4.90}$$

Collecting the terms not dependent on an $f_j$ and incorporating a term based on the net external force, $F_A(t_0)$, and net external torque, $\tau_A(t_0)$ acting on $A$, the part of $\ddot{p}_A(t)$ independent of the $f_j$'s is shown in Equation 4.91. A similar expression is generated for $\ddot{p}_B(t)$. To compute $b_i$ in Equation 4.87, the constant parts of $\ddot{p}_A(t)$ and $\ddot{p}_B(t)$ are combined and dotted with $n_i(t_0)$. To this, the term $2 \cdot \dot{n}_i(t_0) \cdot (\dot{p}_A(t_0) - \dot{p}_B(t_0))$ is added.

$$\frac{F_A(t_0)}{m_A} + I_A^{-1}(t) \cdot \tau_A(t) + \omega_A(t) \times (\omega_A(t) \times r_A)$$

$$+ (I_A^{-1}(t) \cdot (L_A(t) \times \omega_A(t))) \times r_A \qquad \text{(Eq. 4.91)}$$

Equation 4.87 must be solved subject to the constraints in Equation 4.85, and Equation 4.86. These systems of equations are solved by *quadratic programming*. It is nontrivial to implement. Baraff [1] uses a package from Stanford [11] [13] [12] [14]. Baraff notes that the quadratic programming code can easily handle $\ddot{d}_i(t) = 0$ instead of $\ddot{d}_i(t) \geq 0$ in order to constrain two bodies to never separate. This enables the modeling of hinges and pin-joints.

## 4.4 Enforcing Soft and Hard Constraints

One of the main problems with using physically based animation is for the animator to get the object to do what he or she wants while at the same time have it react to the forces modeled in the environment. One way to solve this is to place constraints on the object that restrict some subset of the degrees of freedom of the object's motion. The remaining degrees of freedom are subject to the physically based animation system. Constraints are simply requirements placed on the object. For animation, constraints can take the form of co-locating points, maintaining a minimum distance between objects, or requiring an object to have a certain orientation in space. The problem for the animation system is in enforcing the constraints while the object's motion is controlled by some other method.

If the constraints are to be strictly enforced, then they are referred to as *hard constraints*. If the constraints are relations the system should only attempt to satisfy, then they are referred to as *soft constraints*. Satisfying hard constraints requires more sophisticated numerical approaches than satisfying soft constraints. To satisfy hard constraints, computations are made that search for a motion that reacts to forces in the system while at the same time satisfying all of the constraints. As more constraints are added to the system, this becomes an increasingly difficult problem. Soft constraints are typically incorporated into a system as additional forces that influence the final motion. One way to model flexible objects is to create soft distance constraints between the vertices of an object. These constraints, modeled by a mesh of interconnected springs and dampers, react to other forces in the system such as gravity and collisions to create a dynamic structure. Soft constraint satisfaction can also be phrased as an energy minimization problem in which deviation from the constraints increases the system's energy. Forces are introduced into the system that decrease the system's energy. These approaches are discussed below.

## 4.4.1 Flexible Objects

In Chapter 3, kinematic approaches to animating flexible bodies are discussed in which the animator is responsible for defining the source and target shapes as well as implying how the shapes are to be interpolated. Various physically based approaches have been proposed that model elastic and inelastic behavior, viscoelasticity, plasticity, and fracture (e.g., [15] [17] [20] [29] [30] [31]). Here, the simplest approach is presented. Flexibility is modeled by a spring-mass-damper system simulating the reaction of the body to external forces.

### Spring-Mass-Damper Modeling of Flexible Objects

Probably the most common approach to modeling flexible shapes is the spring-mass-damper model. The most straightforward strategy is to model each vertex of an object as a point mass and each edge of the object as a spring. Each spring's rest length is set equal to the original length of the edge. A mass can be arbitrarily assigned to an object by the animator and the mass evenly distributed among the object's vertices. If there is an uneven distribution of vertices throughout the object definition, then masses can be assigned to vertices in an attempt to more evenly distribute the mass. Spring constants are similarly arbitrary and are usually assigned uniformly throughout the object to some user-specified value.

As external forces are applied to specific vertices of the object, either because of collisions, gravity, wind, or explicitly scripted forces, vertices will be displaced relative to other vertices of the object. This displacement will induce spring forces, which will impart forces to the adjacent vertices as well as reactive forces back to the initial vertex. These forces will result in further displacements, which will induce more spring forces throughout the object, resulting in more displacements, and so on. The result will be an object that is wriggling and jiggling as a result of the forces propagating along the edge springs and producing constant relative displacements of the vertices.

One of the drawbacks with using springs to model the effects of external forces on objects is that the effect has to propagate through the object, one time step at a time. This means that the number of vertices used to model an object and the length of edges used have an effect on the object's reaction to forces. Because the vertices carry the mass of the object, using a different distribution of vertices to describe the same object will result in a difference in the way the object reacts to external forces.

### A Simple Example

In a simple two-dimensional example, an equilateral triangle composed of three vertices and three edges with uniform mass distribution and uniform spring constants will react to an initial external force applied to one of the vertices (Figure 4.43).

In the example, an external force is applied to vertex $V2$, pushing it generally toward the other two vertices. Assume that the force is momentary and is applied only for one time step during the animation. At the application of the force, an acceleration ($a_2 = F/m_2$) is imparted to vertex $V2$ by the force. The acceleration gives rise to a velocity at point $V2$, which in turn creates a change in its position. At the next time step, the external force has already been removed, but, because vertex $V2$ has been displaced, the lengths of edges $E12$ and $E23$ have been changed. As a result of this, a spring force is created along the two edges. The spring that models edge $E12$ imparts a restoring force to vertices $V1$ and $V2$, while the spring that models edge $E23$ imparts a restoring force to vertices $V2$ and $V3$. Notice that the springs push back against the movement of $V2$, trying to restore its position, while at the same time they are pushing the other two vertices away from $V2$. In a polygonal mesh, the springs associated with mesh edges propagate the effect of the force throughout the object.

The force, $F_{i,j}$, applied by the spring between vertex $i$ and vertex $j$ at vertex $j$ is based on Hooke's law and is given in Equation 4.92, in which $len_{i,j}$ is the rest length of the spring, $dist_{i,j}(t)$ is the distance between the two vertices, $k_s$ is the spring constant, and $v_{i,j}$ is the unit vector from vertex $i$ to vertex $j$.

$$F^{\text{spring}}_{i,j} = -F^{\text{spring}}_{j,i} = k_s \cdot (dist_{i,j}(t) - \text{len}_{i,j}) \cdot v_{i,j} \qquad \textbf{(Eq. 4.92)}$$

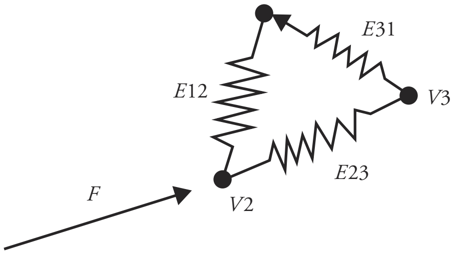


**Figure 4.43** A simple spring-mass model of a flexible object

Depending on the size of forces applied to spring-mass points, the size of the spring constant, and the size of the time step used to sample the system, a spring simulation may numerically explode, resulting in computed behavior in which the motions get larger and larger. This is because the force is assumed to be constant throughout the time step. A smaller time step can be used to control the simulation, or smaller spring constants can be assigned or larger masses used. But these modifications slow down the simulation. The size of a force can be clamped to some maximum, but this, too, can result in undesirable behavior. A common modification to better control the reaction of the springs is to incorporate dampers. A damper introduces an additional force in the spring that works against the velocity of the spring length, thus helping to limit the speed at which a spring changes length.

### Dampers

A damper imparts a force in the direction opposite to the velocity of the spring length and is proportional to that velocity. As a spring starts to change length faster and faster, the damper will help to control the change in length and keep it within some range that does not allow the simulation to explode. In Equation 4.93, $k_d$ is the damping coefficient and $v_i(t)$ is the velocity of the spring length. The dampers are incorporated into the spring models, and the damper force is proportional to the relative velocity of the endpoints of the spring-damper model (Figure 4.44).

$$F_i^{\text{damper}} = -k_d \cdot v_i(t) \qquad \text{(Eq. 4.93)}$$

Modeling only the object edges with spring dampers can result in a model that has more than one stable configuration. For example, if a cube's edges are modeled with springs, during applications of extreme external forces, the cube can turn inside out. Additional spring dampers can help to stabilize the shape of an object. Springs can be added across the object's faces and across its volume. To help prevent such undesirable, but stable, configurations in a cube, one or more springs that stretch diagonally from one vertex across the interior of the cube to the opposite vertex can be included in the model. These springs will help to model the internal material of a solid object (Figure 4.45).

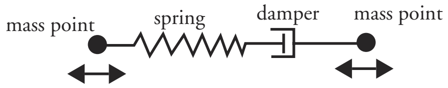


**Figure 4.44** Spring-damper configuration

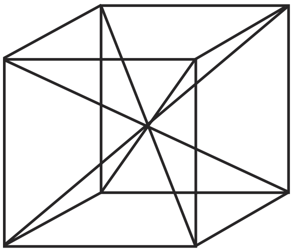


Figure 4.45 Interior springs to help stabilize the object's configuration

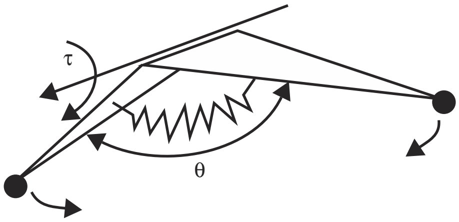


Figure 4.46 An angular spring imparting restoring torques

If specific angles between adjacent faces (dihedral angles) are desired, then angular springs (and dampers) can be applied. The spring resists deviation to the rest angle between faces and imparts a torque along the edge that attempts to restore the rest angle (Equation 4.94 and Figure 4.46). The damper works to limit the resulting motion. The torque is applied to the vertex from which the dihedral angle is measured; a torque of equal magnitude but opposite direction is applied to the other vertex.

$$\tau = k_s \cdot (\theta(t) - \theta_{\mathrm{rest}}) - k_d \cdot \dot{\theta}(t) \qquad \text{(Eq. 4.94)}$$

## 4.4.2 Virtual Springs

Virtual springs introduce forces into the system that do no directly model physical elements. These can be used to control the motion of objects in a variety of circumstances. The penalty method is an example of a virtual spring. Virtual springs with zero rest lengths can be used to constrain one object to lie on the surface of another, for example, or, with nonzero rest lengths, to maintain separation between moving objects.

Proportional derivative controllers (PDCs) are another type of virtual spring used to keep a control variable and its derivative within the neighborhood of desired values. For example, to maintain a joint angle and joint velocity close to desired values, the user can introduce a torque into a system, as in Equation 4.95. If the desired values are functions of time, PDCs are useful for biasing a model toward a given motion while still allowing it to react to system forces.

$$\tau = k_s \cdot (\theta(t) - \theta_{\mathrm{desired}}) - k_d \cdot (\dot{\theta}(t) - \dot{\theta}_{\mathrm{desired}}) \qquad \text{(Eq. 4.95)}$$

Springs, dampers, and PDCs are an easy and handy way to control the motion of objects by incorporating forces into a physically based modeling system. The springs can model physical elements in the system as well as represent arbitrary control forces. Dampers and PDCs are commonly used to keep system parameters close to desired values. They are not without problems, however. One drawback to using springs, dampers, and PDCs is that the user-supplied constants in the equations can be difficult to choose to obtain a desired effect. A drawback to using a spring mesh is that the effect of the forces must ripple through the mesh as the force propagates from one spring to the next. Still, they are often a useful and effective tool for the animator.

### 4.4.3 Energy Minimization

The concept of a system's energy can be used in a variety of ways to control the motion of objects. Used in this sense, energy is not defined solely as physically realizable but is free to be defined in whatever form serves the animator. Energy functions can be used to pin objects together, to restore the shape of objects, or to minimize the curvature in a spline as it interpolates points in space.

As presented by Witkin, Fleischer, and Barr [32], energy constraints induce restoring forces based on arbitrary functions of a model's parameters. The current state of a model can be expressed as a set of parameter values. These are external parameters such as position and orientation as well as internal parameters such as joint angles, the radius of a cylinder, or the threshold of an implicit function. They are any value of the object model subject to change. The set of state parameters will be referred to as $\Psi$.

A constraint is phrased in terms of a non-negative smooth function, $E(\Psi)$, of these parameters, and a local minimum of the constraint function is searched for in the space of all parametric values. The local minimum can be searched for by evaluating the gradient of the energy function and stepping in the direction of the negative of the gradient, $-\nabla E$. The gradient of a function is the parameter space vector in the direction of greatest increase of that function. Usually the parametric state set will have quite a few elements, but for illustrative purposes suppose there are only two state parameters. Taking the energy function to be the surface of a height function of those two parameters, one can consider the gradient as pointing uphill from any point on that surface. See Figure 4.47.

Given an initial set of parameters, $\Psi_0$, the parameter function at time zero is defined as $F(0) = \Psi_0$. From this it follows that $(d/dt)\ F(t) = -\nabla E$. The force vector in parameter space is the negative of the gradient of the energy function, $-\nabla E$. Any of a number of numerical techniques can be used to solve the equation, but a particularly simple one is Euler's method, as shown in Equation 4.96 (see Appendix B or [22] for better methods).

$$F(t_{i+1}) = F(t_i) - h \cdot \nabla E \qquad \text{(Eq. 4.96)}$$

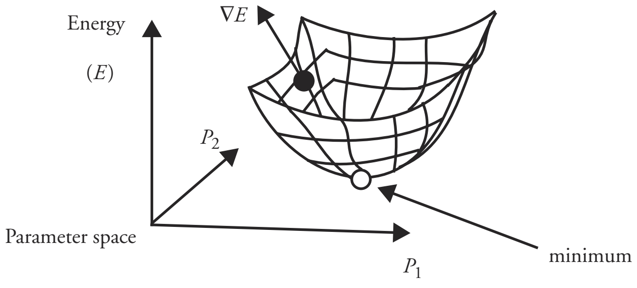


**Figure 4.47** Sample simple energy function

Determining the gradient, $\nabla E$, is usually done numerically by stepping along each dimension in parameter space, evaluating the energy function, and taking differences between the new evaluations and the original value.

### Three Useful Functions

As presented by Witkin, Fleischer, and Barr [32], three useful functions in defining an energy function are

The parametric position function, $P(u, v)$

The surface normal function, $N(u, v)$

The implicit function $I(x)$

The parametric position function and surface normal function are expressed as functions of surface parameters $u$ and $v$ (although for a given model, the surface parameters may be more complex, incorporating identifiers of object parts, for example). Given a value for $u$ and $v$, the position of that point in space and the normal to the surface at that point are given by the functions. The implicit function takes as its parameter a position in space and evaluates to an approximation of the signed distance to the surface where a point on the surface returns a zero, a point outside the object returns a positive distance to the closest point on the surface, and a point inside the object returns a negative distance to the closest point on the surface. These functions will, of course, also be functions of the current state of the model, $\Psi$.

### Useful Constraints

The functions above can be used to define several useful constraint functions. The methods of Witkin, Fleischer, and Barr can also lead to several intuitive constraint functions. The functions typically are associated with one or more user-specified weights; these are not shown.

**Point-to-Fixed-Point** The point $Q$ in space is fixed and is given in terms of absolute coordinate values. The point $P$ is a specific point on the surface of the model and is specified by the $u$, $v$ parameters. The energy function will be zero when these two points coincide.

$$E = |P(u, v) - Q|^2$$

**Point-to-Point** A point on the surface of one object is given and the point on the surface of a second object is given. The energy function will be zero when they coincide. Notice that within a constant, this is a zero-length spring. Also notice that the orientations of the objects are left unspecified. If this is the only constraint given, then the objects could completely overlap or partially penetrate each other to satisfy this constraint.

$$E = \left|P^q(u_a, v_a) - P^b(u_b, v_b)\right|^2$$

**Point-to-Point Locally Abutting** The energy function is zero when the points coincide, and the dot product of the normals at those points is equal to minus one (i.e., they are pointing away from each other).

$$E = \left|P^q(u_a, v_a) - P^b(u_b, v_b)\right|^2 + N^a(u_a, v_a) \cdot N^b(u_b, v_b) + 1.0$$

**Floating Attachment** With the use of the implicit function of object $b$, a specific point on object $a$ is made to lie on the surface of object $b$.

$$E = (I^b(P^q(u_a, v_a)))^2$$

**Floating Attachment Locally Abutting** A point of object $a$ is constrained to lie on the surface of $b$, using the implicit function of object $b$ as above. In addition, the normal of object $a$ and the point must be colinear to and in the opposite direction of the normal of object $b$ at the point of contact. The normal of object $b$ is computed as the gradient of its implicit function.

Other constraints are possible. Witkin, Fleischer, and Barr [32] present several others as well as some examples of animations using this technique.

$$E = (I^b(P^q(u_a, v_a)))^2 + N^a(u_a, v_a) \cdot \frac{\nabla I^b(P^q(u_a, v_a))}{\left|\nabla I^b(P^q(u_a, v_a))\right|} + 1.0$$

### Energy Constraints Are Not Hard Constraints

While such energy functions can be implemented quite easily and can be effectively used to assemble a structure defined by relationships such as the ones discussed above, a drawback to this approach is that the constraints used are not *hard constraints* in the sense that the constraint being imposed on the model is not always met. For example, if a point-to-point constraint is specified and one of the points is moved rapidly away, the other point, as moved by the constraint satisfaction system, will chase around the moving point. If the moving point comes to a halt, then the second point will, after a time, come to rest on top of the first point, thus satisfying the point-to-point constraint.

## 4.4.4 Space-Time Constraints

Space-time constraints view motion as a solution to a constrained optimization problem that takes place over time in space. Hard constraints which include equations of motion as well as nonpenetration constraints and locating the object at certain points in time and space are placed on the object. An objective function that is to be optimized is stated; for example, minimize the amount of force required to produce the motion over some time interval.

The material here is taken from Witkin and Kass [33]. Their introductory example will be used to illustrate the basic points of the method. Interested readers are urged to refer to that article as well as follow-up articles on space-time constraints (e.g., [5] [19]).

### Space-Time Particle

Consider a particle's position to be a function of time, $x(t)$. A time-varying force function, $f(t)$, is responsible for moving the particle. Its equation of motion is given in Equation 4.97.

$$m \cdot \ddot{x}(t) - f(t) - m \cdot g = 0 \tag{Eq. 4.97}$$

Given the function $f(t)$ and values for the particle's position and velocity, $x(t_0)$ and $\dot{x}(t_0)$, at some initial time, the position function, $x(t)$, can be obtained by integrating Equation 4.97 to solve the initial value problem.

However, the objective here is to determine the force function, $f(t)$. Initial and final positions for the particle are given as well as the times the particle must be at these positions (Equation 4.98). These equations along with Equation 4.97 are the constraints on the motion.

$$\begin{aligned} x(t_0) &= a \\ x(t_1) &= b \end{aligned} \tag{Eq. 4.98}$$

The function to be minimized is the fuel consumption, which here, for simplicity, is given as $|f|^2$. For a given time period $t_0 < t < t_1$, this results in Equation 4.99 as the function to be minimized subject to the time-space constraints and the motion equation constraint.

$$R = \int_{t_0}^{t_1} |f|^2 \mathrm{dt} \tag{Eq. 4.99}$$

In solving this, discrete representations of the functions $x(t)$ and $f(t)$ are considered. Time derivatives of $x$ are approximated by finite differences (Equation 4.100 and Equation 4.101) and substituted into Equation 4.97 to form $n$ physics constraints (Equation 4.102) and the two boundary constraints (Equation 4.103).

$$\dot{x}_i = \frac{x_i - x_{i-1}}{h} \tag{Eq. 4.100}$$

$$\ddot{x}_i = \frac{x_{i+1} - 2 \cdot x_i + x_{i-1}}{h^2} \tag{Eq. 4.101}$$

$$p_i = m \cdot \frac{x_{i+1} - 2 \cdot x_i + x_{i-1}}{h^2} - f - m \cdot g = 0 \tag{Eq. 4.102}$$

$$\begin{aligned} c_a &= x_1 - a = 0 \\ c_b &= x_n - b = 0 \end{aligned} \tag{Eq. 4.103}$$

If one assumes that the force function is constant between samples, the object function, $R$, becomes a sum of the discrete values of $f$. The discrete function $R$ is to be minimized subject to the discrete constraint functions, which are expressed in terms of the sample values, $x_i$ and $f_i$, that are to be solved for.

### Numerical Solution

The problem as stated fits into the generic form of constrained optimization problems, which is to "find the $S_j$ values that minimize $R$ subject to $C_i(S_j) = 0$." The $S_j$ values are the $x_i$ and $f_i$. Solution methods can be considered black boxes that request current values for the $S_j$ values, $R$, and the $C_i$ values as well as the derivatives of $R$ and the $C_i$ with respect to the $S_j$ as the solver iterates toward a solution.

The solution method used by Witkin and Kass [33] is a variant of sequential quadratic programming (SQP) [14]. The method computes a second-order Newton-Raphson step in $R$, which is taken irrespective of any constraints on the system. A first-order Newton-Raphson step is computed in the $C_i$ to reduce the constraint functions. The step to minimize the objective function, $R$, is projected onto the null space of the constraint step, that subspace in which the constraints are constant to a first-order approximation. Therefore, as steps are being taken to minimize the constraints, a particular direction for the step is chosen that does not affect the constraint minimization and that reduces the objective function.

Because it is first order in the constraint functions, the first derivative matrix (the Jacobian) of the constraint function must be computed (Equation 4.104).

Because it is second order in the objective function, the second derivative matrix (the Hessian) of the objective function must be computed (Equation 4.105). The first derivative vector of the objective function must also be computed, $\partial R / \partial S_j$.

$$J_{ij} = \frac{\partial C_i}{\partial S_j} \tag{Eq. 4.104}$$

$$H_{ij} = \frac{\partial^2 R}{\partial S_i \partial S_j} \tag{Eq. 4.105}$$

The second-order step to minimize R, irrespective of the constraints, is taken by solving the linear system of equations shown in Equation 4.106. A first-order step to drive the $C_j$'s to zero is taken by solving the linear system of equations shown in Equation 4.107. The final update is $\Delta S_j = \hat{S}_j + \tilde{S}_j$. The algorithm reaches a fixed point when the constraints are satisfied, and any further step that minimizes $R$ would violate one or more of the constraints.

$$-\frac{\partial}{\partial S_i}(R) = \sum_j H_{ij} \hat{S}_j \tag{Eq. 4.106}$$

$$-C_i = \sum_j J_{ij} (\hat{S}_j + \tilde{S}_j) \tag{Eq. 4.107}$$

Although one of the more complex methods presented here, space-time constraints are a powerful and useful tool for producing realistic motion while maintaining given constraints. They are particularly effective for creating the illusion of self-propelled objects whose movement is subject to user-supplied time constraints.

## 4.5 Controlling Groups of Objects

Managing complexity is one of the most important uses of the computer in animation, and nothing exemplifies that better than particle systems. A *particle system* is a large collection of individual elements, which, taken together, represent a conglomerate, fuzzy object. Both the behavior and the appearance of each individual particle are very simple. The individual particles typically behave according to simple physical principles with respect to their environment but not with respect to other particles of the system. When viewed together, the particles create the impression of a single, dynamic, complex object. This illusion of a greater whole is referred to as *emergent behavior* and is an identifying characteristic of both particle systems and *flocking*. The members of a flock, typically fewer in number than

**Table 4.5** Characteristics of Group Types

| Type of Group | Number of Elements | Incorporated Physics | Intelligence |
|---|---|---|---|
| **Particles** | many | much—with environment | none |
| **Flocks** | some | some—with environment and other elements | limited |
| **Autonomous behavior** | few | little | much |

particles in a particle system, behave according to more sophisticated physics and a bit of intelligence. Simple cognitive processes that control the movement of a member are modeled and might include such behavior as goal-directed motion and the avoidance of obstacles. Adding more intelligence to the members in a group results in more interesting individual behaviors and sometimes goes by the name *autonomous behavior*. Modeling autonomous behavior tends to involve fewer participants, less physics, and more intelligence. Particle systems, flocking, and autonomous behavior are examples of independently behaving members of groups with varying levels of autonomy, physical characteristics, and simulated motions (Table 4.5).

## 4.5.1 Particle Systems

Because typically there are a large number of elements in a particle system, simplifying assumptions are used in both the rendering and the calculation of their motions. Various implementations make different simplifying assumptions. Some of the common assumptions made are

Particles do not collide with other particles

Particles do not cast shadows, except in an aggregate sense

Particles only cast shadows on the rest of the environment, not on each other

Particles do not reflect light—they are each modeled as point light sources

Particles are often modeled as having a finite life span, so that during an animation there may be hundreds of thousands of particles used but only thousands active at any one time. Randomness is introduced into most of the modeling and processing of particles to avoid excessive orderliness. In computing a frame of motion for a particle system, the following steps are taken:

1. Any new particles that are born during this frame are generated
2. Each new particle is assigned attributes
3. Any particles that have exceeded their allocated life span are terminated

4. The remaining particles are animated and their shading parameters changed according to the controlling processes
5. The particles are rendered

See Figure 4.48. The steps are then repeated for each frame of the animation. If the user can enforce the constraint that there are a maximum number of particles active at any one time, then the data structure can be static and particle data structures can be reused as one dies and another is created in its place.

### Particle Generation

Particles are typically generated according to a controlled stochastic process. For each frame, a random number of particles are generated using some user-specified distribution centered at the desired average number of particles per frame (Equation 4.108). The distribution could be uniform or Gaussian or anything else the animator wants. In Equation 4.108, *Rand*(_) returns a random number from –1.0 to +1.0 in the desired distribution, and *range* scales it into the desired range. If the particles are used to model a fuzzy object, then it is best to use the area of the screen covered by the object to control the number of particles generated (Equation 4.109). The features of the random number generator, such as average value and variance, can be a function of time to enable the animator to have more control over the particle system.

$$\#\text{ of particles} = \mathit{average} + \mathit{Rand}(\_) \cdot \mathit{range} \qquad \textbf{(Eq. 4.108)}$$

$$\#\text{ of particles} = \mathit{average} + \mathit{Rand}(\_) \cdot \mathit{range} \cdot \mathit{screenArea} \qquad \textbf{(Eq. 4.109)}$$

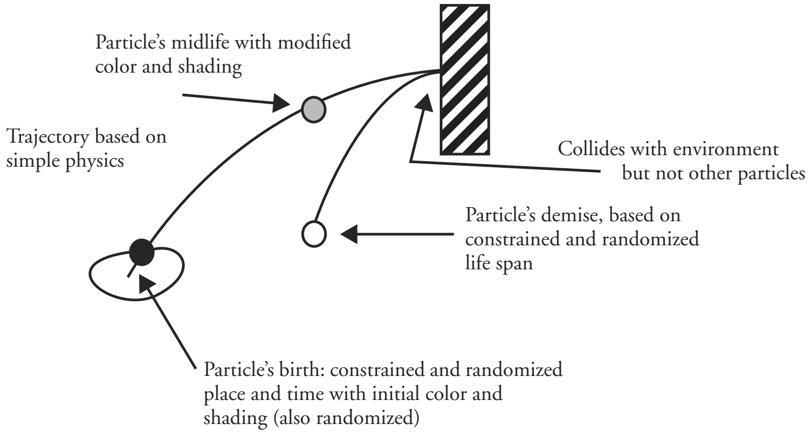

Particle's midlife with modified color and shading
Trajectory based on simple physics
Collides with environment but not other particles
Particle's demise, based on constrained and randomized life span
Particle's birth: constrained and randomized place and time with initial color and shading (also randomized)

**Figure 4.48** The life of a particle

### Particle Attributes

The attributes of a particle determine its motion status, its appearance, and its life in the particle system. Typical attributes are

position
velocity
shape parameters
color
transparency
lifetime

Each of the attributes is initialized when the particle is created. Again, to avoid uniformity, the user typically randomizes values in some controlled way. The position and velocity are updated according to the particle's motion. The shape parameters, color, and transparency control the particle's appearance. The lifetime attribute is a count of how many frames the particle will exist in.

### Particle Termination

At each new frame, each particle's lifetime attribute is decremented by one. When the attribute reaches zero, the particle is removed from the system. This completes the particle's life cycle and can be used to keep the number of particles active at any one time within some desired range of values.

### Particle Animation

Typically, each active particle in a particle system is animated throughout its life. This activation includes not only its position and velocity but also its display attributes: shape, color, and transparency. To animate the particle's position in space, the user considers forces and computes the resultant particle acceleration. The velocity of the particle is updated from its acceleration, and then the average velocity is computed and used to update the particle's position. Gravity, other global force fields (e.g., wind), local force fields (vortices), and collisions with objects in the environment are typical forces modeled in the environment.

The particle's color and transparency can be a function of global time, its own life span remaining, its height, and so on. The particle's shape can be a function of its velocity. For example, an ellipsoid can be used to represent a moving particle where the major axis of the ellipsoid is aligned with the direction of travel and the ratio of the ellipsoid's length to the radius of its circular cross section is related to its speed.

### Particle Rendering

To simplify rendering, model each particle as a point light source so that it adds color to the pixel(s) it covers but is not involved in visible surface algorithms (except to be hidden) or shadowing. In some applications, shadowing effects have been determined to be an important visual cue. The density of particles between a position in space and a light source can be used to estimate the amount of shadowing. See Blinn [2], Ebert, Carlson, and Parent [8], and Reeves [23] for some of these considerations. For more information, see Reeves and Blau on particle systems [24] and Sims on particle animation and rendering [28].

### Particle System Representation

A particle is represented by a tuple [*x, v, f, m*], which holds its position, velocity, force accumulator, and mass.

```
typedef particle_struct struct {
  vector3D      p;
  vector3D      v;
  vector3D      f;
  float         mass;
} particle;
```

The state of a particle, [*x, v*], will be updated by [*v, f*/*m*] by the ODE (ordinary differential equation) solver. The solver can be considered to be a black box to the extent that the actual method used by the solver does not have to be known.

A particle system is an array of particles with a time variable: [**p, n, t*].

```
typedef particleSystem_struct struct {
  particle  *p;
  int           n;
  float         t;
} particleSystem;
```

### Updating Particle System Status

The particle system is updated by

```
update(pSystem)
{
  clearForces(pSystem)
  computeForces(pSystem)
  getState(pSystem,array1)
  computeDerivative(pSystem,arrayw)
  addVector(array1,array2,array2,n)
  saveState(array2,pSystem)
  t += deltaT
}
```

### Forces

Forces can be unary, particle pair, or environmental. Unary forces include gravity and viscous drag. Particle pair forces, if desired, can be represented by springs and dampers. However, implementing particle pair forces for each possible pair of particles can be prohibitively expensive in terms of computational requirements. Auxiliary processing, such as bucket sorting, can be employed to consider only *n*-nearest-neighbor pairs and reduce the computation required. Environmental forces arise from a particle's relationship to the rest of the environment. Gravitational forces or forces arising from spring-damper configurations may be modeled using various geometric elements in the environment.

### Particle Life Span

Typically, each particle will have a life span. The particle data structure itself can be reused in the system so that a particle system might have tens of thousands of particles over the life of the simulation but only, for example, one thousand in existence at any one time. Initial values are set pseudo randomly so that particles are spawned in a controlled manner but with some variability.

## 4.5.2 Flocking Behavior

Flocking can be characterized as having a moderate number of members (relative to particle systems and autonomous behavior), each of which is controlled by a relatively simple set of rules that operate locally. The members exhibit limited intelligence and are governed by relatively simple physics. The physics-based modeling typically includes collision avoidance, gravity, and drag. As compared to particle systems, there are fewer elements and there is usually more interelement interaction modeled. In addition, members' behavior usually models some limited intelligence as opposed to being strictly physics based.

As with particle systems, aggregate behavior emerges from the local behavior of members of the flock; it is *emergent behavior*. The flocking behavior manifests itself as a goal-directed body, able to split into sections and re-form, creating organized patterns of flock members that can perform coordinated maneuvers.

For purposes of this discussion, the birds-in-flight analogy will be used, although, in general, any collection of participants that exhibits this kind of group behavior falls under the rubric *flocking*. For example, flocking behavior is often used to control herds of animals moving over terrain. Of course, in this case, their motion is limited to the surface of a two-dimensional manifold. To use the flock analogy but acknowledge that it refers to a more general concept, Reynolds uses the term *boid* to refer to a member of the generalized flock. Much of this discussion is taken from his paper [25].

There are two main forces at work in keeping a collection of objects behaving like a flock: *collision avoidance* and *flock centering*. These are competing tendencies and must be balanced, with collision avoidance taking precedence.

Collision avoidance is relative to other members of the flock as well as other obstacles in the environment. Avoiding collision with other members in the flock means that some spacing must be maintained between the members even though all members are usually in motion and that motion usually has some element of randomness associated with it to break up unnatural-looking regularity. However, because the objects, as members of a flock, are typically heading in the same direction, the relative motion between flock members is usually small. This facilitates maintaining intermember spacing and, therefore, collision avoidance among members. Considering only intermember collision avoidance, the objective should be that resulting adjustments bring about small and smooth movements.

Flock centering has to do with each member trying to be just that—a member of a flock. As Reynolds [25] points out, a global flock centering force does not work well in practice because it prohibits flock splitting, such as that often observed when a flock passes around an obstacle. Flock centering should be a localized tendency so that members in the middle of a flock will stay that way and members on the border of a flock will have a tendency to veer toward their neighbors on one side. Localizing the flocking behavior also reduces the order of complexity of the controlling procedure.

### Local Control

Controlling the behavior of a flock member with strictly local behavior rules is not only computationally desirable; it also seems to be intuitively the way that flocks operate in the real world. The objective is to be as local as possible, with no reference to global conditions of the flock or environment. There are three processes that might be modeled: *physics, perception,* and *reasoning and reaction*. The physics modeled is similar to that described in particle systems: gravity, collision detection, and collision response. Perception concerns the information about the environment to which the flock member has access. Reasoning and reaction are incorporated into the module that negotiates among the various demands produced as a result of the perception and includes the aforementioned collision avoidance and flock centering. An additional force that is useful in controlling the member's reaction to movements of the flock is velocity matching, whereby a flock member has an urge to match its own velocity with that of its immediate neighbors. Velocity matching helps a flock member to avoid collision with other members and keeps a flock-centered boid flock centered.

### Perception

The main distinction between flocking and particle systems is the modeling of perception and the subsequent use of the resulting information to drive the reaction and reasoning processes. When one localizes the control of the members, a localized area of perception is modeled. Usually the "sight" of the member is restricted to just those members around it—or further to just those members generally in front of it. The position of the member can be made more precise to generate better-defined arrangements of the flock members. For example, if a flock member always stays at a 45-degree angle, to the side and behind, to adjacent members, then a tendency to form a diamond pattern results. If a flock member always stays behind and to the side of one member with no members on the other side, then a V pattern can be formed. Speed can also affect perception; the field of view can extend forward and be slimmer in width as speed increases. To effect a localized field of view, a boid should

Be aware of itself and two or three of its neighbors

Be aware of what is in front of it and have a limited field of view (*fov*)

Have a distance-limited field of view (*fov*)

Be influenced by objects within the line of sight

Be influenced by objects based on distance and size (angle subtended in the *fov*)

Be affected by things using a distance-squared or distance-cubed weighting function

Have a general migratory urge but no global objective

Not follow a designated leader

Not have knowledge about a global flock center

### Interacting with Other Members

A member interacts with other members of the flock to maintain separation without collision while trying to maintain membership in the flock. There is an attractive force toward other members of the flock while a stronger, but shorter-range, repulsion from individual members of the flock exists. In analyzing the behavior of actual flocks, Potts [21] observed a *chorus line effect* in which a wave motion travels faster than any rate chained reaction time could produce. This may be due to perception extending beyond a simple closest-neighbor relationship.

### Interacting with the Environment

The main interaction between a flock member and the environment is collision avoidance, for which various approaches can be used. Force fields are the simplest to implement and give good results in simple cases. However, in more demanding situations, force fields can give undesirable results. The trade-offs of various strategies are discussed later, under "Collision Avoidance."

### Global Control

There is usually a global goal that is used to control and direct the flock. This can be used either to influence all members of the flock or to influence just the leader. The animation of the current leader of the flock is often scripted to follow a specific path or is given a specific global objective. Members can have migratory urge, follow the leader, stay with the pack, or exhibit some combination of these urges.

### Flock Leader

To simulate the behavior of actual flocks, the animator can have the leader change periodically. Presumably, actual flocks change leaders because the wind resistance is strongest for the leader and rotating the job allows the birds to conserve energy. However, unless changing the flock leader adds something substantive to the resulting animation, it is easier to have one designated leader whose motion is scripted along a path to control the flock's general behavior.

### Negotiating the Motion

In producing the motion, three low-level controllers are commonly used. They are, in order of priority, collision avoidance, velocity matching, and flock centering. Each of these controllers produces a directive that indicates desired speed and direction (a velocity vector). The task is to negotiate a resultant velocity vector given the various desires.

As previously mentioned, control can be enforced with repulsion from other members and environmental obstacles and attraction to flock center. However, this has major problems as forces can cancel each other out. Reynolds refers to the programmatic entity that resolves competing urges as the *navigation module*. As Reynolds points out, averaging the requests is usually a bad idea in that requests can cancel each other out and result in nonintuitive motion. He suggests a prioritized acceleration allocation strategy in which there is a finite amount of control available, for example, one unit. A control value is generated by the low-level controllers in addition to the velocity vector. A fraction of control is allocated according to priority order of controllers. If the amount of control runs out, then one or more of the controllers receive less than what they requested. If less than the amount of total possible control is not allocated, then the values are normalized (to sum to the value of one, for example). A weighted average is then used to compute the final velocity vector. Governors may be used to dampen the resulting motion, as by clamping the maximum velocity or clamping the maximum acceleration.

In addition to prioritized behaviors, the physical constraints of the flock member being modeled need to be incorporated into the control strategy. Reynolds suggests a three-stage process consisting of navigation, piloting, and flying. See Figure 4.49. Navigation, as discussed above, negotiates among competing desires and resolves them into a final desire. The pilot module incorporates this desire into something the member model is capable of doing at the time, and the flight model is responsible for the final articulation of the commands from the pilot module.

The navigation module arbitrates among the urges and produces a resultant directive to the member. This information is passed to the pilot model, which instructs the flock member to react in a certain way in order to satisfy the directive. The pilot model is responsible for incorporating the directive into the constraints imposed on the flock member. For example, the weight, current speed and acceleration, and internal state of the member can be taken into account at this stage. Common constraints include clamping acceleration, clamping velocity, and clamping velocity from below. The result from the pilot module is the specific action that is to be put into effect by the flock member model. The flight module is responsible for producing the parameters that will animate that action.

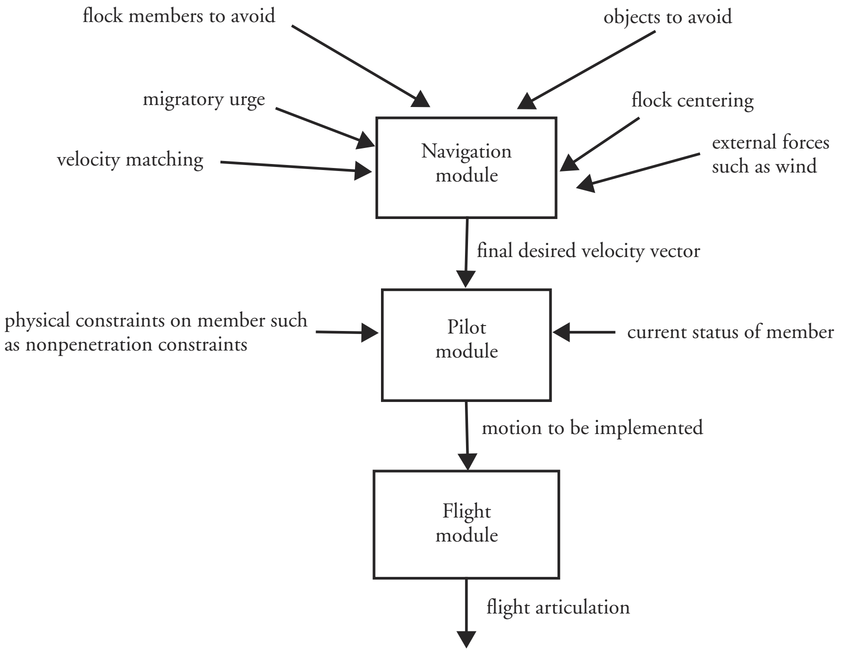


**Figure 4.49** Negotiating the motion

### *n*-Squared Complexity

One of the problems with flocking systems, in which knowledge about neighbors is required and the number of members of a flock can get somewhat large, is that the processing complexity is *n*-squared. Even when interactions are limited to some *k* nearest neighbors, it is still necessary to find those *k* nearest neighbors out of the total population of *n*. One way to find the nearest neighbors efficiently is to perform a 3D bucket sort and then check adjacent buckets for neighbors. Such a bucket sort can be updated incrementally by adjusting bucket positions of any members that deviate too much from the bucket center as the buckets are transformed along with the flock. There is, of course, a time-space trade-off involved in bucket size; the smaller the buckets, the more buckets needed but the fewer members per bucket on average. This does not completely eliminate the *n*-squared problem because of worst-case distributions, but it is effective in practice.

Instead of searching for adjacent members, members can use message passing, with each member informing other members of its whereabouts. But either each member must send a message to every other member or the closest neighbors must still be searched for.

### Collision Avoidance

Several strategies can be used to avoid collisions. The ones mentioned here are from Reynolds's paper [25] and from his course notes [26]. These strategies, in one way or another, model the flock member's field of view and visual processing. A trade-off must be made between the complexity of computation involved and how effective the technique is in producing realistic and intuitive motion.

The simple strategy is to position a limited-extent, repelling force field around every object. As long as a flock member maintains a safe distance from an object, there is no force imparted by the object to the flock member. Whether this is so can easily be determined by calculating the distance[3] between the center point of the flock member and the center of the object. Once this distance gets below a certain threshold, the distance-based force starts to gently push the flock member away from the object. As the flock member gets closer, the force grows accordingly. See Figure 4.50. The advantages of this are that in many cases it effectively directs a flock member away from collisions, it is easy to implement, and its effect smoothly decreases the farther away the flock member is from the object. It has its drawbacks, too.

In some cases, the force field approach fails to produce the motion desired (or at least expected). It prevents a flock member from traveling close and parallel to the

[3] As in many cases in which the calculation of distance is required, one can use distance squared, thus avoiding the square root required in the calculation of distance.

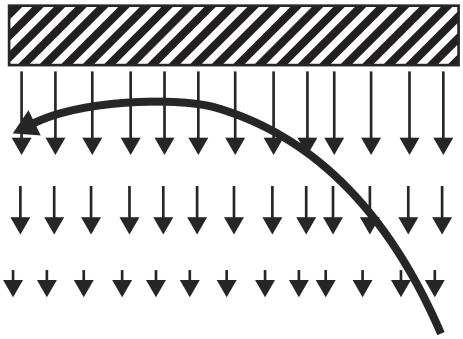


**Figure 4.50** Force field collision avoidance

surface of an object. The repelling force is as strong for a flock member moving parallel to a surface as it is for a flock member moving directly toward the surface. A typical behavior for a flock member attempting to fly parallel to a surface is to veer away and then toward the surface. The collision avoidance force is initially strong enough to repel the member so that it drifts away from the surface. When the force weakens, the member heads back toward the surface. This cycle of veering away and then toward the surface keeps repeating. Another problem with simple collision avoidance forces occurs when they result in a force vector directly toward the flock member. In this case, there is no direction indicated in which the member should veer; it has to stop and back up—an unnatural behavior. Aggregate forces can also prevent a flock member from finding and moving through an opening that may be more than big enough for passage but for which the forces generated by surrounding objects is enough to repel the member. See Figure 4.51.

The problem with a simple repelling force field approach is that there is no reasoned strategy for avoiding the potential collision. Various path planning heuristics that can be viewed as attempts to model simple cognitive processes in the flock member are useful. For example, a bounding sphere can be used to divert the flock member's path around objects by steering away from the center toward the rim of the sphere (Figure 4.52).

Once the flock member is inside the sphere of influence of the object, its direction vector can be tested to see if indicates a potential intersection with the bounding sphere of the object. This calculation is the same as that used in ray tracing to see if a ray intersects a sphere (Figure 4.53).

When a potential collision has been detected, the steer-to-avoid procedure can be invoked. It is useful to calculate the point, *B,* on the boundary of the bounding sphere that is in the plane defined by the direction vector, *V;* the location of the flock member, *P;* and the center of the sphere, *C* (Figure 4.54).

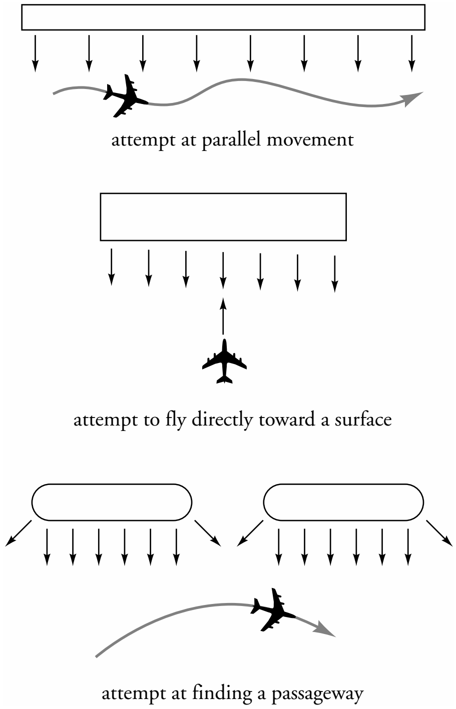

attempt at parallel movement
attempt to fly directly toward a surface
attempt at finding a passageway

**Figure 4.51** Problems with force field collision avoidance

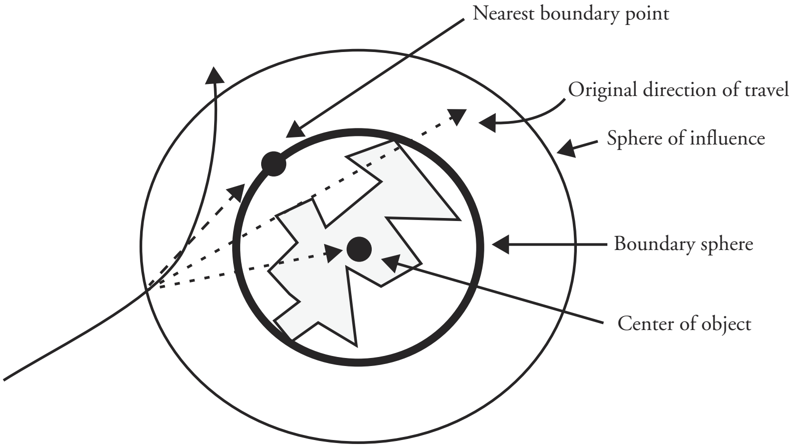
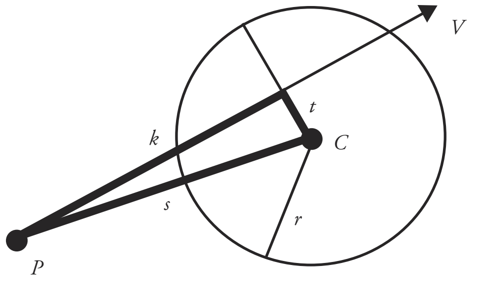


**Figure 4.52** Steering to avoid a bounding sphere



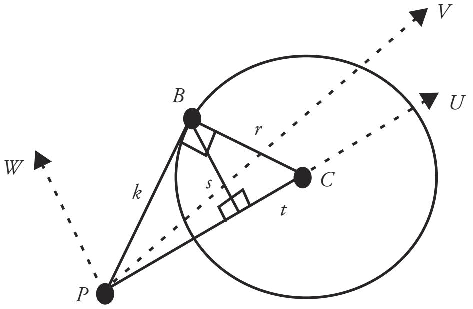
$$s = |C - P|$$

$$k = (C - P) \bullet \frac{V}{|V|}$$

$$t = \sqrt{s^2 - k^2}$$

$t < r$ indicates penetration with bounding sphere

**Figure 4.53** Testing for potential collision with a bounding sphere



$$k = \sqrt{|C - P|^2 - r^2}$$

$$r^2 = s^2 + t^2$$

$$k^2 = s^2 + (|C - P| - t)^2$$

$$k^2 = r^2 - t^2 + (|C - P| - t)^2$$

$$k^2 = r^2 - t^2 + |C - P|^2 - 2 \cdot |C - P| \cdot t + t^2$$

$$t = \frac{k^2 - r^2 - |C - P|^2}{-2 \cdot |C - P|}$$

$$s = \sqrt{r^2 - t^2}$$

$$U = \frac{C - P}{|C - P|}$$

$$W = \frac{(U \times V) \times U}{|(U \times V) \times U|}$$

$$B = P + (|C - P| - t) \cdot U + s \cdot W$$

**Figure 4.54** Calculation of point *B* on the boundary of a sphere

Steering to the boundary point on the bounding sphere is a useful strategy if a close approach to the object's edge is not required. For a more accurate and more computationally expensive strategy, the flock member can steer to the silhouette edges of the object, those edges that share a back face and a front face with respect to the flock member's position and, collectively, define a nonplanar manifold. Faces can be tagged as front or back by taking the dot product of the normal with a vector from the flock member to the face. The closest distance between the semi-infinite direction of travel vector to a silhouette edge can be calculated. If there is space for the flock member's body and a comfort zone in which to fly around this point, then this point can be steered toward Figure 4.55.

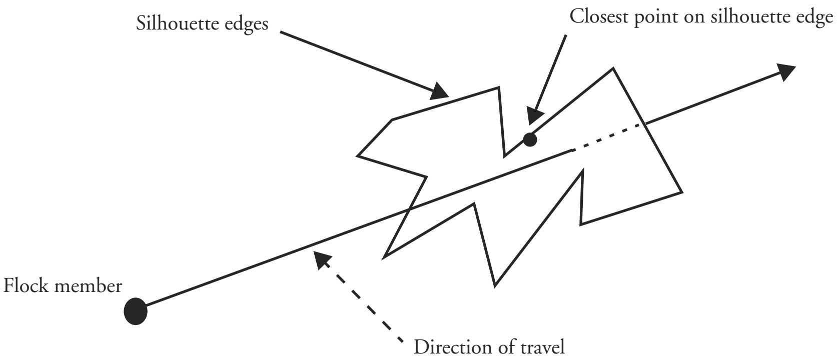


**Figure 4.55** Determining steer-to point from silhouette edges

An alternative strategy is to sample the environment by emitting virtual feelers to detect potential collision surfaces. The feelers can be emitted in a progressively divergent pattern, such as a spiral, until a clear path is found.

Another strategy is to project the environment to an image plane. By generating a binary image, the user can search for an uncovered group of pixels. Depth information can be included to allow searching for discontinuities and to estimate the size of the opening in three-space. The image can be smoothed until a gradient image is attained, and the gradient can be followed to the closest edge.

### Splitting and Rejoining

When a flock is navigating among obstacles in the environment, one of their more interesting behaviors is the splitting and rejoining that result as members veer in different directions and break into groups as they attempt to avoid collisions. If the groups stay relatively close, then flock membership urges can bring the groups back together to re-form the original, single flock. Unfortunately, this behavior is difficult to produce because a balance must be created between collision avoidance and the flock membership urge, with collision avoidance taking precedence in critical situations. Without this precise balance, either a flock faction will split and never return to the flock or flock members will not split off as a separate group but will only individually avoid the obstacle and disrupt the flock formation.

### Modeling Flight

Since flocking is often used to model the behavior of flying objects, it is useful to review the principles of flight. A local coordinate system of *roll, pitch,* and *yaw* is commonly used to discuss the orientation of the flying object. The roll of an object is the amount that it rotates to one side from an initial orientation. The pitch is the amount that its nose rotates up or down, and the yaw is the amount it rotates about its up vector. See Figure 4.56.

Specific to modeling flight, but also of more general use, is a description of the forces involved in aerodynamics. In *geometric flight,* flight is controlled by *thrust, drag, gravity,* and *lift* (Figure 4.57). Thrust is the force used to propel the object forward and is produced by an airplane's engine or the flapping of a bird's wings. Drag is the force induced by an object traveling through a medium such as air (i.e., wind resistance) and works in the direction directly against that in which the object is traveling. Gravity is the force that attracts all objects to the earth and is modeled by a constant downward acceleration. Lift is the upward force created by air traveling around an airfoil such as the wings of an airplane or bird. The airfoil's shape is such that air traveling over it has to travel a longer distance than air traveling under it. As a result, there is less pressure above the airfoil, which produces most of the lift. Additional lift is generated at high angles of attack by the pressure of wind against the lower surface of the wing (Figure 4.58). For straight and level flight, lift cancels gravity and thrust exceeds drag.

In flight, a turn is induced by a *roll,* in which the up vector of the object rotates to one side. Because the up vector is rotated to one side, the lift vector is not directly opposite to gravity. Such a lift vector can be decomposed into a vertical

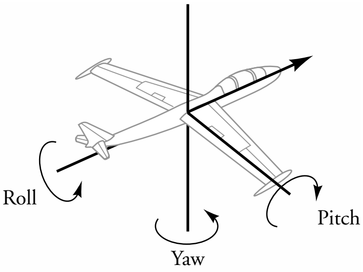


**Figure 4.56** Roll, pitch, and yaw of local coordinate system

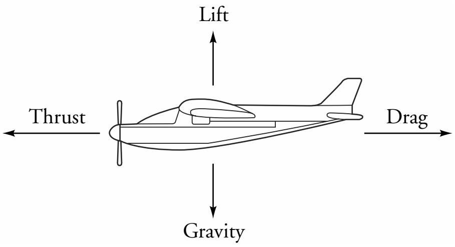


**Figure 4.57** Forces of flight

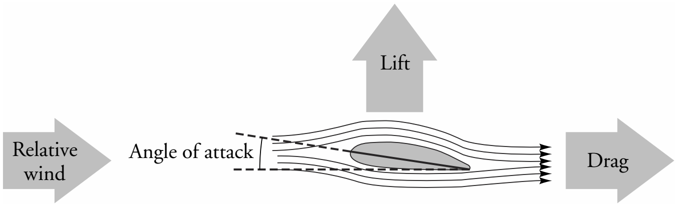


**Figure 4.58** Lift produced by an airfoil

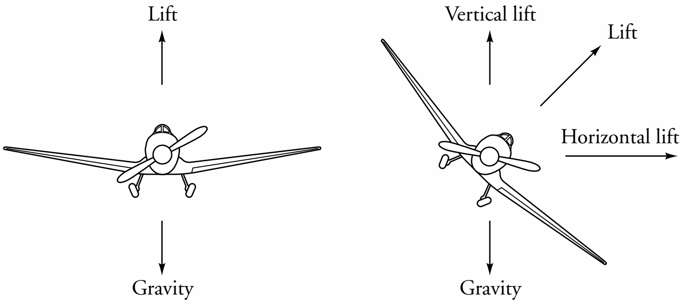


**Figure 4.59** Lifting forces

component, the component that is directly opposite to gravity, and the horizontal component, the component to the side of the object. A flying object turns by being lifted sideways by the vertical component of the lift; this is why flying objects tilt into a turn. For the object to be directed into the turn, there must be some yaw. If a plane is flying level so that lift cancels gravity, tilting the lift vector to one side will result in a decrease in the vertical component of lift. Thus, to maintain level flight during a turn, one must maintain the vertical component of lift by increasing the speed. See Figure 4.59.

Increasing the pitch increases the angle the wing makes with the direction of travel. This results in increased lift and drag. However, if thrust is not increased when pitch is increased, the increased drag will result in a decrease in velocity, which, in turn, results in decreased lift. So, to fly up, thrust and pitch need to be increased.

These same principles are applicable to a soaring bird. The major difference between the flight of a bird and that of an airplane is in the generation of thrust. In a plane, thrust is produced by the propeller. In a bird, thrust is produced by the flapping of the wings.

Some of the important points to notice in modeling flight are as follows:

Turning is effected by horizontal lift

Increasing pitch increases drag

Increasing speed increases lift

### 4.5.3 Autonomous Behavior

*Autonomous behavior,* as used here, refers to the motion of an object that results from modeling its cognitive processes. Usually such behavior is applied to relatively few objects in the environment. As a result, emergent behavior is not typically associated with autonomous behavior, as it is with particle systems and flocking behavior, simply because the numbers are not large enough to support the sensation of an aggregate body. To the extent that cognitive processes are modeled in flocking, autonomous behavior shares many of the same issues, most of which result from the necessity to balance competing urges.

Autonomous behavior models an object that knows about the environment in which it exists, reasons about the state it is in, plans a reaction to its circumstances, and carries out actions that affect its environment. The environment, as it is sensed and perceived by the object, constitutes the external state. In addition, there is an internal state associated with the object made up of time-varying urges, desires, and emotions, as well as (usually static) rules of behavior.

It should not be hard to appreciate that modeling behavior can become arbitrarily complex. Autonomous behavior can be described at various levels of complexity, depending on how many and what type of cognitive processes are modeled. Is the objective simply to produce some interesting behavior, or is it to simulate how the actual object would operate in a given environment? How much and what kinds of control are to be exercised by the user over the autonomous agent? Possible aspects to include in the simulation are sensors, especially vision and touch; perception; memory; causal knowledge; commonsense reasoning; emotions; and predispositions. Many of these issues are more the domain of artificial intelligence than of computer graphics. However, besides its obvious role in rendering results, computer graphics is also a relevant domain for which to discuss this work because the objective of the cognitive simulation is motion control. In addition, spatial reasoning is usually a major component of cognitive modeling and, therefore, draws heavily on algorithms associated with computer graphics.

Autonomous behavior is usually associated with animal-like articulated objects. However, it can be used with any type of object, especially if that object is typically controlled by a reasoning agent. Obvious examples are cars on a highway, planes in the air, or tanks on a battlefield. Autonomous behavior can also be imparted to inanimate objects whose behavior can be humanized, such as kites, falling leaves, or clouds. The current discussion focuses on the fundamentals of modeling behavior. Issues specific to managing the behavior of articulated figures are taken up later.

### Knowing the Environment

There are various ways in which an object can know about its environment. At the simplest level, the object simply has access to the environment database. It has perfect and complete knowledge about its own place and about other objects in the environment. More complex modeling of acquiring knowledge of the environment can involve sensors, including vision, and memory. Touch, modeled by detecting collisions, can be used in navigating through the environment. Vision can be modeled by rendering a scene from the point of view of the object. One could imagine situations in which it might be important to monitor sounds and smells. Memory consists of recording the sensed information such as current positions, motions, and display attributes (e.g., color) of other objects in the environment.

### Internal State

Internal state is modeled partially by intentions. Intentions take on varied importance depending on the urges they are meant to satisfy. The instinct to survive is perhaps the strongest urge and, as such, takes precedence over, say, the desire to scratch an itch. Internal state also includes such things as inhibitions, identification of areas of interest, and emotional state. These are the internal state variables that are inputs to the rest of the behavioral model.While the internal state variables may actually represent a continuum of importance, Blumberg and Galyean [4] group them into three precedence classes: imperatives, things that must be done; desires, things that should be done, if they can be accommodated by the reasoning system; and suggestions, ways to do something should the reasoning system decide to do that something.

### Levels of Behavior

There are various levels at which an object's motion can be modeled. In addition to Blumberg and Galyean [4], Zeltzer [34] and Korein and Badler [18] discuss the importance of decomposing high-level goals into object-specific manipulation of the available degrees of freedom afforded by the geometry of the object. The levels of behavior are differentiated according to the level of abstraction at which the motion is conceptualized. They provide a convenient hierarchy in which the implementation of behaviors can be discussed. The number of levels used is somewhat arbitrary. Those presented here are used to emphasize the motion control aspects of autonomous behavior as opposed to the actual articulation of that motion. *Internal state* and *knowledge of the external world* are inputs to the *reasoning unit,* which produces a strategy intended to satisfy some objective. A *strategy* is meant to define the *what* that needs to be done. The *planner* turns the strategy into a *sequence of actions* (the *how*), which is passed to the *movement coordinator.* The movement coordinator selects the appropriate *motor activities* at the appropriate time. The motor activities control specific *degrees of freedom* (DOF) of the object. See Figure 4.60.

### Keeping Behavior under Control

One of the fundamental concerns with autonomous behavior, as with all high-level motion control, is how to model the behaviors so that they are generated automatically but are still under the control of the animator. One of the motivations for modeling behaviors is to relieve the animator from specifying the time-

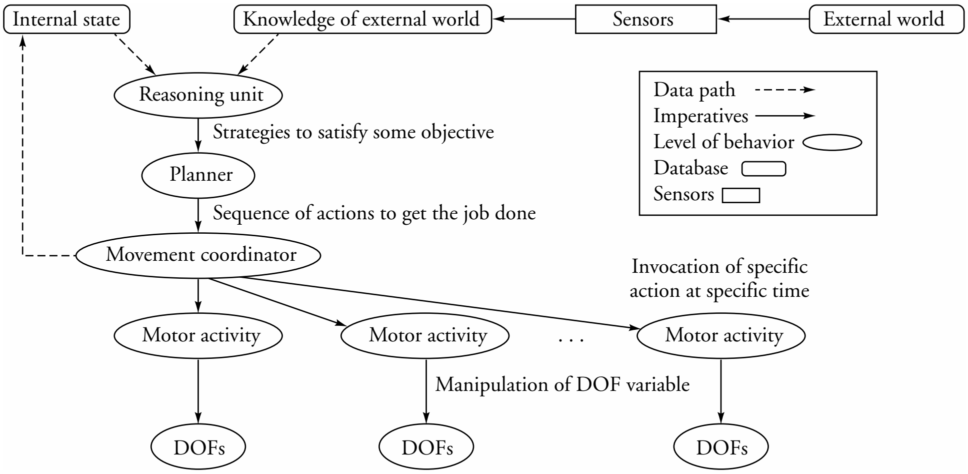


**Figure 4.60** Levels of behavior

varying values for each of the degrees of freedom of the underlying geometry associated with the object. Pure autonomy should probably not be the ultimate goal in designing autonomous agents; often behavior needs to be controlled in order to be relevant to the objectives of the animator. Also, the control needs to occur at various levels of specificity.

The various levels of behavior provide hooks with which control over behavior can be exercised. The animator can insert external control over the behavior by generating imperatives at any of the various levels: strategies, action sequences, and/or activity invocation. In addition, more general control can be exercised by setting internal state variables.

### Arbitration between Competing Intentions

As with arbitration among competing forces in flocking, behaviors cannot, in general, be averaged and be expected to produce reasonable results. The highest precedent behavior must be accommodated at all costs, and other behaviors may be accommodated if they do not contradict the effect of the dominant behavior. One approach is to group behaviors into sets in which one behavior is selected. Selected behaviors may then be merged to form the final behavior.

Modeling behavior, especially human behavior, can get arbitrarily complex. Covering all of the complexities is the domain of artificial intelligence. However, even a simple rule-based system can generate behavior that, if not always interesting, is at least autonomous.

## 4.6 Implicit Surfaces

Implicitly defined surfaces are surfaces defined by all those points that satisfy some equation, $f(P) = 0$, called the *implicit function*. A common approach to using implicit surfaces to define objects useful for animation is to construct the implicit function as a summation of implicitly defined primitive functions. Interesting animations can be produced by animating the relative position and orientation of the primitive functions or by animating parameters used to define the functions themselves. Implicit surfaces lend themselves to shapes that are smooth and organic looking; animated implicit surfaces are useful for modeling liquids, clouds, and fanciful animal shapes.

An extensive presentation of implicit surface formulations is not appropriate material for this book but can be found in several sources, in particular in Bloomenthal's edited volume [3]. A brief overview of commonly used implicit surfaces and their use in animations is presented here.

### 4.6.1 Basic Implicit Surface Formulation

In general, an *implicit surface* is defined by the collection of points that satisfy some implicit function, $f(P) = 0$. The surface is referred to as implicit because it is only implicitly defined, not explicitly represented. As a result, when an explicit definition of the surface is needed, as in a graphics display procedure, the surface has to be searched for by inspecting points in space in some organized way.

Implicit surfaces can be directly ray traced. Rays are constructed according to the camera parameters and display resolution, as is typical in ray tracers. Points along the ray are then sampled in such a way as to locate a surface point within some error tolerance.

An explicit polygonal representation of an implicitly defined surface can be constructed by sampling the implicit function at the vertices of a three-dimensional mesh that is constructed so that its extent contains the nonzero extent of the implicit function. The implicit function values at the mesh vertices are then interpolated along mesh edges to estimate the location of points that lie on the implicit surface. Polygonal fragments are then constructed in each cell of the mesh by using any surface points located on the edges of that mesh cell.

The best-known implicit primitive is often referred to as the *metaball* and is defined by a central point ($C$), a radius of influence ($R$), a density function ($f$), and a threshold value ($T$). All points in space that are within a distance $R$ from $C$ are said to have a density of $f(\text{distance}(P, C)/R)$ with respect to the metaball (where distance(*P*, *C*) computes the distance between *P* and *C* and where distance(*P*, *C*)/*R* is the *normalized distance*). The set of points for which *f*(distance (*P*, *C*)/*R*) − *T* = 0 (implicitly) defines the surface, *S*. In Figure 4.61, *r* is the distance at which the surface is defined for the isolated metaball shown because that is the distance at which the function, *f*, evaluates to the threshold value.

Two generalizations of this formulation are useful. The first uses the weighted sum of a number of implicit surface primitives so that the surface-defining implicit function becomes a smooth blend of the individual surfaces (Equation 4.110).

$$F(P) = \Sigma w_i f_i(P) - T \tag{Eq. 4.110}$$

The weights are arbitrarily specified by the user to construct some desired surface. If all of the weights, $w_i$, are one, then the implicit primitives are summed. Because the standard density function has zero slope at zero and one, the primitives will blend together smoothly. Using negative weights will create smooth concavities. Severer concavities can be created by making the weight more negative. Integer weights are usually sufficient for most applications, but noninteger weights can also be used. See Figure 4.62.

The second generalization provides more primitives with which the user can build interesting objects. Most primitives are distance based, and most of those are offset surfaces. Typical primitives use the same formulation as the metaball but allow a wider range of central elements. Besides a single point, a primitive can be defined that uses a line segment, a triangle, a convex polyhedron, or even a concave polyhedron. Any central element for which there is an associated well-defined distance function can be used. The drawback to using more complex central elements, such as a concave polyhedron, is that the distance function is more computationally expensive. Other primitives, which are not strictly offset surfaces but which are still distance based, are the cone-sphere and the ellipse. See Figure 4.63 for examples of offset and other distance-based primitives.

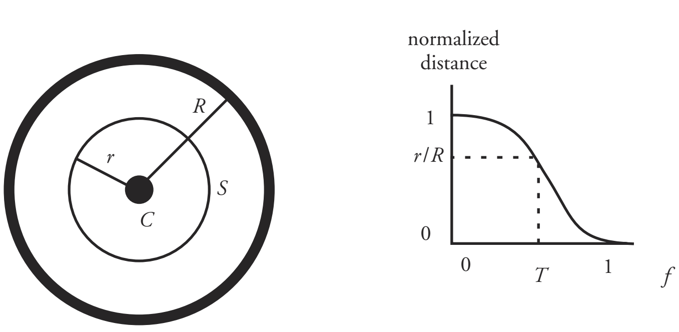
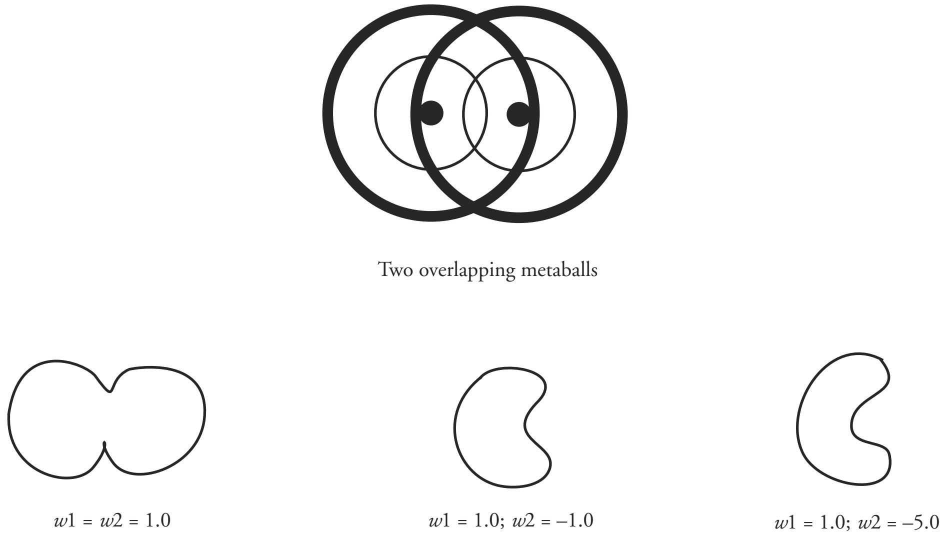


**Figure 4.61** The metaball and a sample density function



Two overlapping metaballs

*w*1 = *w*2 = 1.0

*w*1 = 1.0; *w*2 = –1.0

*w*1 = 1.0; *w*2 = –5.0

**Figure 4.62** Compound implicit surface

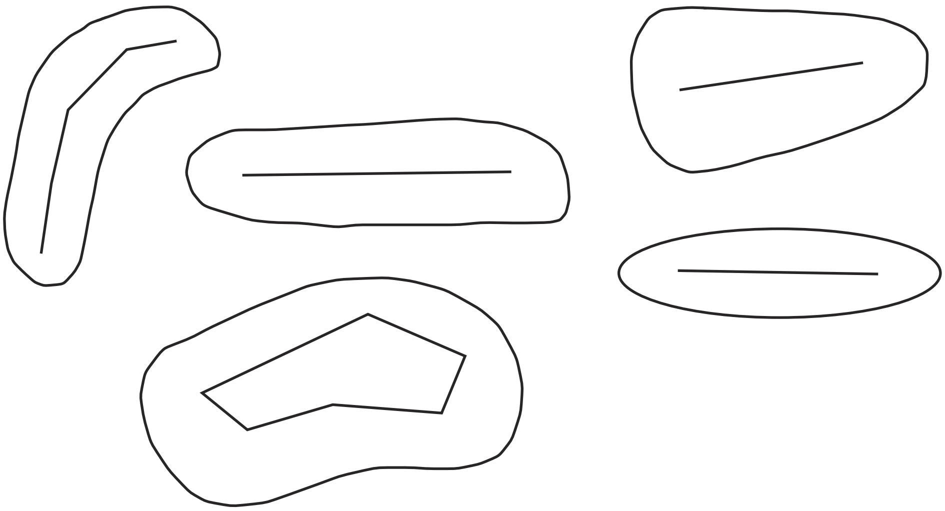


**Figure 4.63** Distance-based implicit surfaces

## 4.6.2 Animation Using Implicitly Defined Objects

In Bloomenthal's book [3],Wyvill discusses several animation effects that can be produced by modifying the shape of implicit surfaces. The most obvious way to achieve these modifications is to control the movement of the underlying central elements. Topological changes are handled nicely by the implicit surface formulation. The points that define a collection of metaballs can be controlled by a simple particle system, and the resulting implicit surface can be used to model water, taffy, or clouds, depending on the display attributes and the number of particles. Central elements consisting of lines can be articulated as a jointed hierarchy.

Motion can also be generated by modifying the other parameters of the implicit surface. The weights of the different implicit primitives can be manipulated to effect bulging and otherwise control the size of an implicit object. A simple blend between objects can be performed by decreasing the weight of one implicit object from one to zero while increasing another implicit object's weight from zero to one. The density function of the implicit objects can also be controlled to produce similar shape deformation effects. Modifying the central element of the implicit is also useful. The elongation of the defining axes of the ellipsoidal implicit can produce squashing & stretching effects. The orientation and length of the axes can be tied to the velocity and acceleration attributes of the implicit object.

### 4.6.3 Collision Detection

Implicitly defined objects, because they are implemented using an implicit function, lend themselves to collision detection. Sample points on the surface of one object can be tested for penetration with an implicit object by merely evaluating the implicit function at those points in space (Figure 4.64). Of course, the effectiveness of this method is dependent on the density and distribution of the sample points on the first object. If a polyhedral object can be satisfactorily approximated by one or more implicit surface primitives, then these can be used to detect collisions of other polyhedral objects.

Because implicit functions can be used effectively in testing for collisions, they can be used to test for collisions between polyhedral objects if they can fit reasonably well on the surface of the polyhedra (Figure 4.65). Of course, the effectiveness

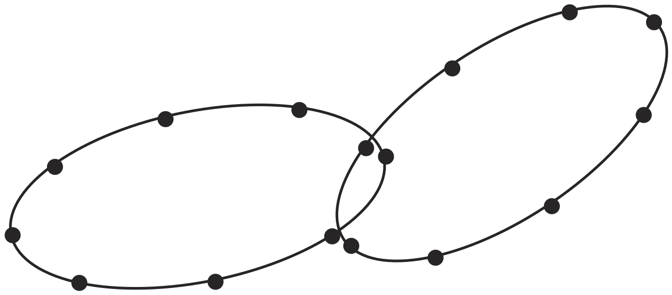


**Figure 4.64** Point samples used to test for collisions

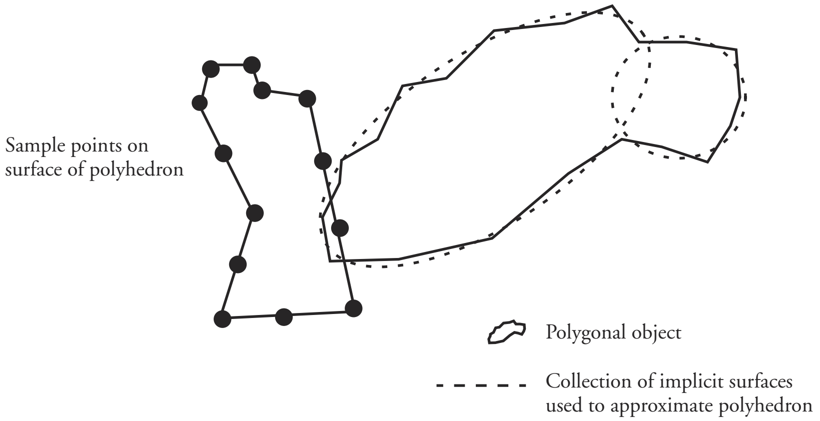

Sample points on
surface of polyhedron
Polygonal object
Collection of implicit surfaces
used to approximate polyhedron

**Figure 4.65** Using implicit surfaces for detecting collisions between polyhedral objects

of the technique is dependent on the accuracy with which the implicit surfaces approximate the polyhedral surface.

### 4.6.4 Deforming the Implicit Surface as a Result of Collision

Marie-Paul Cani [3] [9] [10] has developed a technique to compute the deformation of colliding implicit surfaces. This technique first detects the collision of two implicit surfaces by testing sample points on the surface of one object against the implicit function of the other, as previously described. The overlap of the areas of influence of the two implicit objects is called the *penetration region.* An additional region just outside the penetration region is called the *propagation region* (Figure 4.66).

The density function of each object is modified by the overlapping density function of the other object so as to deform the implicitly defined surface of both objects so that they coincide in the region of overlap, thus creating a contact surface (Figure 4.67). As shown in the example in Figure 4.67, a deformation term is added to $F_i$ as a function of Object$_j$'s overlap with Object$_i$, $G_{ij}$, to form the contact surface. Similarly, a deformation term is added to $F_j$ as a function of Object$_i$'s overlap with Object$_j$, G$_{ji}$. The deformation functions are defined so that the isosurface of the modified density functions, $F_i(p) + G_{ij}(p) = 0$ and $F_j(p) + G_{ji}(p) = 0$, coincide with the surface defined by $F_i(p) = F_j(p)$. Thus, the implicit functions, after they have been modified for deformation and evaluated for points $p$ on the contact surface, are merely $F_1(p) - F_2(p) = 0$ and $F_2(p) - F_1(p) = 0$, respectively. Thus, $G_{ij}$ evaluates to $-F_j$ at the surface of contact.

In the penetration region, the $G$'s are negative in order to compress the respective implicit surfaces as a result of the collision. They are defined so that their

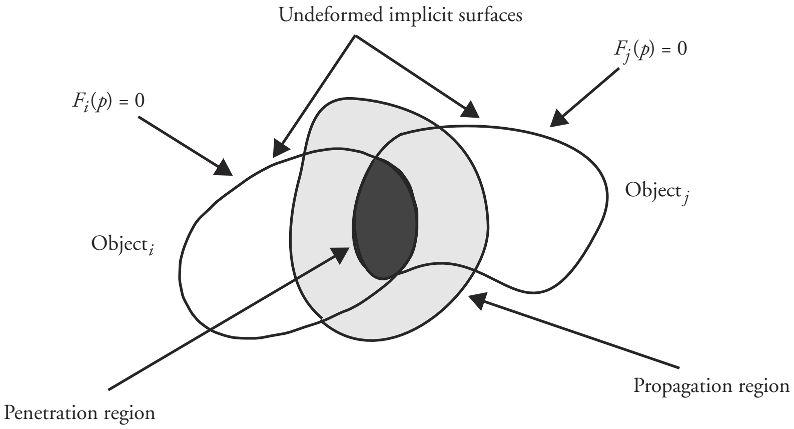


**Figure 4.66** Penetrating implicit surfaces

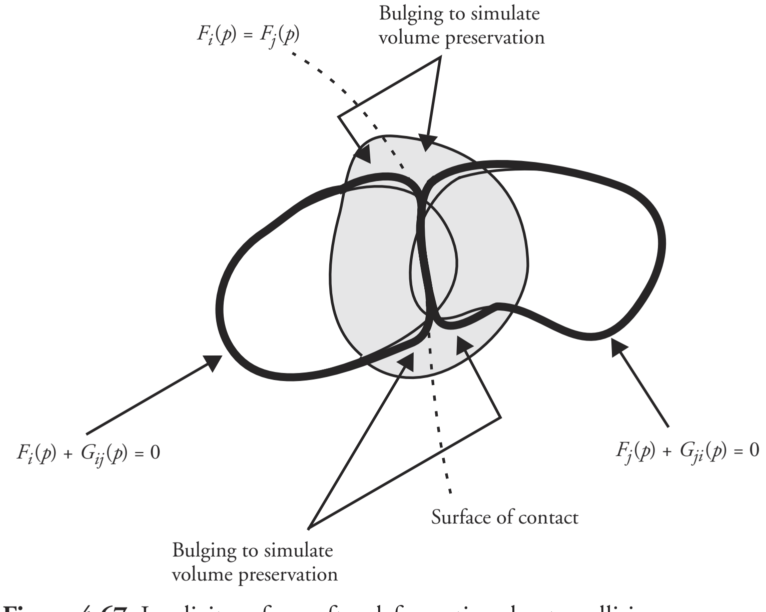


**Figure 4.67** Implicit surfaces after deformation due to collision

effect smoothly fades away for points at the boundary of the penetration region. Consequently, the *G*'s are merely the negative of the *F*'s for points in the penetration region (Equation 4.111).

$$G_{ij}(p) = -F_j(p)$$
$$G_{ji}(p) = -F_i(p) \qquad \textbf{(Eq. 4.111)}$$

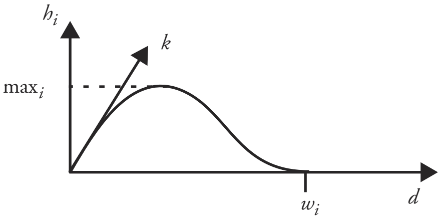


**Figure 4.68** Deformation function in the propagation region

To simulate volume preservation, an additional term is added to the *G*'s so that they evaluate to a positive value in the propagation region immediately adjacent to, but just outside, the penetration region. The effect of the positive evaluation will be to bulge out the implicit surface around the newly formed surface of contact (Figure 4.67).

In the propagation region, *G*(*p*) is defined as a function, *h,* of the distance to the border of the interpenetration region. To ensure $C^1$ continuity between the interpenetration region and the propagation region, $h'(0)$ must be equal to the directional derivative of *G* along the gradient at point *p* (*k* in Figure 4.68). See Cani (née Gascuel) [9] and Gascuel and Cani [10] for more details.

Restoring forces, which arise as a result of the deformed surfaces, are computed and added to any other external forces acting on the two objects. The magnitude of the force is simply the deformation term, *G;* it is in the direction of the normal to the deformed surface at point *p.*

### 4.6.5 Summary

Although their display may be problematic for some graphic systems, implicit surfaces provide unique and interesting opportunities for modeling and animating unusual shapes. They produce very organic-looking shapes and, because of their indifference to changes in genus of the implicitly defined surface, lend themselves to the modeling and animating of fluids and elastic material.

## 4.7 Chapter Summary

The models used to control animation can become complex and math intensive. However, producing more complex motion requires the use of more complex mathematical machinery. Modeling physics and automatically enforcing arbitrary constraints is a nontrivial task. There has been much work in this area that has not been included in this chapter. The discussion here is intended for those without specialized knowledge of math, physics, and numerical methods.

## References

1. D. Baraff, "Rigid Body Simulation," Course Notes, SIGGRAPH 92 (July 1992, Chicago, Ill.).
2. J. Blinn, "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces," *Computer Graphics* (Proceedings of SIGGRAPH 82), 16 (3), pp. 21–29 (July 1982, Boston, Mass.).
3. J. Bloomenthal, ed., *Introduction to Implicit Surfaces,* Morgan Kaufmann, San Francisco, 1997.
4. J. Blumberg and T. Galyean, "Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments," Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 47–54 (August 1995, Los Angeles, Calif.). Addison-Wesley. Edited by Robert Cook. ISBN 0-201-84776-0.
5. M. Cohen, "Interactive Spacetime Control for Animation," *Computer Graphics* (Proceedings of SIGGRAPH 92), 26 (2), pp. 293–302 (July 1992, Chicago, Ill.). Edited by Edwin E. Catmull. ISBN 0-201-51585-7.
6. D. Coynik, *Film: Reel to Reel,* McDougal, Littell, Evanston, Ill., 1976.
7. J. Craig, *Introduction to Robotics Mechanics and Control,* Addison-Wesley, New York, 1989.
8. D. Ebert, W. Carlson, and R. Parent, "Solid Spaces and Inverse Particle Systems for Controlling the Animation of Gases and Fluids," *Visual Computer,* 10 (4), pp. 179–190 (March 1994).
9. M.-P. Gascuel, "An Implicit Formulation for Precise Contact Modeling between Flexible Solids," Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series, pp. 313–320 (August 1993, Anaheim, Calif.). Edited by James T. Kajiya. ISBN 0-201-58889-7.
10. J.-D. Gascuel and M.-P. Gascuel, "Displacement Constraints for Interactive Modeling and Animation of Articulated Structures," *Visual Computer,* 10 (4), pp. 191–204 (March 1994). ISSN 0-178-2789.
11. P. Gill, S. Hammarling, W. Murray, M. Saudners, and M. Wright, "User's Guide for LSSOL: A Fortran Package for Constrained Linear Least-Squares and Convex Quadratic Programming," Technical Report Sol 84-2, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.
12. P. Gill, W. Murray, M. Saunders, and M. Wright. "User's Guide for NPSOL: A Fortran Package for Nonlinear Programming," Technical Report Sol 84-2, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.
13. P. Gill, W. Murray, M. Saunders, and M. Wright. "User's Guide for QPSOL: A Fortran Package for Quadratic Programming," Technical Report Sol 84-6, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1984.
14. P. Gill, W. Murray, and M. Wright, *Practical Optimization,* Academic Press, New York, 1981.
15. J. Gourret, N. Magnenat-Thalmann, and D. Thalmann, "Simulation of Object and Human Skin Deformations in a Grasping Task," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 21–30 (July 1989, Boston, Mass.). Edited by Jeffrey Lane.

16. J. Hahn, "Introduction to Issues in Motion Control," Tutorial 3, SIGGRAPH 88 (August 1988, Atlanta, Ga.).
17. D. Haumann, "Modeling the Physical Behavior of Flexible Objects," SIGGRAPH 87 Course Notes: Topics in Physically Based Modeling (July 1987, Anaheim, Calif.).
18. J. Korein and N. Badler, "Techniques for Generating the Goal-Directed Motion of Articulated Structures," *IEEE Computer Graphics and Applications,* November 1982.
19. J. Ngo and J. Marks, "Spacetime Constraints Revisited," Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series, pp. 343–350 (August 1993, Anaheim, Calif.). Edited by James T. Kajiya. ISBN 0-201-58889-7.
20. J. O'Brien and J. Hodges, "Graphical Modeling and Animation of Brittle Fracture," Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, pp. 137–146 (August 1999, Los Angeles, Calif.). Addison-Wesley Longman. Edited by Alyn Rockwood. ISBN 0-20148-560-5.
21. W. Potts, "The Chorus Line Hypothesis of Maneuver Coordination in Avian Flocks," *Nature,* Vol. 309, May 1984, pp. 344–345.
22. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing,* Cambridge University Press, Cambridge, 1986.
23. W. Reeves, "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects," *Computer Graphics* (Proceedings of SIGGRAPH 83), 17 (3), pp. 359–376 (July 1983, Detroit, Mich.).
24. W. Reeves and R. Blau, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 31–40 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
25. C. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics* (Proceedings of SIGGRAPH 87), 21 (4), pp. 25–34 (July 1987, Anaheim, Calif.). Edited by Maureen C. Stone.
26. C. Reynolds, "Not Bumping into Things," *Physically Based Modeling Course Notes,* SIGGRAPH 88 (August 1988, Atlanta, Ga.), pp. G-1–G-13.
27. E. Ribble, "Synthesis of Human Skeletal Motion and the Design of a Special-Purpose Processor for Real-Time Animation of Human and Animal Figure Motion," Master's thesis, Ohio State University, Columbus, Ohio, June 1982.
28. K. Sims, "Particle Animation and Rendering Using Data Parallel Computation," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 405–414 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
29. D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture," *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 269–278 (August 1988, Atlanta, Ga.). Edited by John Dill.
30. D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically Deformable Models," *Computer Graphics* (Proceedings of SIGGRAPH 87), 21 (4), pp. 205–214 (July 1987, Anaheim, Calif.). Edited by Maureen C. Stone.
31. D. Terzopoulos and A. Witkin, "Physically Based Models with Rigid and Deformable Components," *IEEE Computer Graphics and Applications,* November 1988, pp. 41–51.
32. A. Witkin, K. Fleischer, and A. Barr, "Energy Constraints on Parameterized Models," *Computer Graphics* (Proceedings of SIGGRAPH 87), 21 (4), pp. 225–232 (July 1987, Anaheim, Calif.). Edited by Maureen C. Stone.

33. A. Witkin and M. Kass, “Spacetime Constraints,” *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 159–168 (August 1988, Atlanta, Ga.). Edited by John Dill.
34. D. Zeltzer, “Task Level Graphical Simulation: Abstraction, Representation and Control,” in *Making Them Move,* edited by N. Badler, B. Barsky, and D. Zeltzer, Morgan Kaufmann, San Francisco, 1991.

CHAPTER 5

# Natural Phenomena

Among the most difficult objects to model and animate computationally are those that are not defined by a static, rigid, topologically simple structure. Many of these complex forms are found in nature. They present especially difficult challenges for those intent on controlling their motion. Of the natural phenomena discussed in this chapter, plants are the only ones with a well-defined surface; the complexity of modeling them derives from their branching structure and their growth process. Fire, smoke, and clouds are gaseous phenomena that have no well-defined surface to model. They are inherently volumetric models, although surface-based techniques have been applied with limited success. Water, when relatively still, has a well-defined surface; however, it changes its shape as it moves. In the case of ocean waves, features on the water's surface move, but the water itself does not travel. The simple surface topology can become arbitrarily complex when the water becomes turbulent. Splashing, foaming, and breaking waves are complex processes best modeled by particle systems and volumetric techniques, but these techniques are inefficient in nonturbulent situations. In addition, water can travel from one place to another, form streams, split into separate pools, and collect again. In modeling these phenomena for purposes of computer graphics, programmers always make simplifying assumptions in order to limit the computational complexity and model only those features of the physical processes that are important visually.

Many of the time-varying models described in this chapter (and the next) represent work that is state of the art. It is not the objective here to cover all aspects of recent research. The basic models are covered, with only brief reference to the more advanced algorithms.

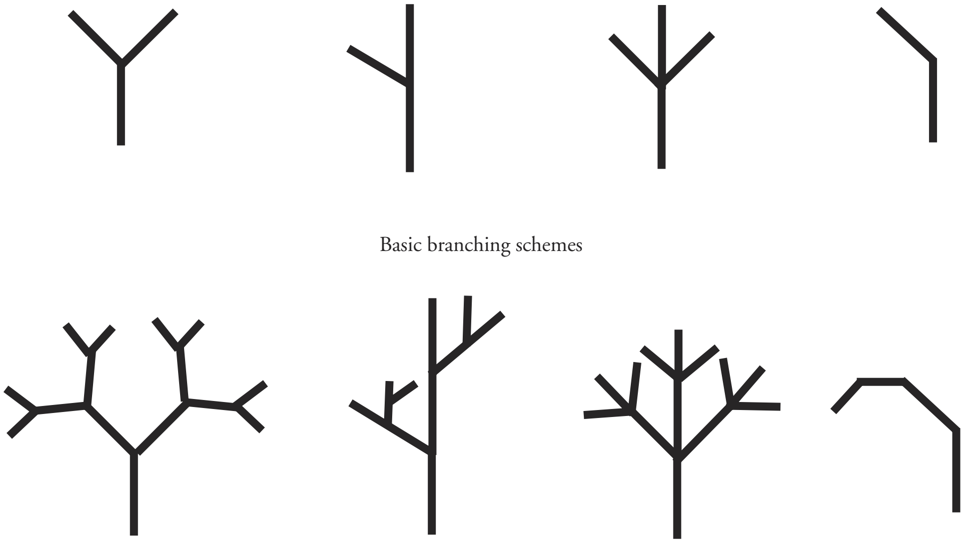
## 5.1 Plants

The modeling and animation of plants represent an interesting and challenging area for computer animation. Plants seem to exhibit arbitrary complexity while possessing a constrained branching structure. They grow from a single source point, developing a branching structure over time while the individual structural elements elongate. Plants have been modeled using particle systems, fractals, and L-systems. There has been much work on modeling the static representations of various plants (e.g., [1] [4] [20] [30] [34] [36] [38]). The intent here is not to delve into the botanically correct modeling of particular plants but rather to explain those aspects of modeling plants that make the animation of the growth process challenging. The representational issues of synthetic plants are discussed in just enough detail to uncover these aspects. Prusinkiewicz and Lindenmayer [33] [35] provide more information on all aspects of modeling and animating plants.

The topology[1] of a plant is characterized by a recursive branching structure. To this extent, plants share with fractals the characteristics of self-similarity under scale. The two-dimensional branching structures typically of interest are shown in Figure 5.1. The three-dimensional branching structures are analogous.

An encoding of the branching structure of a given plant is one of the objectives of plant modeling. Plants are immensely varied, yet most share many common characteristics. These shared characteristics allow efficient representations to be formed by abstracting out the features that are common to plants of interest. But the representation of the static structure of a mature plant is only part of the story. A plant, being a living thing, is subject to changes due to growth. The modeling and animation of the growth process are the subject of this section.

[1] The term *topology,* as applied to describing the form of plants, refers to the number and arrangement of convex regions of the plant delineated by concave areas of attachment to other convex regions.



Basic branching schemes

Structures resulting from repeated application of a single branching scheme

**Figure 5.1** Branching structures of interest in two dimensions

### 5.1.1 A Little Bit of Botany

Botany is, of course, useful when trying to model and animate realistic-looking plants. For computer graphics and animation, it is only useful to the extent that it addresses the visual characteristics of the plant. Thus, the structural components and surface elements of plants are briefly reviewed here. Simplifications are made so as to highlight the information most relevant for computer graphics modeling and animation.

The structural components of plants are *stems, roots, buds, leaves,* and *flowers*. Roots are typically not of interest when modeling the visual aspects of plants and have not been incorporated into plant models. Most plants of interest in visualization have a definite branching structure. Such plants are either *herbaceous* or *woody*. The latter are larger plants whose branches are heavier and more structurally independent. The branches of woody plants tend to interfere and compete with one another. They are also more subject to the effects of wind, gravity, and sunlight. Herbaceous plants are smaller, lighter plants, such as ferns and mosses, whose branching patterns tend to be more regular and less subject to environmental effects.

Stems are usually above ground, grow upward, and bear leaves. The leaves are attached in a regular pattern at nodes along the stem. The portions of the stem between the nodes are called *internodes. Branching* is the production of subordinate stems from a main stem (the *axis*). Branches can be formed by the main stem bifurcating into two equally growing stems (*dichotomous*), or they can be formed when a stem grows laterally from the main axis while the main axis continues to grow in its original direction (*monopodial*).

Buds are the embryonic state of stems, leaves, and flowers; they are classified as either *vegetative* or *flower buds.* Flower buds develop into flowers, whereas vegetative buds develop into stems or leaves. A bud at the end of a stem is called a *terminal* bud; a bud that grows along a stem is called a *lateral* bud. Not all buds develop; non-developing buds are referred to as *dormant.* Sometimes in woody plants, dormant buds will suddenly become active, producing young growth in an old-growth area of a tree.

Leaves grow from buds. They are arranged on a stem in one of three ways: *alternate, opposite,* or *whorled. Alternate* means that the leaves shoot off one side of a stem and then off the other side in an alternating pattern. *Opposite* means that a pair of leaves shoot off the stem at the same point, but on opposite sides, of the stem. *Whorled* means that three or more leaves radiate from a node.

Growth of a cell in a plant has four main influences: *lineage, cellular descent, tropisms,* and *obstacles. Lineage* refers to growth controlled by the age of the cell. *Cellular descent* refers to the passing of nutrients and hormones from adjacent cells. Growth controlled exclusively by lineage would result in older cells always being larger than younger cells. Cellular descent can be bidirectional, depending on the growth processes and reactions to environmental influences occurring in the plant. Thus, cellular descent is responsible for the ends of some plants growing more than the interior sections. Plant hormones are specialized chemicals produced by plants. These are the main internal factors that control the plant's growth and development. Hormones are produced in specific parts of the plants and are transported to others. The same hormone may either promote or inhibit growth, depending on the cells it interacts with.

Tropism responses are an important class of external influences that change the direction of a plant's growth. These include *phototropism,* the bending of a stem toward light, and *geotropism,* the response of a stem or root to gravity. Physical obstacles also affect the shape and growth of plants. Collision detection and response can be calculated for temporary changes in the plant's shape. Permanent changes in the growth patterns can occur when such forces are present for extended periods.

## 5.1.2 L-Systems

### D0L-Systems

*L-systems* are parallel rewriting systems. They were conceived as mathematical models of plant development by the biologist Aristid Lindenmayer (the *L* in *L-systems*). The simplest class of L-system is deterministic and context-free (as in Figure 5.2); it is called a *D0L-system.*[2] A D0L-system is a set of *production rules* of the form $\alpha_i \rightarrow \beta_i$, in which $\alpha_i$, the *predecessor,* is a single symbol and $\beta_i$, the *successor,* is a sequence of symbols. In deterministic L-systems, $\alpha_i$ occurs only once on the left-hand side of a production rule. A sequence of one or more symbols is given as the initial string, or *axiom.* A production rule can be applied to the string if its left-hand side occurs in the string. The effect of applying a production rule to a string means that the occurrence of $\alpha_i$ in the string is rewritten as $\beta_i$. Production rules are applied in parallel to the initial string. This replacement happens in parallel for all occurrences of any left-hand side of a production in the string. Symbols of the string that are not on the left-hand side of any production rule are assumed to be operated on by the identity production rule, $\alpha_i \rightarrow \alpha_i$. The parallel application of the production rules produces a new string. The production rules are then applied again to the new string. This happens iteratively until no occurrences of a left-hand side of a production rule occur in the string. Sample production rules and the string they generate are shown in Figure 5.2.

### Geometric Interpretation of L-Systems

The strings produced by L-systems are just that—strings. To produce images from such strings, one must interpret them geometrically. Two common ways of doing this are *geometric replacement* and *turtle graphics.* In geometric replacement, each symbol of a string is replaced by a geometric element. For example, the string XXTTXX can be interpreted by replacing each occurrence of X with a straight line segment and each occurrence of T with a V shape so that the top of the V aligns with the endpoints of the geometric elements on either side of it. See Figure 5.3.

| | | |
|---|---|---|
| S -> ABA | S | ← axiom |
| A -> XX | ABA | |
| B -> TT | XXTTXX | |
| Production rules | String sequence | |

**Figure 5.2** Simple D0L-system and the sequence of strings it generates

[2] The *D* in *D0L* clearly stands for *deterministic;* the *0* indicates, as is more fully explained later, that the productions are context-free.

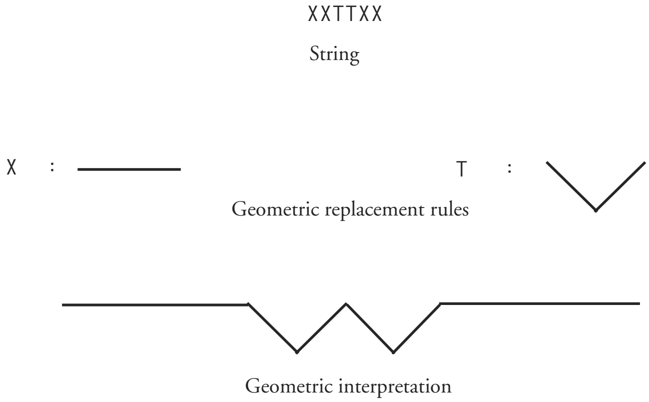


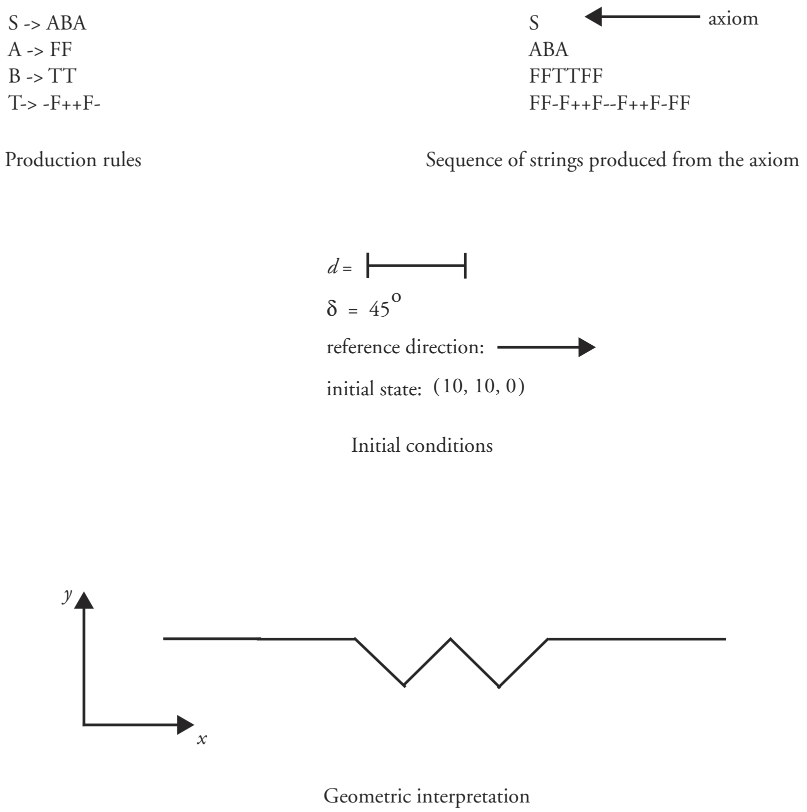
**Figure 5.3** Geometric interpretation of a simple string

In turtle graphics, a geometry is produced from the string by interpreting the symbols of the string as drawing commands given to a simple cursor called a turtle. The basic idea of turtle graphics interpretation, taken from Prusinkiewicz and Lindenmayer [35], uses the symbols from Table 5.1 to control the turtle. The state of the turtle at any given time is expressed as by a triple ($x$, $y$, $\alpha$), where $x$ and $y$ give its coordinates in a two-dimensional space and $\alpha$ gives the direction it is pointing relative to some given reference direction (here, a positive angle is measured counterclockwise from the reference direction). The values $d$ and $\delta$ are user-specified system parameters and are the linear and rotational step sizes, respectively.

Given the reference direction, the initial state of the turtle ($x_0$, $y_0$, $\alpha_0$), and the parameters $d$ and $\delta$, the user can generate the turtle interpretation of a string containing the symbols of Table 5.1(Figure 5.4).

**Table 5.1** Turtle Graphics Commands

| Symbol | Turtle Graphic Interpretation |
|---|---|
| F | Move forward a distance $d$ while drawing a line. Its state will change from $(x, y, \alpha)$ to $(x + d \cdot \cos\alpha, y + d \cdot \sin\alpha, \alpha)$. |
| f | Move forward a distance $d$ without drawing a line. Its state will change as above. |
| + | Turn left by an angle $\delta$. Its state will change from $(x, y, \alpha)$ to $(x, y, \alpha + \delta)$. |
| – | Turn right by an angle $\delta$. Its state will change from $(x, y, \alpha)$ to $(x, y, \alpha - \delta)$. |



S -> ABA
A -> FF
B -> TT
T-> -F++F-

Production rules

S
ABA
FFTTFF
FF-F++F--F++F-FF

Sequence of strings produced from the axiom

$d =$

$\delta = 45^{o}$

reference direction:

initial state: (10, 10, 0)

Initial conditions

Geometric interpretation

**Figure 5.4** Turtle graphic interpretation of a string generated by an L-system

## Bracketed L-Systems

The D0L-systems described above are inherently linear, and the graphical interpretations reflect this. To represent the branching structure of plants, one introduces a mechanism to represent multiple segments attached at the end of a single segment. In *bracketed L-systems,* brackets are used to mark the beginning and the end of additional offshoots from the main lineage. The turtle graphics interpretation of the brackets is given in Table 5.2. A stack of turtle graphic states is used, and the brackets push and pop states onto and off this stack. The state is defined by the current position and orientation of the turtle. This allows branching from a stem to be represented. Further, because a stack is used, it allows an arbitrarily deep branching structure.

Figure 5.5 shows some production rules. The production rules are context-free and *nondeterministic.* They are context-free because the left-hand side of the production rule does not contain any context for the predecessor symbol. They are

**Table 5.2** Turtle Graphic Interpretation of Brackets

| Symbol | Turtle Graphic Interpretation |
|---|---|
| [ | Push the current state of the turtle onto the stack |
| ] | Pop the top of the state stack and make it the current state |

$$
\begin{aligned}
&S \Rightarrow FAF\\
&A \Rightarrow [+FBF]\\
&A \Rightarrow F\\
&B \Rightarrow [-FBF]\\
&B \Rightarrow F
\end{aligned}
$$

**Figure 5.5** Nondeterministic, context-free production rules

nondeterministic because there are multiple rules with identical left-hand sides; one is chosen at random from the set whenever the left-hand side is to be replaced. In this set of rules, $S$ is the start symbol, and $A$ and $B$ represent a location of possible branching; $A$ branches to the left and $B$ to the right. The production stops when all symbols have changed into ones that have a turtle graphic interpretation.

Figure 5.6 shows some possible terminal strings and the corresponding graphics produced by the turtle interpretation. An added feature of this turtle interpretation of the bracketed L-system is the reduction of the size of the drawing step by one-half for each branching level, where *branching level* is defined by the current number of states on the stack. The bracketed L-system can be expanded to include attribute symbols that explicitly control line length, line width, color, and so on [35] and that are considered part of the state.

Not only does this representation admit *database amplification* [41], but the expansion of the start symbol into a terminal string parallels the topological growth process of the plants. In some sense, the sequence of strings that progress to the final string of all turtle graphic symbols represents the growth of the plant at discrete events in its evolution (see Figure 5.7). This addresses one of the animation issues with respect to plants—that of animating the development of the branching structure. However, the gradual appearance and subsequent elongation of elements must also be addressed if a growing structure is to be animated in a reasonable manner.

## Stochastic L-Systems

The previous section introduced nondeterminism into the concept of L-systems, but the method used to select the possible applicable productions for a given symbol was not addressed. *Stochastic L-systems* assign a user-specified probability to

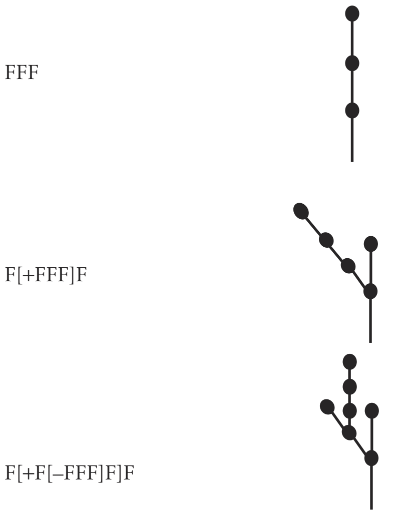


**Figure 5.6** Some possible terminal strings

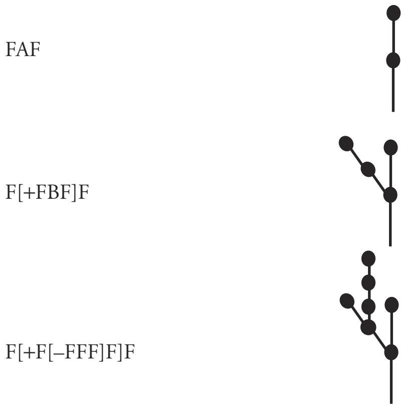


**Figure 5.7** Sequence of strings produced by bracketed L-system

each production so that the probabilities assigned to productions with the same left-hand side sum to one. These probabilities indicate how likely it is that the production will be applied to the symbol on a symbol-by-symbol basis.

The productions of Figure 5.5 might be assigned the probabilities shown in Figure 5.8. These probabilities will control how likely a production will be to form a branch at each possible branching point. In this example, left branches are very likely to form, while right branches are somewhat unlikely. However, any arbitrarily complex branching structure has a nonzero probability of occurring. Using such stochastic (nondeterministic) L-systems, one can set up an L-system that produces a wide variety of branching structures that still exhibit some family-like similarity [35].

$$
\begin{aligned}
&S_{1.0} \Rightarrow FAF\\
&A_{0.8} \Rightarrow [+FBF]\\
&A_{0.2} \Rightarrow F\\
&B_{0.4} \Rightarrow [-FBF]\\
&B_{0.6} \Rightarrow F
\end{aligned}
$$

**Figure 5.8** Stochastic L-system

$S \Rightarrow FAT$
$A > T \Rightarrow [+FBF]$
$A > F \Rightarrow F$
$B \Rightarrow [-FAF]$
$T \Rightarrow F$

Production rules

S
FAT
F[+FBF]F
F[+F[–FAF]F]F

String sequence

**Figure 5.9** Context-sensitive L-system production rules

### Context-Free versus Context-Sensitive

So far, only context-free L-systems have been presented. *Context-sensitive L-systems* add the ability to specify a context, in which the left-hand side (the predecessor symbol) must appear in order for the production rule to be applicable. For example, in the deterministic productions of Figure 5.9,[3] the symbol A has different productions depending on the context in which it appears. The context-sensitive productions shown in the figure have a single right-side context symbol in two of the productions. This concept can be extended to *n* left-side context symbols and *m* right-side context symbols in the productions, called (*n, m*)*L-systems,* and, of course, they are compatible with nondeterministic L-systems. In (*n, m*)L-systems, productions with fewer than *n* context symbols on the left and *m* on the right are allowable. Productions with shorter contexts are usually given precedence over productions with longer contexts when they are both applicable to the same symbol. If the context is one-sided, then L-systems are referred to as *n*L-systems, where *n* is the number of context symbols and the side of the context is specified independently.

## 5.1.3 Animating Plant Growth

There are three types of animation in plants. One type is the flexible movement of an otherwise static structure, for example, a plant being subjected to a high wind. Such motion is an example of a flexible body reacting to external forces and is not dealt with in this section. The other types of animation are particular to plants and involve the modeling of the growth process.

The two aspects of the growth process are (1) changes in topology that occur during growth and (2) the elongation of existing structures. The topological changes are captured by the L-systems already described. They occur as discrete

[3] In the notation used here, as it is in the book by Prusinkiewicz and Lindenmayer [35], the predecessor is the symbol on the "greater than" side of the inequality symbols. This is used so that the antecedent can be visually identified when using either left or right contexts.

events in time and are modeled by the application of a production that encapsulates a branching structure, as in $A \Rightarrow F[+F]B$.

Elongation can be modeled by productions of the form $F \Rightarrow FF$. The problem with this approach is that growth is chunked into units equal to the length of the drawing primitive represented by $F$. If $F$ represents the smallest unit of growth, then an internode segment can be made to grow arbitrarily long. But the production rule $F \Rightarrow FF$ lacks termination criteria for the growth process. Additional drawing symbols can be introduced to represent successive steps in the elongation process, resulting in a series of productions $F_0 \Rightarrow F_1$, $F_1 \Rightarrow F_2$, $F_2 \Rightarrow F_3$, $F_3 \Rightarrow F_4$, and so on. Each symbol would represent a drawing operation of a different length. However, if the elongation process is to be modeled in, say, one hundred time steps, then approximately one hundred symbols and productions are required. To avoid this proliferation of symbols and productions, the user can represent the length of the drawing operation parametrically with the drawing symbol in *parametric L-systems.*

### Parametric L-Systems

In *parametric L-systems*, symbols can have one or more parameters associated with them. These parameters can be set and modified by productions of the L-system. In addition, optional conditional terms can be associated with productions. The conditional expressions are in terms of parametric values. The production is applicable only if its associated condition is met. In the simple example of Figure 5.10, the symbol A has a parameter t associated with it. The productions create the symbol A with a parameter value of 0.0 and then increase the parametric value in increments of 0.01 until it reaches 1.0. At this point the symbol turns into an F.

Context-sensitive productions can be combined with parametric systems to model the passing of information along a system of symbols. Consider the production of Figure 5.11. In this production, there is a single context symbol on both sides of the left-hand symbol that is to be changed. These productions allow for the relatively easy representation of such processes as passing nutrients along the stem of a plant.

### Timed L-Systems

*Timed L-systems* add two more concepts to L-systems: a *global time variable,* which is accessible to all productions and which helps control the evolution of the string;

```
S               =>  A(0)
A(t)            =>  A(t+0.01)
A(t) : t>=1.0   =>  F
```

**Figure 5.10** Simple parametric L-system

```
A(t0)<A(t1)>A(t2): t2>t1 & t1>t0 => A(t1+0.01)
```

**Figure 5.11** Parametric, context-sensitive L-system production

and a *local age value*, $\tau_i$, associated with each letter $\mu_i$. Timed L-system productions are of the form shown in Equation 5.1. By this production, the letter $\mu_0$ has a *terminal age* of $\beta_0$ assigned to it. The terminal age must be uniquely assigned to a symbol. Also by this production, each symbol $\mu_i$ has an *initial age* of $\alpha_i$ assigned to it. The terminal age assigned to a symbol $\mu_i$ must be larger than its initial age so that its *lifetime*, $\beta_i - \alpha_i$, is positive.

$$(\mu_0, \beta_0) \Rightarrow ((\mu_1, \alpha_1), (\mu_2, \alpha_2), \ldots, (\mu_n, \alpha_n)) \qquad \textbf{(Eq. 5.1)}$$

A timed production can be applied to a matching symbol when that symbol's terminal age is reached. When a new symbol is generated by a production, it is commonly initialized with an age of zero. As global time progresses from that point, the local age of the variable increases until its terminal age is reached, at which point a production is applied to it and it is replaced by new symbols.

A string can be derived from an axiom by jumping from terminal age to terminal age. At any point in time, the production to be applied first is the one whose predecessor symbol has the smallest difference between its terminal age and its local age. Each symbol appearing in a string has a local age less than its terminal age. The geometric interpretation of each symbol is potentially based on the local age of that symbol. Thus, the appearance of buds and stems can be modeled according to their local age.

In the simple example of Figure 5.12, the symbol A can be thought of as a plant seed; S can be thought of as an internode stem segment; and B can be thought of as a bud that turns into the stem of a branch. After three units of time the seed becomes a stem segment, a lateral bud, and another stem segment. After two more time units, the bud develops into a branching stem segment.

### Interacting with the Environment

The environment can influence plant growth in many ways. There are local influences such as physical obstacles, including other plants and parts of the plant itself. There are global influences such as amount of sunlight, length of day, gravity, and wind. But even these global influences are felt locally. The wind can be blocked from part of the plant, as can the sun. And even gravity can have more of an effect on an unsupported limb than on a supported part of a vine.The nutrients and moisture in the soil affect growth. These are transported throughout the plant, more or less effectively depending on local damage to the plant structure.

```
axiom: (A,0)

(A,3) => (S,0) [+ (B,0)] (S,0)
(B,2) => (S,0)
```

**Figure 5.12** Simple timed L-system

Mech and Prusinkiewicz [27] describe a framework for the modeling and animation of plants that bidirectionally interacts with the environment. They describe *open L-systems,* in which communication terms of the form ?$E(x_1, x_2, \ldots, x_m)$ are used to transmit information as well as request information from the environment. In turtle graphic interpretation of an L-system string, the string is scanned left to right. As communication terms are encountered, information is transmitted between the environment and the plant model. The exact form of the communication is defined in an auxiliary specification file so that only relevant information is transmitted. Information about the environment relevant to the plant model includes distribution of nutrients, direction of sunlight, and length of day. The state of the plant model can be influenced as a result of this information and can be used to change the rate of elongation as well as to control the creation of new offshoots. Information from the plant useful to the environmental model includes use of nutrients and shade formation, which, in turn, can influence other plant models in the environment.

### 5.1.4 Summary

L-systems, in all the variations, are an interesting and powerful modeling tool. Originally intended only as a static modeling tool, L-systems can be used to model the time-varying behavior of plantlike growth. Because of the iterative nature of string development, the topological changes of plant growth can be successfully captured by L-systems. By adding parameters, time variables, and communication modules, one can model other aspects of plant growth. Most recently, Deussen et al. [9] have used open L-systems to model plant ecosystems.

## 5.2 Water

Water presents a particular challenge for computer animation because its appearance and motion take various forms [15] [17] [42] [48]. Water can be modeled as a still, rigid-looking surface to which ripples can be added as display attributes by perturbing the surface normal as in bump mapping [3]. Alternatively, water can be modeled as a smoothly rolling height field in which time-varying ripples are incorporated into the geometry of the surface [25]. In ocean waves, it is assumed that there is no transport of water even though the waves travel along the surface in forms that vary from sinusoidal to cycloidal[4] [16] [32]. Breaking, foaming, and splashing of the waves are added on top of the model in a postprocessing step [16]

[4] A *cycloid* is the curve traced out by a point on the perimeter of a rolling disk.

[32]. The transport of water from one location to another adds more computational complexity to the modeling problem [23].

## 5.2.1 Still Waters and Small-Amplitude Waves

The simplest way to model water is merely to assign the color blue to anything below a given height. If the *y*-axis is “up,” then color any pixel blue in which the world space coordinate of the corresponding visible surface has a *y*-value less than some given constant. This creates the illusion of still water at a consistent “sea level.” It is sufficient for placid lakes and puddles of standing water. Equivalently, a flat blue plane perpendicular to the *y*-axis and at the height of the water can be used to represent the water’s surface. These models, of course, do not produce any animation of the water.

Normal vector perturbation (essentially the approach employed in bump mapping) can be used to simulate the appearance of small amplitude waves on an otherwise still body of water. To perturb the normal, one or more simple sinusoidal functions are used to modify the direction of the surface’s normal vector. The functions are parameterized in terms of a single variable, usually relating to distance from a *source point*. It is not necessarily the case that the wave starts with zero amplitude at the source. When standing waves in a large body of water are modeled, each function usually has a constant amplitude. The wave crests can be linear, in which case all the waves generated by a particular function travel in a uniform direction, or the wave crests can radiate from a single user-specified or randomly generated source point. Linear wave crests tend to form self-replicating patterns when viewed from a distance. For a different effect, radially symmetrical functions that help to break up these global patterns can be used. Radial functions also simulate the effect of a thrown pebble or raindrop hitting the water (Figure 5.13). The time-varying height for a point at which the wave begins at time zero is a function of the amplitude and wavelength of the wave. (Figure 5.14). Combining the two, Figure 5.15 shows the height of a point at some distance *d* from the start of the wave. This is a two-dimensional function relative to a point at which the function is zero at time zero. This function can be rotated and translated so that it is positioned and oriented appropriately in world space. Once the height function for a given point is defined, the normal to the point at any instance in time can be determined by computing the tangent vector and forming the vector perpendicular to it, as shown in Figure 5.16.

Superimposing multiple sinusoidal functions of different amplitude and with various source points (in the radial case) or directions (in the linear case) can generate interesting patterns of overlapping ripples. Typically, the higher the frequency of the wave component, the lower the amplitude (Figure 5.17). Notice

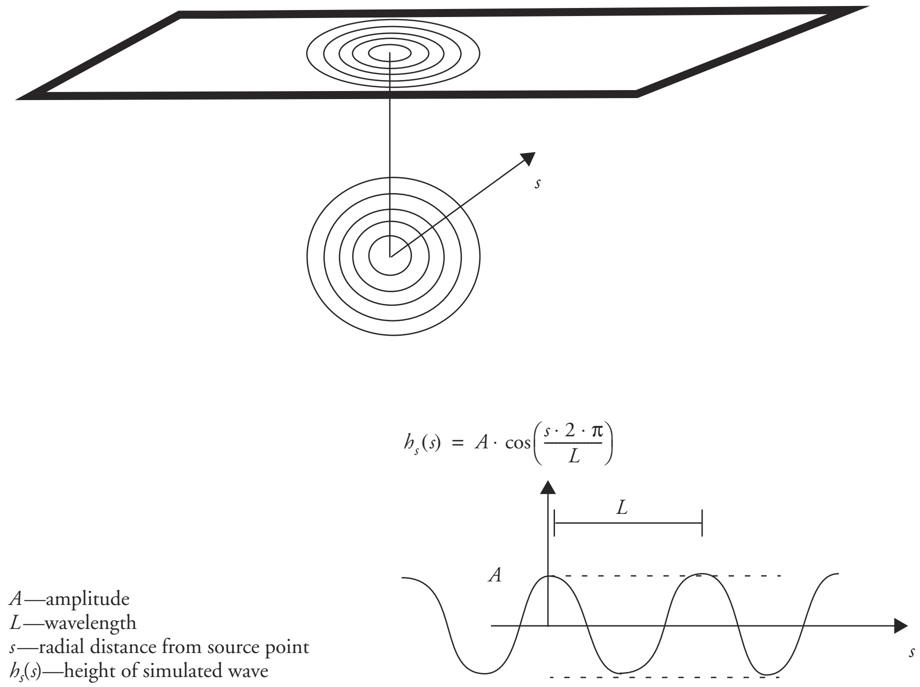

s
$h_s(s) = A \cdot \cos\left(\frac{s \cdot 2 \cdot \pi}{L}\right)$
L
A
s
A—amplitude
L—wavelength
s—radial distance from source point
$h_s(s)$—height of simulated wave

**Figure 5.13** Radially symmetric standing wave

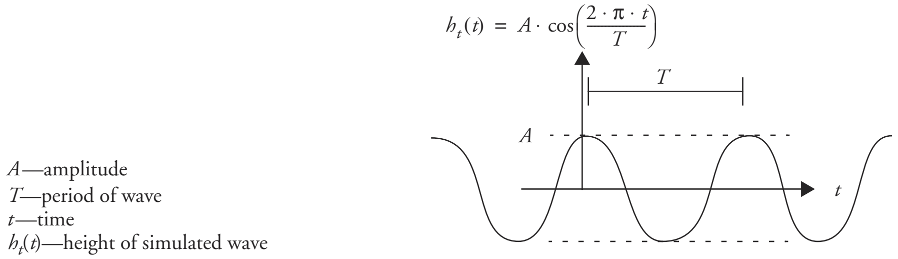

$h_t(t) = A \cdot \cos\left(\frac{2 \cdot \pi \cdot t}{T}\right)$
T
A
t
A—amplitude
T—period of wave
t—time
$h_t(t)$—height of simulated wave

**Figure 5.14** Time-varying height of a stationary point

that these do not change the geometry of the surface used to represent the water (e.g., a flat blue plane) but are used only to change the shading properties. Also notice that it must be a time-varying function that propagates the wave along the surface.

Calculating the normals without changing the actual surface creates the illusion of waves on the surface of the water. However, whenever the water meets a

$$h_P(s,\,t) \;=\; A \cdot \cos\left(\left(\frac{s}{L} + \frac{t}{T}\right) \cdot 2 \cdot \pi\right)$$

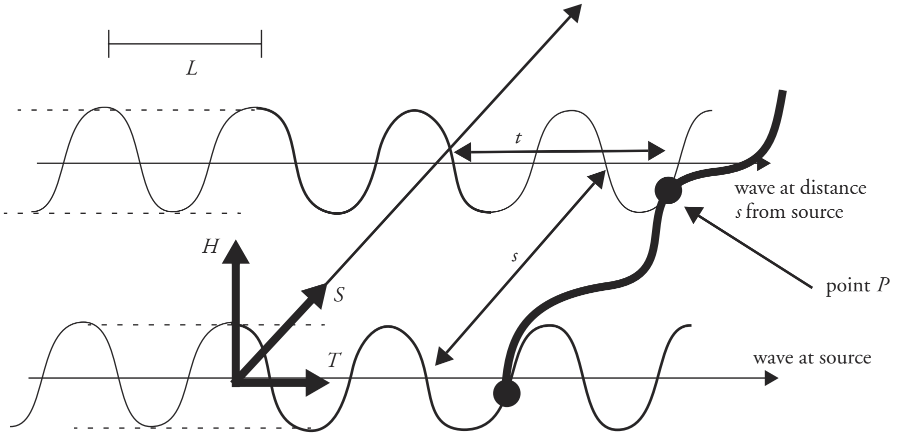
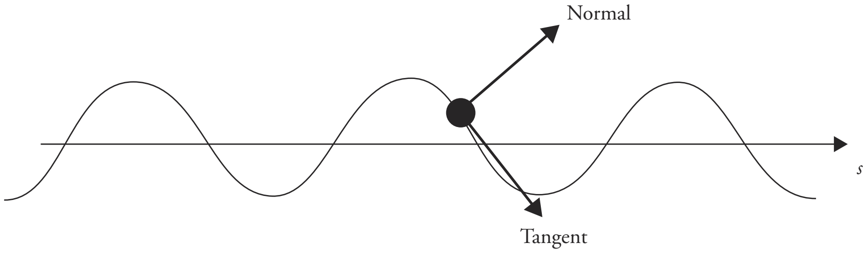


**Figure 5.15** Time-varying function at point $P$



$$h_P(s,\,t) \;=\; A \cdot \cos\left(\left(\frac{s}{L} + \frac{t}{T}\right) \cdot 2 \cdot \pi\right)$$

$$\frac{d}{ds}h_p(s,\,t) \;=\; -\left(A \cdot \frac{2 \cdot \pi}{L} \cdot \sin\left(\left(\frac{s}{L} + \frac{t}{T}\right) \cdot 2 \cdot \pi\right)\right)$$

$$\text{Tangent} \;=\; \left(1, \frac{d}{ds}h_p(s,\,t)\right)$$

$$\text{Normal} \;=\; \left(-\left(\frac{d}{ds}h_p(s,\,t)\right), 1\right)$$

**Figure 5.16** Normal vector for wave function

protruding surface, like a rock, the lack of surface displacement will be evident. See Figure 5.18 (Plate 3). The same approach used to calculate wave normals can be used to modify the height of the surface. A mesh of points can be used to model the surface of the water and the heights of the individual points can be controlled by the overlapping sinusoidal functions, as shown in Figure 5.16. Either a faceted

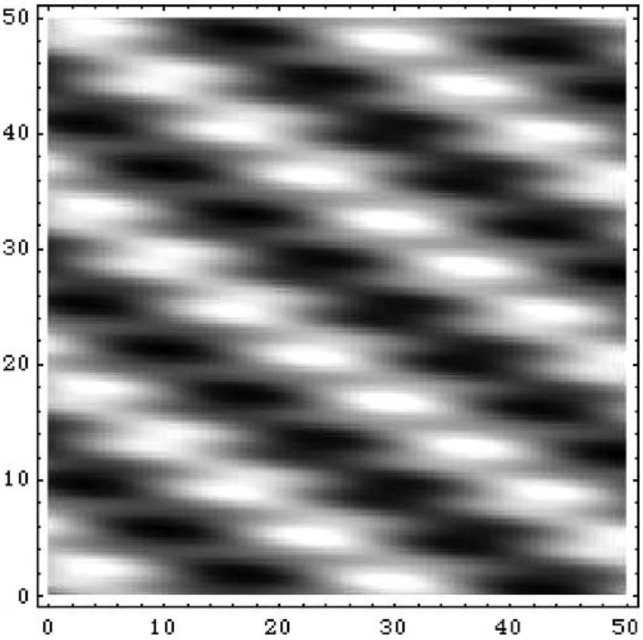


**Figure 5.17** Superimposed linear waves of various amplitudes and frequencies

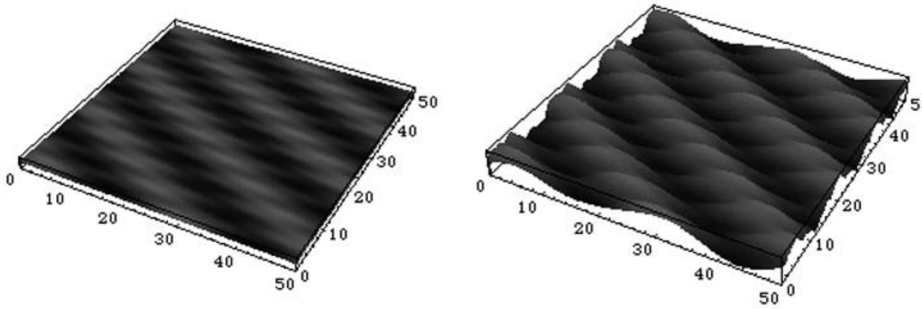


**Figure 5.18** Normal vector displacement versus height displacement

surface with smooth shading can be used or the points can be the control points of a higher-order surface such as a B-spline surface. The points must be sufficiently dense to sample the height function accurately enough for rendering. An option to reduce the density required is to use only the low-frequency, high-amplitude functions to control the height of the surface points and to include the high-frequency, low-amplitude functions to calculate the normals.

## 5.2.2 The Anatomy of Waves

A more sophisticated model must be used to model waves with greater realism, one that incorporates more of the physical effects that produce their appearance and behavior. Waves come in various frequencies, from tidal waves to capillary waves, which are created by wind passing over the surface of the water. The waves collectively called wind waves are those of most interest for visual effects. The sinusoidal form of a simple wave has already been described and is reviewed here in a more appropriate form for the equations that follow. In Equation 5.2, the function $f(s, t)$ describes the amplitude of the wave in which $s$ is the distance from the source point, $t$ is a point in time, $A$ is the maximum amplitude, $C$ is the propagation speed, and $L$ is the wavelength. The period of the wave, $T$, is the time it takes for one complete wave to pass a given point. The wavelength, period, and speed are related by the equation $C = L/T$.

$$f(x, t) = A \cdot \cos\left(\frac{2 \cdot \pi \cdot (s - (C \cdot t))}{L}\right) \tag{Eq. 5.2}$$

The motion of the wave is different from the motion of the water. The wave travels linearly across the surface of the water, while a particle of water moves in nearly a circular orbit (Figure 5.19). While riding the crest of the wave, the particle will move in the direction of the wave. As the wave passes and the particle drops into the trough between waves, it will travel in the reverse direction. The steepness, $S$, of the wave is represented by the term $H/L$.

Waves with a small steepness value have a basically sinusoidal shape. As the steepness value increases, the shape of the wave gradually changes into a sharply crested peak with flatter troughs. Mathematically, the shape approaches that of a cycloid.

In an idealized wave, there is no net transport of water. The particle of water completes one orbit in the time it takes for one complete cycle of the wave to pass. The average orbital speed of a particle of water is given by the circumference of the orbit, $\pi \cdot H$, divided by the time it takes to complete the orbit, $T$ (Equation 5.3).

$$Q_{\text{average}} = \frac{\pi \cdot H}{T} = \frac{\pi \cdot H \cdot C}{L} = \pi \cdot S \cdot C \tag{Eq. 5.3}$$

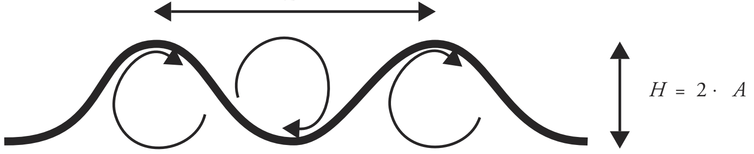


**Figure 5.19** Circular paths of particles of water subjected to waves

If the orbital speed, $Q$, of the water at the crest exceeds the speed of the wave, $C$, then the water will spill over the wave, resulting in a breaking wave. Because the average speed, $Q$, increases as the steepness, $S$, of the wave increases, this limits the steepness of a nonbreaking wave. The observed steepness of ocean waves, as reported by Peachey [32], is between 0.5 and 1.0.

A common simplification of the full computational fluid dynamics simulation of ocean waves is called the Airy model, and it relates the depth of the water, $d$, the propagation speed, $C$, and the wavelength of the wave, $L$ (Equation 5.4).

$$C = \sqrt{\frac{g}{\kappa} \cdot \tanh(\kappa \cdot d)} = \sqrt{\frac{g \cdot L}{2 \cdot \pi} \cdot \tanh\left(\frac{2 \cdot \pi \cdot d}{L}\right)}$$

$$L = C \cdot T \qquad \textbf{(Eq. 5.4)}$$

In Equation 5.4, g is the acceleration of a body due to gravity at sea level, 9.81 m/sec$^2$, and $\kappa = 2 \cdot \pi / L$ is the spatial equivalent of wave frequency. As the depth of the water increases, the function $\tanh(\kappa \cdot d)$ tends toward one, so $C$ approaches $g \cdot L/2 \cdot \pi$. As the depth decreases and approaches zero, $\tanh(\kappa \cdot d)$ approaches $\kappa \cdot d$, so $C$ approaches $\sqrt{g \cdot d}$. Peachey suggests using *deep* to mean $d \geq L/4$ and *shallow* to mean $d \leq L/20$.

As a wave approaches the shoreline at an angle, the part of the wave that approaches first will slow down as it encounters a shallower area. The wave will progressively slow down along its length as more of it encounters the shallow area. This will tend to straighten out the wave and is called *wave refraction*.

Interestingly, even though speed ($C$) and wavelength ($L$) of the wave are reduced as the wave enters shallow water, the period, $T$, of the wave remains the same and the amplitude, $A$ (and, equivalently, $H$), remains the same or increases. As a result, the orbital speed, $Q$ (Equation 5.3), of the water remains the same. Because orbital speed remains the same as the speed of the wave decreases, waves tend to break as they approach the shoreline because the speed of the water exceeds the speed of the wave. The breaking of a wave means that water particles break off from the wave surface as they are "thrown forward" beyond the front of the wave.

## 5.2.3 Modeling Ocean Waves

The description of modeling ocean waves presented here follows Peachey [32]. The ocean surface is represented as a height field, $y = f(x, z, t)$, where $(x, z)$ defines the two-dimensional ground plane, $t$ is time, and $y$ is the height. The wave function $f$ is a sum of various waveforms at different amplitudes (Equation 5.5).

$$f(x, z, t) = \sum_{i=1}^{n} A_i \cdot W_i(x, z, t) \qquad \textbf{(Eq. 5.5)}$$

The wave function, $W_i$, is formed as the composition of two functions: a wave profile, $w_i$; and a phase function, $\theta_i(x, z, t)$, according to Equation 5.6. This allows the description of the wave profile and phase function to be addressed separately.

$$W_i(x, z, t) = w_i(\text{fraction}[\theta_i(x, z, t)]) \qquad \text{(Eq. 5.6)}$$

Each waveform is described by its period, amplitude, source point, and direction. It is convenient to define each waveform, actually each phase function, as a linear rather than radial wave and to orient it so the wave is perpendicular to the *x*-axis and originates at the source point. The phase function is then a function only of the *x*-coordinate and can then be rotated and translated into position in world space.

Equation 5.7 gives the time dependence of the phase function. Thus, if the phase function is known for all points *x* (assuming the alignment of the waveform along the *x*-axis), then the phase function can be easily computed at any time at any position. If the depth of water is constant, the Airy model states that the wavelength and speed are also constant. In this case, the aligned phase function is given in Equation 5.8.

$$\theta_i(x, z, t) = \theta_i(x, y, t_0) - \frac{t - t_0}{T_i} \qquad \text{(Eq. 5.7)}$$

$$\theta_i(x, z) = \frac{x_i}{L_i} \qquad \text{(Eq. 5.8)}$$

However, if the depth of the water is variable, then $L_i$ is a function of depth and $\theta_i$ is the integral of the depth-dependent phase-change function from the origin to the point of interest (Equation 5.9). Numerical integration can be used to produce phase values at predetermined grid points. These grid points can be used to interpolate values within grid cells. Peachey [32] successfully uses bilinear interpolation to accomplish this.

$$\theta_i(x, z) = \int_0^x \theta_i'(u, z)\,du \qquad \text{(Eq. 5.9)}$$

The wave profile function, $w_i$, is a single-value periodic function of the fraction of the phase function (Equation 5.6) so that $w_i(u)$ is defined for $0.0 \le u < 1.0$. The values of the wave profile function range over the interval [–1, 1]. The wave profile function is designed so that its value is one at both ends of the interval (Equation 5.10).

$$w_i(0.0) = \lim_{u \to 1.0} w_i(u) = 1.0 \qquad \text{(Eq. 5.10)}$$

Linear interpolation can be used to model the changing profile of the wave according to steepness. Steepness (*H*/*L*) can be used to blend between a sinusoidal function (Equation 5.2) and a cycloidlike function (Equation 5.11) designed to resemble a sharp-crested wave profile. In addition, wave asymmetry is introduced as a function of the depth of the water to simulate effects observed in waves as they approach a coastline. The asymmetry interpolant, *k,* is defined as the ratio between the water depth, *d,* and deep-water wavelength, $L_i$. See Equation 5.12. When *k* is large, the wave profile is handled with no further modification. When *k* is small, *u* is raised to a power in order to shift its value toward the low end of the range between zero and one. This has the effect of stretching out the back of the wave and steepening the front of the wave as it approaches the shore.

$$w_i(u) = 8 \cdot |u - 1/2|^2 - 1 \qquad \text{(Eq. 5.11)}$$

$$L_i^{\text{deep}} = \frac{g \cdot T_i^2}{2 \cdot \pi}$$

$$k = \frac{d}{L_i^{\text{deep}}} \qquad \text{(Eq. 5.12)}$$

As the wave enters very shallow water, the amplitudes of the various wave components are reduced so the overall amplitude of the wave is kept from exceeding the depth of the water.

Spray and foam resulting from breaking waves and waves hitting obstacles can be simulated using a stochastic but controlled (e.g., Gaussian distribution) particle system. When the speed of the water, $Q_{\text{average}}$, exceeds the speed of the wave, *C,* then water spray leaves the surface of the wave and is thrown forward. Equation 5.3 indicates that this condition happens when $\pi \cdot S > 1.0$ or, equivalently, $S > 1.0/\pi$. Breaking waves are observed with steepness values less than this (around 0.1), which indicates that the water probably does not travel at a uniform orbital speed. Instead, the speed of the water at the top of the orbit is faster than at other points in the orbit. Thus, a user-specified spray-threshold steepness value can be used to trigger the particle system. The number of particles generated is based on the difference between the calculated wave steepness and the spray-threshold steepness.

For a wave hitting an obstacle, a particle system can be used to generate spray in the direction of reflection based on the incoming direction of the wave and the normal of the obstacle surface. A small number of particles are generated just before the moment of impact, are increased to a maximum number at the point of impact, and are then decreased as the wave passes the obstacle. As always, stochastic perturbation should be used to control both speed and direction.

## 5.2.4 Finding Its Way Downhill

One of the assumptions used to model ocean waves is that there is no transport of water. However, in many situations, such as a stream of water running downhill, it is useful to model how water travels from one location to another. In situations in which the water can be considered a height field and the motion assumed to be uniform through a vertical column of water, the vertical component of the velocity can be ignored. In such cases, differential equations can be used to simulate a wide range of convincing motion [23]. The Navier-Stokes equations (which describe flow through a volume) can be simplified to model the flow.

To develop the equations in two dimensions, the user parameterizes functions are in terms of distance $x$. Let $z = h(x)$ be the height of the water and $z = b(x)$ be the height of the ground at location $x$. The height of the water is $d(x) = h(x) - b(x)$. If one assumes that motion is uniform through a vertical column of water and that $v(x)$ is the velocity of a vertical column of water, then the shallow-water equations are as shown in Equation 5.13 and Equation 5.14, where $g$ is the gravitational acceleration. See Figure 5.20. Equation 5.13 considers the change in velocity of the water and relates its acceleration, the difference in adjacent velocities, and the acceleration due to gravity when adjacent columns of water are at different heights. Equation 5.14 considers the transport of water by relating the temporal change in the height of the vertical column of water with the spatial change in the amount of water moving.

$$\frac{\partial v}{\partial t} + v \cdot \frac{\partial v}{\partial x} + g \cdot \frac{\partial h}{\partial x} = 0 \qquad \textbf{(Eq. 5.13)}$$

$$\frac{\partial d}{\partial t} + \frac{\partial}{\partial x}(v \cdot d) = 0 \qquad \textbf{(Eq. 5.14)}$$

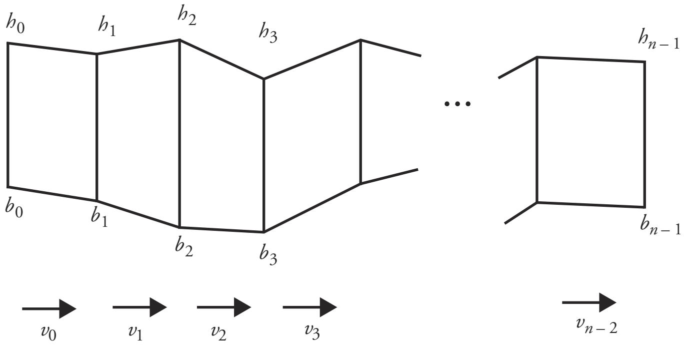


**Figure 5.20** Discrete two-dimensional representation of height field with water surface $h$, ground $b$, and horizontal water velocity $v$

These equations can be further simplified if the assumptions of small fluid velocity and slowly varying depth are used. The former assumption eliminates the second term of Equation 5.13, while the latter assumption implies that the term d can be removed from inside the derivative in Equation 5.14. These simplifications result in Equation 5.15 and Equation 5.16.

$$\frac{\partial v}{\partial t} + g \cdot \frac{\partial h}{\partial x} = 0 \tag{Eq. 5.15}$$

$$\frac{\partial d}{\partial t} + d \cdot \frac{\partial v}{\partial x} = 0 \tag{Eq. 5.16}$$

Differentiating Equation 5.15 with respect to *x* and Equation 5.16 with respect to *t* and substituting for the cross derivatives results in Equation 5.17. This is the one-dimensional wave equation with a wave velocity $\sqrt{g \cdot d}$. As Kass and Miller [23] note, this degree of simplification is probably not accurate enough for engineering applications.

$$\frac{\partial^2 h}{\partial t^2} = g \cdot d \cdot \frac{\partial^2 h}{\partial x^2} \tag{Eq. 5.17}$$

This partial differential equation is solved using finite differences. The discretization, as used by Kass and Miller, is set up as in Figure 5.20, with samples of *v* positioned halfway between the samples of *h*. The authors report a stable discretization, resulting in Equation 5.18 and Equation 5.19. Putting these two equations together results in Equation 5.20, which is the discrete version of Equation 5.17.

$$\frac{\partial h_i}{\partial t} = \left(\frac{d_{i-1} + d_i}{2 \cdot \Delta x}\right) \cdot v_{i-1} - \left(\frac{d_i + d_{i+1}}{2 \cdot \Delta x}\right) \cdot v_i \tag{Eq. 5.18}$$

$$\frac{\partial v_i}{\partial t} = \frac{-g \cdot (h_{i+1} - h_i)}{\Delta x} \tag{Eq. 5.19}$$

$$\frac{\partial^2 h_i}{\partial t^2} = -g \cdot \left(\frac{d_{i-1} + d_i}{2 \cdot (\Delta x)^2}\right) \cdot (h_i - h_{i-1}) + g \cdot \left(\frac{d_i + d_{i+1}}{2 \cdot (\Delta x)^2}\right) \cdot (h_{i+1} - h_i) \tag{Eq. 5.20}$$

Equation 5.20 states the relationship of the height of the water surface to the height's acceleration in time. This could be solved by using values of $h_i$ to compute the left-hand side and then using this value to update the next time step. As Kass and Miller report, however, this approach diverges quickly because of the sample spacing.

A first-order implicit numerical integration technique is used to provide a stable solution to Equation 5.20. Numerical integration uses current sample values to approximate derivatives. Explicit methods use approximated derivatives to update the current samples to their new values. Implicit integration techniques find the value whose derivative matches the discrete approximation of the current samples. Implicit techniques typically require more computation per step, but, because they are less likely to diverge significantly from the correct values, larger time steps can be taken, thus producing an overall savings.

Kass and Miller find that a first-order implicit method (Equation 5.21, Equation 5.22) is sufficient for this application. Using these equations to solve for $h(n)$ and substituting Equation 5.20 for the second derivative ($\ddot{h}(n)$) results in Equation 5.23.

$$\dot{h}(n) = \frac{h(n) - h(n-1)}{\Delta t} \qquad \text{(Eq. 5.21)}$$

$$\ddot{h}(n) = \frac{\dot{h}(n) - \dot{h}(n-1)}{\Delta t} \qquad \text{(Eq. 5.22)}$$

$$\begin{aligned} h_i(n) &= 2 \cdot h_i(n-1) - h_i(n-2) \\ &\quad -g \cdot (\Delta t)^2 \cdot \frac{d_{i-1} + d_i}{2 \cdot \Delta x^2} \cdot (h_i(n) - h_{i-1}(n)) \\ &\quad + g \cdot (\Delta t)^2 \cdot \frac{d_i + d_{i+1}}{2 \cdot \Delta x^2} \cdot (h_{i+1}(n) - h_i(n)) \end{aligned} \qquad \text{(Eq. 5.23)}$$

Assuming $d$ is constant during the iteration, the next value of $h$ can be calculated from previous values with the symmetric tridiagonal linear system represented by Equation 5.24, Equation 5.25, and Equation 5.26.

$$A \cdot h_i(n) = 2 \cdot h_i(n-1) + h_i(n-2) \qquad \text{(Eq. 5.24)}$$

$$A = \begin{bmatrix} e_0 & f_0 & 0 & 0 & 0 & 0 & 0 \\ f_0 & e_1 & f_1 & 0 & 0 & 0 & 0 \\ 0 & f_1 & e_2 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & \dots & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & e_{n-3} & f_{n-3} & 0 \\ 0 & 0 & 0 & 0 & f_{n-3} & e_{n-2} & f_{n-2} \\ 0 & 0 & 0 & 0 & 0 & f_{n-2} & e_{n-1} \end{bmatrix} \qquad \text{(Eq. 5.25)}$$

$$e_0 = 1 + g \cdot (\Delta t)^2 \cdot \left(\frac{d_0 + d_1}{2 \cdot (\Delta x)^2}\right)$$

$$e_i = 1 + g \cdot (\Delta t)^2 \cdot \left(\frac{d_{i-1} + 2 \cdot d_i + d_{i+1}}{2 \cdot (\Delta x)^2}\right) \qquad (0 < i < n - 1)$$

$$e_{n-1} = 1 + g \cdot (\Delta t)^2 \cdot \left(\frac{d_{n-2} + d_{n-1}}{2 \cdot (\Delta x)^2}\right)$$

$$f_i = -\left(g \cdot (\Delta t)^2 \cdot \left(\frac{d_i + d_{i+1}}{2 \cdot (\Delta x)^2}\right)\right) \tag{Eq. 5.26}$$

To simulate a viscous fluid, Equation 5.24 can be modified to incorporate a parameter that controls the viscosity, thus producing Equation 5.27. The parameter $\tau$ ranges between 0 and 1. When $\tau = 0$, Equation 5.27 reduces to Equation 5.24.

$$A \cdot h_i(n) = h_i(n-1) + (1 - \tau) \cdot (h_i(n-1) - h_i(n-2) \tag{Eq. 5.27}$$

Volume preservation can be compromised when $h_i < b_i$. To compensate for this, search for the connected areas of water ($h_j > b_j$ for $j = i, i + 1, \ldots, i + n$) at each iteration and compute the volume. If the volume changes from the last iteration, then distribute the difference among the elements of that connected region.

The algorithm for the two-dimensional case is shown in Figure 5.21.

### Three-Dimensional Case

Extending the algorithm to the three-dimensional case considers a two-dimensional height field. The computations are done by decoupling the two dimensions of the field. Each iteration is decomposed into two subiterations, one in the $x$-direction and one in the $y$-direction.

```
Specify h(t=0), h(t=1), and b
for j=2 ... n
     to simulate sinks and sources, modify appropriate h values
     calculate d from h(j-1) and b; if $h_i$< $b_i$, then $d_i$=0
     use Equation 5.11 (Equation 5.14) to calculate h(j) from h(j-1) and h(j-2)
     adjust the values of h to conserve volume (as discussed above)
     if $h_i$<$b_i$, set $h_i$(j) and $h_{i-1}$(j) to $b_i$-$\varepsilon$
```

**Figure 5.21** Two-dimensional algorithm for water transport

### 5.2.5 Summary

Animating all of the aspects of water is a difficult task because of water's ability to change shape over time. Great strides have been made in animating individual aspects of water such as standing waves, ocean waves, spray, and flowing water. An efficient approach to an integrated model of water remains a challenge.

## 5.3 Gaseous Phenomena

Modeling gaseous phenomena (smoke, clouds, fire) is particularly challenging because of their ethereal nature. Gas has no definite geometry, and, as a result, its modeling, rendering, and animating are often interrelated. In scientific terms, gas is usually lumped together with liquids, and their motions are commonly referred to as *fluid dynamics;* the equations used to model them are referred to as *computational fluid dynamics* (CFD). Gas is usually treated as *compressible,* meaning that density is spatially variable and computing the changes in density is part of the computational cost. Liquids are usually treated as *incompressible,* which means the density of the material remains constant. In fact, the equations in the previous section, on the transport of water, were derived from the CFD equations.

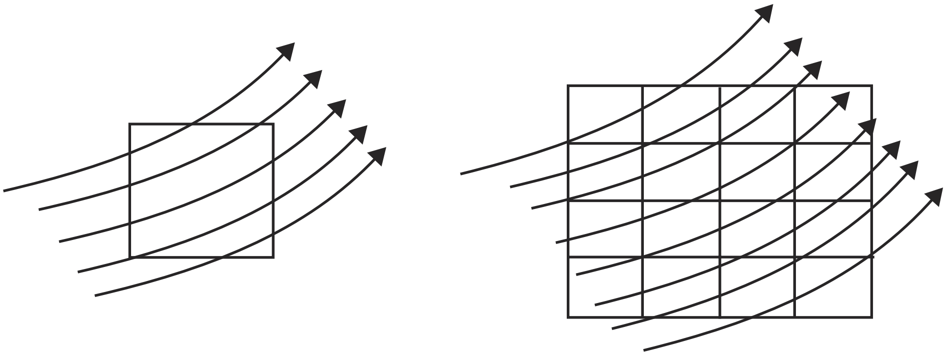
In a *steady state flow,* the motion attributes (e.g., velocity and acceleration) at any point in space are constant. Particles traveling through a steady state flow can be tracked similarly to how a space-curve can be traced out when the derivatives are known. *Vortices,* circular swirls of material, are important features in fluid dynamics. In steady state flow, vortices are attributes of space and are time-invariant. In time-varying flows, particles that carry a nonzero vortex strength can travel through the environment and can be used to modify the acceleration of other particles in the system by incorporating a distance-based force.

### 5.3.1 General Approaches to Modeling Gas

There are three approaches to modeling gas: grid-based methods (*Eulerian formulations*), particle-based methods (*Lagrangian formulations*), and hybrid methods. The approaches are illustrated here in two dimensions, but the extension to three dimensions should be obvious.

#### Grid-Based Method

The grid-based method decomposes space into individual cells, and the flow of the gas into and out of each cell is calculated (Figure 5.22). In this way, the density of gas in each cell is updated from time step to time step. The density in each cell is used to determine the visibility and illumination of the gas during rendering.



Gas flowing through an individual cell

Grid of cells

**Figure 5.22** Grid-based method

Attributes of the gas within a cell, such as velocity, acceleration, and density, can be used to track the gas as it travels from cell to cell.

The flow out of the cell can be computed based on the cell velocity, the size of the cell, and the cell density. The flow into a cell is determined by distributing the densities out of adjacent cells. External forces, such as wind and obstacles, are used to modify the acceleration of the particles within a cell.

The rendering phase uses standard volumetric graphics techniques to produce an image based on the densities projected onto the image plane. Illumination of a cell from a light source through the volume must also be incorporated into the display procedure.

The disadvantage of this approach is that if a static data structure for the cellular decomposition is used, the extent of the interesting space must be identified before the simulation takes place in order to initialize the cells that will be needed during the simulation of the gas phenomena. Alternatively, a dynamic data structure that adapts to the traversal of the gas through space could be used, but this increases overhead.

### Particle-Based Method

In the particle-based method, particles or globs of gas are tracked as they progress through space, often with a standard particle system approach (Figure 5.23). The particles can be rendered individually, or they can be rendered as spheres of gas with a given density. The advantage of this technique is that it is similar to rigid body dynamics and therefore the equations are relatively simple and familiar. The equations can be simplified if the rotational dynamics are ignored. In addition, there are no restrictions imposed by the simulation setup as to where the gas may travel. The disadvantage of this approach is that a large number of particles are

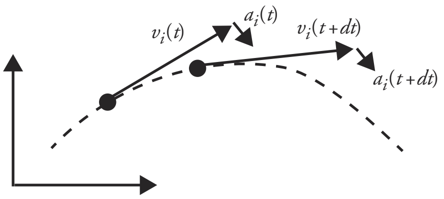


**Figure 5.23** Particle-based method

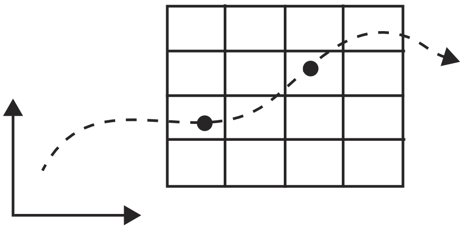


**Figure 5.24** Hybrid method

needed to simulate a dense, expansive gas. Particles are assigned masses, and external forces can be easily incorporated by updating particle accelerations and, subsequently, velocities.

### Hybrid Method

Some models of gas trace particles through a spatial grid. Particles are passed from cell to cell as they traverse the interesting space (Figure 5.24). The display attributes of individual cells are determined by the number and type of particles contained in the cell at the time of display. The particles are used to carry and distribute attributes through the grid, and then the grid is used to produce the display.

## 5.3.2 Computational Fluid Dynamics

The more physically based methods derive their equations from the Navier-Stokes (NS) equations, which are the basis of computational fluid dynamics (CFD) calculations used in scientific visualization. The standard NS approach is grid based and sets up differential equations based on the conservation of momentum, mass, and energy as it considers flow into and out of differential elements (Figure 5.25). There are also vortex-based methods, which tend to be particle based. Such approaches quickly get mathematically complex and are beyond the scope of this book.

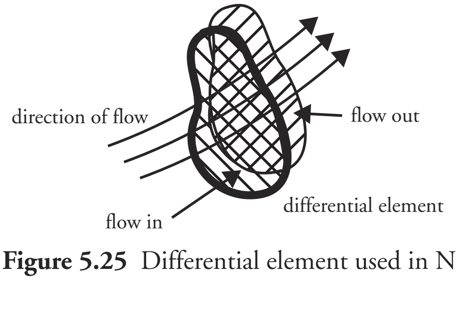


**Figure 5.25** Differential element used in Navier-Stokes

### 5.3.3 Clouds

Modeling clouds is a very difficult task because of their complex, amorphous, space-filling structure and because even an untrained eye can easily judge the realism of a cloud model. The ubiquitous nature of clouds makes them an important modeling and animation task. This section describes their important visual and physical characteristics, important rendering issues, and several approaches for cloud modeling and animation.

#### Basic Cloud Types and Physics

Clouds are made of visible ice crystals and/or water droplets suspended in air, depending on altitude and, hence, air temperature. Clouds are formed when air rises, its water vapor cooling to the saturation point and condensing. The visible condensed water vapor is what constitutes the cloud [8]. The shape of the cloud varies based on processes that force the air to rise or bubble up (convection, convergence, lifting along frontal boundaries, lifting due to mountains, or orography, Kelvin-Helmholtz shearing, etc.) and the height (and other conditions) at which the cloud forms [8]. Several sources [7] [8] [21] [46] present a very good introduction to clouds and their identification. Clouds formed above 20,000 feet (cirrus) are wispy and white in appearance and composed primarily of ice crystals. Clouds formed between 6,500 feet and 23,000 feet (i.e., altocumulus) are primarily composed of water droplets; they are small and puffy and they collect into groups, sometimes forming waves. Clouds formed below 6,500 feet (e.g., stratus, stratocumulus) are again composed primarily of water droplets; they extend over a large area and have a layered or belled appearance. The most characteristic cloud type is the puffy cumulus. Cumulus clouds are normally formed by convection or frontal lifting and can vary from having little vertical height to forming huge vertical towers (cumulonimbus) created by strong convection.

#### Visual Characteristics of and Rendering Issues for Clouds

Clouds have several easily identifiable visual characteristics that must be modeled to produce accurate images and animations. First, clouds have a volumetrically varying amorphous structure with detail at many different scales. Second, cloud formation often results from swirling, bubbling, and turbulent processes that produce characteristic time-varying patterns. Third, clouds have several illumination and shading characteristics that must be accurately rendered to obtain convincing images. Clouds are a three-dimensional medium of small ice and water droplets that absorb, scatter, and reflect light. Illumination models for clouds are classified as low-albedo and high-albedo models. A low-albedo reflectance model assumes that secondary scattering effects are negligible, whereas a high-albedo illumination model calculates the secondary and higher-order scattering effects. For optically thick clouds, such as cumulus, stratus, and cumulonimbus, secondary scattering effects are significant and high-albedo illumination models (e.g., [2] [22] [25] [30] [40]) should be used. Detailed descriptions of implementing a low-albedo illumination algorithm can be found in several sources [13] [22]. Simulation of wavelength-dependent scattering is also important for creating correct atmospheric dispersion effects for sunrise and sunset scenes (see Figure 5.26 [Plate 4] for a rendering of sunset illumination). Self-shadowing of clouds and cloud shadowing on landscapes are also important for creating realistic images of cloud scenes and landscapes. Correct cloud shadowing requires volumetric shadowing techniques to create accurate images, which can be very expensive when volumetric ray tracing is used. A much faster alternative is to use volumetric shadow tables [12] [13] [22].

### Early Approaches to Cloud Modeling

Modeling clouds in computer graphics has been a challenge for more than twenty years [10]. Many early approaches used semitransparent surfaces to produce convincing images of clouds [18] [19] [47]. Voss [47] has used fractal synthesis of parallel plane models to produce images of clouds seen from a distance. Gardner [18] [19] has produced convincing images and animations of clouds by using Fourier synthesis to control the transparency of large, hollow ellipsoids. This approach

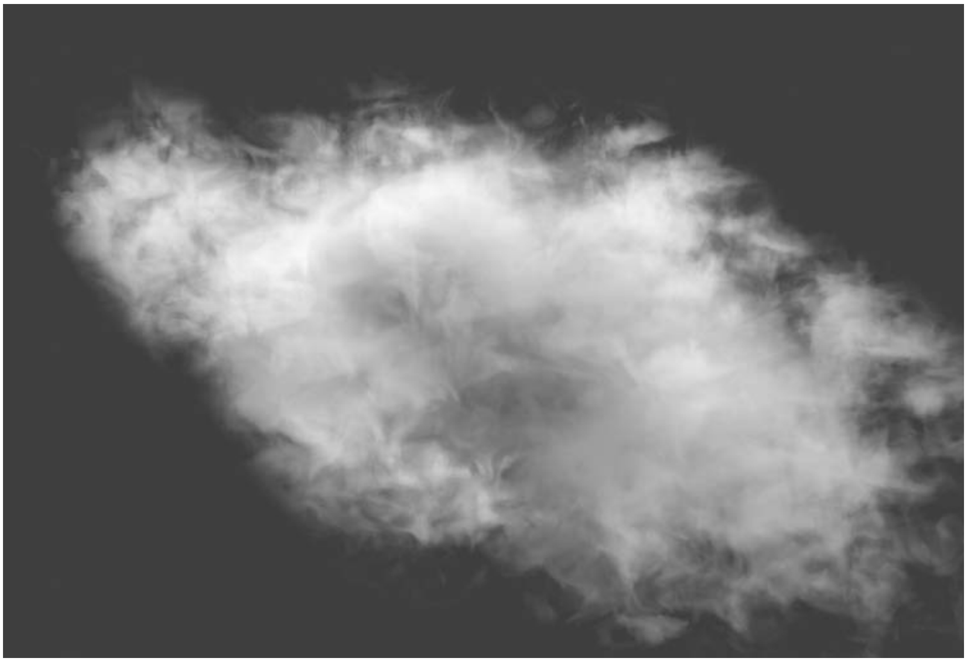


**Figure 5.26** An example of cirrus and cirrostratus clouds at sunset
(Copyright 1998 David S. Ebert)

uses groups of ellipsoids to define and animate the general shape of clouds while using procedurally textured transparency to produce the appearance of cloud detail.

A similar approach has been taken by Kluyskens [24] to produce clouds in Alias/Wavefront's Maya$^{TM}$ animation systems. He uses randomized, overlapping spheres to define the general cloud shape. A solid *cloud texture* is then used to color the cloud and to control the transparency of the spheres. Finally, Kluyskens increases the transparency of the spheres near their edges so that the defining shape is not noticeable.

### Volumetric Cloud Modeling

Although surface-based techniques can produce realistic images of clouds viewed from a distance, these cloud models are hollow and do not allow the user to seamlessly enter, travel through, and inspect their interior. Volumetric density-based models must be used to capture the three-dimensional structure of a cloud. Kajiya [22] produced the first volumetric cloud model in computer graphics. Stam and Fiume [43] and Foster and Metaxas [14] have produced convincing volumetric models of smoke and steam but have not done substantial work on modeling clouds.

Neyret [28] has produced some preliminary results of a convective cloud model based on general physical characteristics, such as bubbling and convection processes. This model seems promising for simulating convective clouds; however, it currently uses surfaces (large particles) to model the cloud structure. Extending this approach to volumetric modeling/animation should produce convincing cloud images and animations.

Particle systems [37] are commonly used to simulate the volumetric gases, such as smoke, with convincing results and provide easy animation control. The difficulty with using particle systems for cloud modeling is the massive number of particles that are necessary to simulate realistic clouds.

Several authors have used the idea of volume-rendered implicit functions (e.g., [5]) for volumetric cloud modeling. Nishita, Nakamae, and Dobashi [29] have concentrated on illumination effects and have used volume-rendered implicits as a basic cloud model in their work on multiple scattering illumination models. Stam [43] [44] [45] has also used volumetric blobbies to create his models of smoke and clouds. Ebert [11] [12] has used volumetric implicits combined with particle systems and procedural detail to simulate the formation and geometry of volumetric clouds. This approach uses implicits to provide a natural way of specifying and animating the global structure of the cloud while using more traditional procedural techniques to model the detailed structure. The implicits are controlled by a modified particle system that incorporates simple simulations of cloud formation dynamics. Example images created by this technique can be seen in Figure 5.26 and Figure 5.27 (Plate 5).

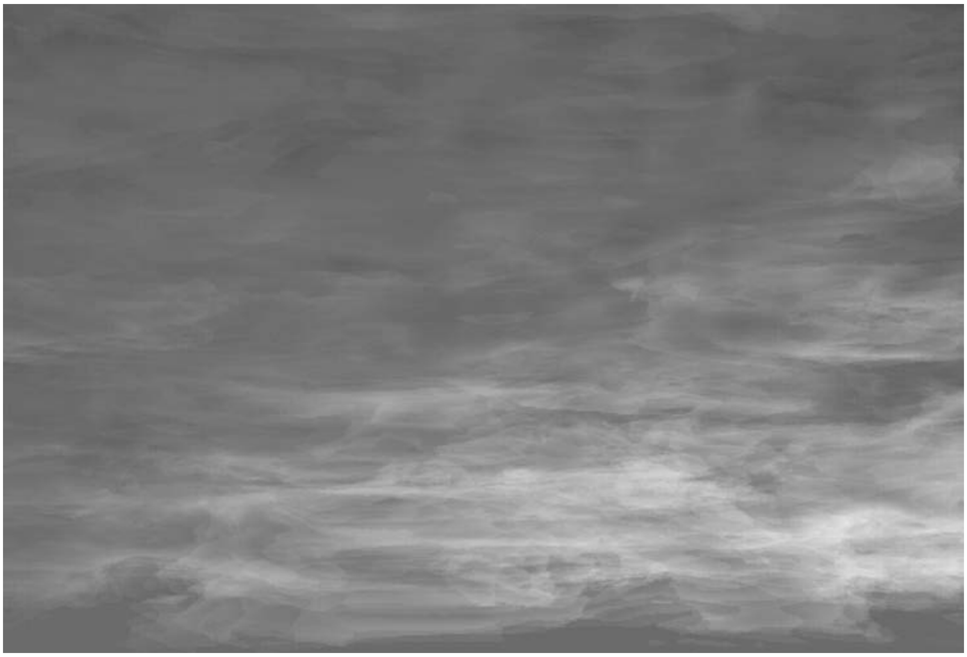


**Figure 5.27** An example of a cumulus cloud (Copyright 1997 David S. Ebert)

### Example Volumetric Cloud Modeling System

Ebert's cloud modeling and animation approach uses procedural abstraction of detail to allow the designer to control and animate objects at a high level. Its inherent procedural nature provides flexibility, data amplification, abstraction of detail, and ease of parametric control. Abstraction of detail and data amplification are necessary to make the modeling and animation of complex volumetric phenomena, such as clouds, tractable. It would be impractical for an animator to specify and control the detailed three-dimensional density of a cloud model. This system does not use a physics-based approach because it is computationally prohibitive and nonintuitive for many animators and modelers. Setting and animating correct physics parameters for condensation temperature, temperature and pressure gradients, and so on is a time-consuming, detailed task. This model was developed to allow the modeler and the animator to work at a much higher level and does not restrict the animator by the laws of physics. Since a procedure is evaluated to determine the object's density, any advanced modeling technique, simple physics simulation, mathematical function, or artistic algorithm can be included in the model.

As mentioned earlier, this volumetric cloud model uses a two-level hierarchy: the cloud macrostructure and the cloud microstructure. These are modeled by implicit functions and turbulent volume densities, respectively. The basic structure of the cloud model combines these two components to determine the final density of the cloud.

The cloud's microstructure is created by using procedural *turbulence* and *noise* functions (see Appendix B). This allows the procedural simulation of natural detail to the level needed. Simple mathematical functions are added to allow shaping of the density distributions and control over the sharpness of the density falloff.

Implicit functions work well to model the cloud macrostructure because of their ease of specification and their smoothly blending density distributions. The user simply specifies the location, type, and weight of the implicit primitives to create the overall cloud shape. Any implicit primitive, including spheres, cylinders, ellipsoids, and skeletal implicits, can be used to model the cloud macrostructure. Since these are volume rendered as a semitransparent medium, the whole volumetric field function is being rendered, as compared to implicit surface rendering, where only a small range of values of the field are used to create the objects.

The implicit density functions are primitive based: they are defined by summed, weighted, parameterized, primitive implicit surfaces. A simple example of the implicit formulation of a sphere centered at the point *center* with radius $r$ is $F(x, y, z) = (x - \text{center.x})^2 + (y - \text{center.y})^2 + (z - \text{center.z})^2 - r^2 = 0$.

The real power of implicit functions is the smooth blending of the density fields from separate primitive sources. A standard cubic function [49] is often used as the density (blending) function for the implicit primitives (Equation 5.28). In Equation 5.28, $r$ is the distance from the primitive. This density function is cubic in the distance squared, and its value ranges from 1, when $r = 0$ (within the primitive), to 0, at $r = R$. This density function has several advantages. First, its value drops off quickly to zero (at the distance $R$), reducing the number of primitives that must be considered in creating the final surface. Second, it has zero derivatives at $r = 0$ and $r = R$ and is symmetrical about the contour value 0.5, providing for smooth blends between primitives. The final implicit density value is then the weighted sum of the density field values of each primitive (Equation 5.29). Variable $w_i$ is the weight of the $i$th primitive, and $q$ is the closest point on element $i$ from $p$.

$$F(r) = -\frac{4}{9}\cdot\frac{r^6}{R^6} + \frac{17}{9}\cdot\frac{r^4}{R^4} - \frac{22}{9}\cdot\frac{r^2}{R^2} + 1 \tag{Eq. 5.28}$$

$$D(p) = \sum_i w_i \cdot F(|p - q|) \tag{Eq. 5.29}$$

To create nonsolid implicit primitives, the animator procedurally alters the location of the point before evaluating the blending functions. This alteration can be the product of the procedure and the implicit function and/or a warping of the implicit space. These techniques are combined into a simple cloud model as shown in the high-level description below.

```
volumetric_procedural_implicit_function(pnt, blend, pixel_size)
    perturbed_point = procedurally alter pnt using noise and turbulence
    density1 = implicit_function(perturbed_point)
    density2 = turbulence(pnt, pixel_size)
    blend = blend * density1 +  (1 - blend) * density2
    density = shape the resulting blend based on user controls for
       wispiness and denseness (e.g., use pow and exponential function)
   return(density)
```

The density from the implicit primitives is combined with a pure turbulence-based density using a user-specified blend (60% to 80% gives good results). The blending of the two densities allows the creation of clouds that range from those entirely determined by the implicit function density to those entirely determined by the procedural turbulence function. When the clouds are completely determined by the implicit functions, they tend to look more like cotton balls. The addition of the procedural alteration and turbulence is what gives them their naturalistic look.

### Cumulus Clouds

Cumulus clouds are very common and can be easily simulated using spherical or elliptical implicit primitives. Figure 5.27 shows the type of result that can be achieved by using nine implicit spheres to model a cumulus cloud. The animator or modeler simply positions the implicit spheres to produce the general cloud structure. Procedural modification then alters the density distribution to create the detailed wisps. The algorithm used to create the clouds in Figure 5.27 follows.

```
cumulus(pnt,density,parms, pnt_w, vol)
        xyz_td   pnt;           /* location of point in cloud space */
        xyz_td   pnt_w;         /* location of point in world space */
        float    *density,*parms;
        vol_td   vol;
{
    float turbulence();         /* turbulence function */
    float  noise();             /* noise function */
    float  metaball_evaluate(); /* function for evaluating the metaball
                                   primitives*/
        float mdens,            /* metaball density value */
        turb,                   /* turbulence amount */
        noise_value;            /* noise value */
    xyz_td path;                /* path for swirling the point */
    extern int frame_num;
    static int ncalcd=1;
    static float sin_theta_cloud, cos_theta_cloud, theta,
            path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;

    /* calculate values that only depend on the frame number once per
       frame */
    if(ncalcd)     {
        ncalcd=0;
        /* create gentle swirling in the cloud */
        theta =(frame_num%600)*01047196;          /* swirling effect */
        cos_theta_cloud = cos(theta);
        sin_theta_cloud = sin(theta);
        path_x = sin_theta_cloud*.005*frame_num;
        path_y = .01215*(float)frame_num;
        path_z = sin_theta_cloud*.0035*frame_num;
        scalar_x = (.5+(float)frame_num*0.010);
        scalar_z = (float)frame_num*.0073;
    }
    /* Add some noise to the point's location  */
    noise_value = noise(pnt);                 /* Use noise function */
    pnt.x -= path_x -noise_value*scalar_x;
    pnt.y = pnt.y - path_y +.5*noise_value;
    pnt.z += path_z - noise_value*scalar_z;

    /* Perturb the location of the point before evaluating the implicit
       primitives.  */
    turb=turbulence(pnt);
    turb_amount=parms[4]*turb;
    pnt_w.x += turb_amount;
    pnt_w.y -= turb_amount;
    pnt_w.z += turb_amount;

    mdens=(float)metaball_evaluate((double)pnt_w.x, (double)pnt_w.y,
                               (double)pnt_w.z, (vol.metaball));

    *density= parms[1]*(parms[3]*mdens + (1.0 - parms[3])*turb*mdens);
    *density= pow(*density,(double)parms[2]);

}
```

*Parms*[3] is the blending function value between implicit (metaball) density and the product of the turbulence density and the implicit density. This method of blending ensures that the entire cloud density is a product of the implicit field values, preventing cloud pieces from occurring outside the defining primitives. Using a large *parms*[3] generates clouds that are mainly defined by their implicit primitives and are, therefore, "smoother" and less turbulent. *Parms*[1] is a density scaling factor; *parms*[2] is the exponent for the *pow*() function; and *parms*[4] controls the amount of turbulence used in displacing the point before evaluation of the implicit primitives. For good images of cumulus clouds, useful values are the following: 0.2 < *parms*[1] < 0.4, *parms*[2] = 0.5, *parms*[3] = 0.4, and *parms*[4] = 0.7.

### Cirrus and Stratus Clouds

Cirrus clouds differ greatly from cumulus clouds in their density, thickness, and falloff. In general, cirrus clouds are thinner, less dense, and wispier. These effects can be created by altering the parameters of the cumulus cloud procedure and also by changing the implicit primitives. The density value parameter for a cirrus cloud is normally chosen as a smaller value and the chosen exponent is larger, producing larger areas of no clouds and a greater number of individual clouds. To create cirrus clouds, the user can simply specify the global shape (envelope) of the clouds with a few implicit primitives, or he or she can specify implicit primitives to determine the location and shape of each cloud. In the former case, the shape of each cloud is controlled mainly by the volumetric procedural function and turbulence simulation, unlike with cumulus clouds, for which the implicit functions are the main shape control. It is also useful to modulate the densities along the direction of the jet stream to produce more natural wisps. This can be created by the user specifying a predominant direction of wind flow and using a turbulent version of this vector in controlling the densities as follows:

```
Cirrus(pnt,density,parms, pnt_w, vol, jet_stream)
        xyz_td pnt;             /* location of point in cloud space */
        xyz_td pnt_w;           /* location of point in world space */
        xyz_td jet_stream;
        float  *density,*parms;
        vol_td vol;
{
    float turbulence();         /* turbulence function */
    float noise();              /* noise function */
    float metaball_evaluate(); /* function for evaluating the metaball
                                   primitives*/
    float   mdens,              /* metaball density value */
            turb,               /* turbulence amount */
            noise_value;        /* noise value */
    xyz_td path;                /* path for swirling the point */
    extern int frame_num;
    static int ncalcd=1;
    static float sin_theta_cloud, cos_theta_cloud, theta,
            path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;

    /* calculate values that only depend on the frame number once per
       frame  */
    if(ncalcd) {
        ncalcd=0;
        /* create gentle swirling in the cloud */
        theta =(frame_num%600)*01047196;          /* swirling effect */
        cos_theta_cloud = cos(theta);
        sin_theta_cloud = sin(theta);
        path_x = sin_theta_cloud*.005*frame_num;
        path_y = .01215*(float)frame_num;
        path_z = sin_theta_cloud*.0035*frame_num;
        scalar_x = (.5+(float)frame_num*0.010);
        scalar_z = (float)frame_num*.0073;
    }

    /* Add some noise to the point's location  */
    noise_value = noise(pnt);
    pnt.x -= path_x -noise_value*scalar_x;
    pnt.y = pnt.y - path_y +.5*noise_value;
    pnt.z += path_z - noise_value*scalar_z;

    /* Perturb the location of the point before evaluating the implicit
       primitives. */
    turb=turbulence(pnt);
    turb_amount=parms[4]*turb;
    pnt_w.x += turb_amount;
    pnt_w.y -= turb_amount;
    pnt_w.z += turb_amount;

 /* make the jet stream turbulent */
    jet_stream.x + =.2*turb;
    jet_stream.y + =.3*turb;
    jet_stream.z + =.25*turb;

    /* warp point along the jet stream vector */
    pnt_w = warp(jet_stream, pnt_w);

    mdens=(float)metaball_evaluate((double)pnt_w.x, (double)pnt_w.y,
                               (double)pnt_w.z, (vol.metaball));

    *density= parms[1]*(parms[3]*mdens + (1.0 - parms[3])*turb*mdens);
    *density= pow(*density,(double)parms[2]);
}
```

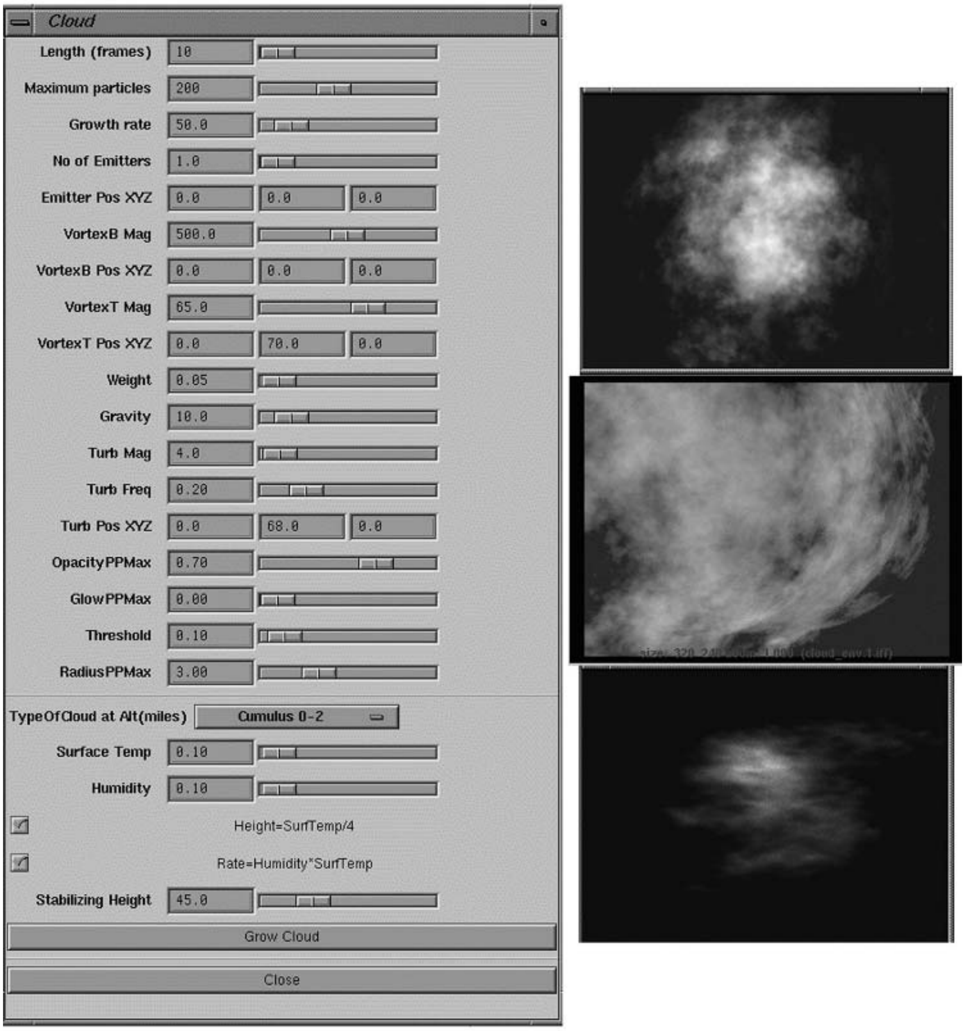
An example of a cirrus cloud formation created using these techniques is given in Figure 5.26.

Stratus clouds can also be modeled by using a few implicits to create the global shape or extent of the stratus layer while using volumetric procedural functions to define the detailed structure of all the clouds within this layer. Stratus cloud layers are normally thicker and less wispy than cirrus clouds. This effect can be created by adjusting the size of the turbulent space (smaller/fewer wisps), using a smaller exponent value (creates more of a cloud layer effect), and increasing the density of the cloud. Some of the more interesting stratus effects, such as a mackerel sky, can be created by using simple mathematical functions to shape the densities.

### Animating Volumetric Procedural Clouds

The volumetric cloud models described above produce nice still images of clouds and also clouds that gently evolve over time. The models can be animated by particle system dynamics with the implicit primitives attached to each particle. Since the implicits are modeling the macrostructure of the cloud, while procedural techniques are modeling the microstructure, fewer primitives are needed to achieve complex cloud models. The smooth blending and procedurally generated detail allow complex results with less than a few hundred primitives, a factor of 100 to 1,000 less than needed with traditional particle systems. The user specifies a few initial implicit primitives and dynamics information, such as speed, initial velocity, force function, and lifetime, and the system generates the location, number, size, and type of implicit for each frame. Unlike traditional particle systems, cloud implicit particles never die; they just become dormant.

Any commercial particle animation program, such as Maya$^{\text{TM}}$, can also be used to control the dynamics of the cloud particle system. A useful approach for cloud dynamics is to use *qualitative dynamics:* simple simulations of the observed properties and formation of clouds. The underlying physical forces that create a wide range of cloud formations are extremely complex to simulate, computationally expensive, and very restrictive. The incorporation of simple, parameterized rules that simulate observable cloud behavior will produce a powerful cloud animation system. Figure 5.28(a) (Plate 6) shows a graphical user interface (GUI) used to generate and animate a particle system for simulating convective (cumulus) cloud formations based on qualitative dynamics. This Maya GUI controls a MEL$^{\text{TM}}$ script that generates the particle system and controls its dynamics. It uses Maya's vortex, airfield, and turbulence fields in its simulation of convection and cloud particle bubbling. Example images from this script can be seen in Figure 5.28(b). The simulation works as follows. Cloud bubbles are generated on the ground by the user specifying either the area and the humidity level or the placement of a particle emitter and its spread. The bubbles rise due to the force generated by temperature difference, and their weight and the force of gravity affect them. A vortex



| | | | |
|---|---|---|---|
| Length (frames) | 10 | | |
| Maximum particles | 200 | | |
| Growth rate | 50.0 | | |
| No of Emitters | 1.0 | | |
| Emitter Pos XYZ | 0.0 | 0.0 | 0.0 |
| VortexB Mag | 500.0 | | |
| VortexB Pos XYZ | 0.0 | 0.0 | 0.0 |
| VortexT Mag | 65.0 | | |
| VortexT Pos XYZ | 0.0 | 70.0 | 0.0 |
| Weight | 0.05 | | |
| Gravity | 10.0 | | |
| Turb Mag | 4.0 | | |
| Turb Freq | 0.20 | | |
| Turb Pos XYZ | 0.0 | 68.0 | 0.0 |
| OpacityPPMax | 0.70 | | |
| GlowPPMax | 0.00 | | |
| Threshold | 0.10 | | |
| RadiusPPMax | 3.00 | | |
| TypeOfCloud at Alt(miles) | Cumulus 0-2 | | |
| Surface Temp | 0.10 | | |
| Humidity | 0.10 | | |

☑ Height=SurfTemp/4

☑ Rate=Humidity*SurfTemp

| | |
|---|---|
| Stabilizing Height | 45.0 |

(a) GUI used to control cloud formation

(b) Example clouds

**Figure 5.28** An example of cloud dynamics GUI and example images created in Maya$^{\mathrm{TM}}$ (Copyright 1999 Ruchigartha)

field is used to simulate the movement of the bubbles in air. At an altitude determined by the surface temperature, the number of dust nuclei at that height, and the humidity content, condensation takes place, so the hot steam cools off and can now be seen as cloud particles. Alternatively, the user can explicitly specify the height at which the stabilization and the aggregation of the bubbles occur to form the cloud. The bubbles are simulated by particles, which have several attributes, such as position, radius, opacity, velocity, and lifetime. When a particle's radius becomes too large, the particle creates child particles and has its radius decreased to conserve matter.

### Summary

Despite the complexity of the physical processes that form clouds, most of their important visual aspects have been effectively modeled by researchers. However, there are still challenges in terms of providing user control of cloud motion and in improving the fine-grain motion and rendering.

## 5.3.4 Fire

Fire is a particularly difficult and computationally intensive process to model. It has all the complexities of smoke and clouds and the added complexity of very active internal processes that produce light and motion and create rapidly varying display attributes.

Recently, impressive advances have been made in the modeling of fire. At one extreme, the most realistic approaches require sophisticated techniques from computational fluid dynamics and are difficult to control (e.g., [43]). Work has also been performed in simulating the development and spread of fire for purposes of tracking its movement in an environment (e.g., [6] [39]). These models tend to be only global, extrinsic representations of the fire's movement and less concerned with the intrinsic motion of the fire itself. Falling somewhere between these two extremes, particle systems provide effective, yet computationally attractive, approaches to fire.

### Particle System Approach

One of the first and most popularly viewed examples of computer-generated fire appears in the movie *Star Trek II: The Wrath of Khan* [31]. In the sequence referred to as the *genesis effect,* an expanding wall of fire spreads out over the surface of the planet from a single point of impact. The simulation is not a completely convincing model of fire, although the sequence is effective in the movie. The model uses a two-level hierarchy of particles. The first level of particles is located at the point of impact to simulate the initial blast; the second level consists of concentric rings of particles, timed to progress from the central point outward, forming the wall of fire and explosions.

Each ring of second-level hierarchy consists of a number of individual particle systems that are positioned on the ring and overlap so as to form a continuous ring. The individual particle systems are modeled to look like explosions (Figure 5.29). The particles in each one of these particle systems are oriented to fly up and

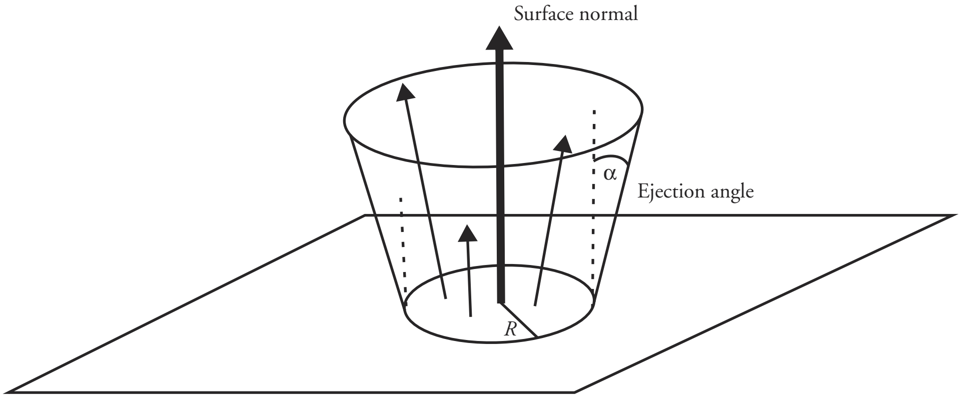


**Figure 5.29** Explosion-like particle system

away from the surface of the planet. The initial position for a particle is randomly chosen from the circular base of the particle system. The initial direction of travel for each particle is constrained to deviate less than the ejection angle away from the surface normal.

### Other Approaches

Various other approaches have been used in animations with varying levels of success. Two-dimensional animated texture maps have been used to create the effect of the upward movement of burning gas, but such models are effective only when viewed from a specific direction. Using a two-dimensional-multiple planes approach adds some depth to the fire, but viewing directions are still limited. Stam and Fiume [43] present advection-diffusion equations to evolve both density and temperature fields. The user controls the simulation by specifying a wind field. The results are effective, but the foundation mathematics are complicated and the model is difficult to control. Other work (e.g., [6] [39]) models the spread of fire in pools or through buildings but does not concentrate on visual realism.

## 5.3.5 Summary

Modeling and animating gaseous phenomena is difficult. Gases are constantly changing shape and lack even a definable surface. Volume graphics holds the most promise for modeling and animating gas, but currently it has computational drawbacks that make such approaches of limited use for animation. A useful and visually accurate model of fire remains the subject of research.

## 5.4 Chapter Summary

Modeling and animating many of the phenomena that occur in nature is challenging. Most of the techniques discussed in this chapter are still the subject of research efforts. Plants exhibit both enormous complexity and a well-defined structure. Much progress has been made in capturing both of these aspects in plant models. However, work on plants interacting with the environment is fairly recent. Water, smoke, clouds, and fire share an amorphous nature, which makes them difficult to model and animate. Approaches that incorporate a greater amount of physics have been developed recently for these phenomena. As processing power becomes cheaper, techniques such as computational fluid dynamics become more practical (and more desirable) tools for animating water and gas, but convenient controls for such models have yet to be developed.

## References

1. M. Aono and T. L. Kunii, "Botanical Tree Image Generation," *IEEE Computer Graphics and Applications,* 4 (5), pp. 10–34 (May 1984).
2. J. Blinn, "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces," *Computer Graphics* (Proceedings of SIGGRAPH 82), 16 (3), pp. 21–29 (July 1982, Boston, Mass.).
3. J. Blinn, "Simulation of Wrinkled Surfaces," *Computer Graphics* (Proceedings of SIGGRAPH 78), 12 (3), pp. 286–292 (August 1978, Atlanta, Ga.).
4. J. Bloomenthal, "Modeling the Mighty Maple," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 305–311 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
5. J. Bloomenthal, C. Bajaj, J. Blinn, M.-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill, *Introduction to Implicit Surfaces,* Morgan Kaufmann, San Francisco, 1997.
6. R. Bukowski and C. Sequin, "Interactive Simulation of Fire in Virtual Building Environments," Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 35–44 (August 1997, Los Angeles, Calif.). Addison-Wesley. Edited by Turner Whitted. ISBN 0-89791-896-7.
7. W. Cotton and A. Anthes, *Storm and Cloud Dynamics,* Academic Press, New York, 1989.
8. Department of Atmospheric Sciences, University of Illinois, Cloud Catalog, http://covis.atmos.uiuc.edu/guide/clouds/.
9. O. Deussen, P. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz, "Realistic Modeling and Rendering of Plant Ecosystems," Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 275–286 (July 1998, Orlando, Fla.). Addison-Wesley. Edited by Michael Cohen. ISBN 0-89791-999-8.
10. W. Dungan, Jr., "A Terrain and Cloud Computer Image Generation Model," *Computer Graphics* (Proceedings of SIGGRAPH 79), 13 (2), pp. 143–150 (August 1979, Chicago, Ill.).

11. D. Ebert, "Volumetric Modeling with Implicit Functions: A Cloud Is Born," Visual Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, p. 147 (August 1997, Los Angeles, Calif.). Addison-Wesley.
12. D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach,* AP Professional, Cambridge, Mass., 1998.
13. D. Ebert and R. Parent, "Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-Buffer Techniques," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 357–366 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
14. N. Foster and D. Metaxas, "Modeling the Motion of a Hot, Turbulent Gas," Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 181–188 (August 1997, Los Angeles, Calif.). Addison-Wesley. Edited by Turner Whitted. ISBN 0-89791-896-7.
15. N. Foster and D. Metaxas, "Realistic Animation of Liquids," *Graphical Models and Image Processing,* 58 (5), pp. 471–483 (September 1996). Academic Press.
16. A. Fournier and W. Reeves, "A Simple Model of Ocean Waves," *Computer Graphics* (Proceedings of SIGGRAPH 86), 20 (4), pp. 75–84 (August 1986, Dallas, Tex.). Edited by David C. Evans and Russell J. Athay.
17. P. Fournier, A. Habibi, and P. Poulin, "Simulating the Flow of Liquid Droplets," *Graphics Interface '98,* pp. 133–142 (June 1998). Edited by Kellogg Booth and Alain Fournier. ISBN 0-9695338-6-1.
18. G. Gardner, "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 11–20 (July 1984, Minneapolis, Minn.).
19. G. Gardner, "Visual Simulation of Clouds," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 297–303 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
20. N. Greene, "Voxel Space Automata: Modeling with Stochastic Growth Processes in Voxel Space," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 175–184 (July 1989, Boston, Mass.). Edited by Jeffrey Lane.
21. R. House, *Cloud Dynamics,* Academic Press, Orlando, Fla., 1993.
22. J. Kajiya and B. Von Herzen, "Ray Tracing Volume Densities," *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 165–174 (July 1984, Minneapolis, Minn.).
23. M. Kass and G. Miller, "Rapid, Stable Fluid Dynamics for Computer Graphics," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 49–57 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
24. T. Kluyskens, "Making Good Clouds," MAYA based QueenMaya magazine tutorial, http://reality.sgi.com/tkluyskens_aw/txt/tutor6.html.
25. N. Max, "Efficient Light Propagation for Multiple Anisotropic Volume Scattering," Fifth Eurographics Workshop on Rendering, pp. 87–104 (June 1994, Darmstadt, Germany).
26. N. Max, "Vectorized Procedural Models for Natural Terrains: Waves and Islands in the Sunset," *Computer Graphics* (Proceedings of SIGGRAPH 81), 15 (3), pp. 317–324 (August 1981, Dallas, Tex.).

27. R. Mech and P. Prusinkiewicz, “Visual Models of Plants Interacting with Their Environment,” Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, pp. 397–410 (August 1996, New Orleans, La.). Addison-Wesley. Edited by Holly Rushmeier. ISBN 0-201-94800-1.
28. F. Neyret, Qualitative Simulation of Convective Clouds Formation and Evolution, Computer Animation and Simulation ’97, pp. 113–124 (September 1997, Budapest, Hungary). *Eurographics.* Edited by D. Thalmann and M. van de Panne. ISBN 3-211-83048-0.
29. T. Nishita, E. Nakamae, and Y. Dobashi, “Display of Clouds and Snow Taking into Account Multiple Anisotropic Scattering and Sky Light,” Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, pp. 379–386 (August 1996, New Orleans, La.). Addison-Wesley. Edited by Holly Rushmeier. ISBN 0-201-94800-1.
30. P. Oppenheimer, “Real-Time Design and Animation of Fractal Plants and Trees,” *Computer Graphics* (Proceedings of SIGGRAPH 86), 20 (4), pp. 55–64 (August 1986, Dallas, Tex.). Edited by David C. Evans and Russell J. Athay.
31. Paramount, *Star Trek II: The Wrath of Khan* (film), June 1982.
32. D. Peachey, “Modeling Waves and Surf,” *Computer Graphics* (Proceedings of SIGGRAPH 86), 20 (4), pp. 65–74 (August 1986, Dallas, Tex.). Edited by David C. Evans and Russell J. Athay.
33. P. Prusinkiewicz, M. Hammel, and E. Mjolsness, “Animation of Plant Development,” Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series, pp. 351–360 (August 1993, Anaheim, Calif.). Edited by James T. Kajiya. ISBN 0-201-58889-7.
34. P. Prusinkiewicz, M. James, and M. R. Mech, “Synthetic Topiary,” Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, pp. 351–358 (July 1994, Orlando, Fla.). ACM Press. Edited by Andrew Glassner. ISBN 0-89791-667-0.
35. P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants,* Springer-Verlag, New York, 1990.
36. P. Prusinkiewicz, A. Lindenmayer, and J. Hanan, “Developmental Models of Herbaceous Plants for Computer Imagery Purposes,” *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 141–150 (August 1988, Atlanta, Ga.). Edited by John Dill.
37. W. T. Reeves, “Particle Systems: A Technique for Modeling a Class of Fuzzy Objects*,” ACM Transactions on Graphics,* 2 (2), pp. 91–108 (April 1983).
38. P. Reffye, C. Edelin, J. Francon, M. Jaeger, and C. Puech, “Plant Models Faithful to Botanical Structure and Development,” *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 151–158 (August 1988, Atlanta, Ga.). Edited by John Dill.
39. H. Rushmeier, A. Hamins, and M. Choi, “Volume Rendering of Pool Fire Data,” *IEEE Computer Graphics and Applications,* 15 (4), pp. 62–67 (July 1995).
40. H. Rushmeier and K. Torrance, “The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium,” *Computer Graphics* (Proceedings of SIGGRAPH 87), 21 (4), pp. 293–302 (July 1987, Anaheim, Calif.). Edited by Maureen C. Stone.
41. A. R. Smith, “Plants, Fractals, and Formal Languages,” *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 1–10 (July 1984, Minneapolis, Minn.).
42. J. Stam, “Stable Fluids,” Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, pp. 121–128 (August 1999, Los Angeles, Calif.). Addison-Wesley Longman. Edited by Alyn Rockwood. ISBN 0-20148-560-5.

43. J. Stam and E. Fiume, “Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes,” Proceedings of SIGGRAPH 95 (Los Angeles, Calif., August 6–11). In *Computer Graphics Proceedings,* Annual Conference Series, 1995, ACM SIGGRAPH, pp. 129–136.
44. J. Stam and E. Fiume, “A Multiple-Scale Stochastic Modelling Primitive,” *Graphics Interface ’91,* pp. 24–31 (June 1991). Canadian Information Processing Society.
45. J. Stam and E. Fiume, “Turbulent Wind Fields for Gaseous Phenomena,” Proceedings of SIGGRAPH 93, *Computer Graphics Proceedings,* Annual Conference Series, pp. 369–376 (August 1993, Anaheim, Calif.). Edited by James T. Kajiya. ISBN 0-201-58889-7.
46. R. Tricker, *The Science of the Clouds,* American Elsevier, New York, 1970.
47. R. Voss, “Fourier Synthesis of Gaussian Fractals: 1/f Noises, Landscapes, and Flakes,” Tutorial, SIGGRAPH 83 (July 1983, Detroit, Mich.).
48. H. Weimer and J. Warren, “Subdivision Schemes for Fluid Flow,” Proceedings of SIGGRAPH 99, *Computer Graphics Proceedings,* Annual Conference Series, pp. 111–120 (August 1999, Los Angeles, Calif.). Addison-Wesley Longman. Edited by Alyn Rockwood. ISBN 0-20148-560-5.
49. Brian Wyvill, Craig McPheeters, and Geoff Wyvill, “Data Structure for Soft Objects,” *Visual Computer,* 2 (4), pp. 227–234 (1986).

CHAPTER 6

# Modeling and Animating Articulated Figures

Modeling and animating an articulated figure is a daunting task. It is especially challenging when the figure is meant to represent a human. There are several major reasons for this. First, the human figure is a very familiar form. This familiarity makes each person a critical observer. When confronted with an animated figure, a person readily recognizes when its movement does not “feel” or “look” right. Second, the human form is very complex, with more than two hundred bones and six hundred muscles. When fully modeled with linked rigid segments, the human form is endowed with approximately two hundred degrees of freedom. The deformable nature of the body’s parts further complicates the modeling and animating task. Third, humanlike motion is not computationally well defined. Some studies have tried to accurately describe humanlike motion, but typically these descriptions apply only to certain constrained situations. Fourth, there is no one definitive motion that is humanlike. Differences resulting from genetics, culture, personality, and emotional state all can affect how a particular motion is carried out. General strategies for motion production have not been described, nor have the nuances of motion that make each of us unique and uniquely recognizable. Although the discussion in this chapter focuses primarily on the human form, many of the techniques apply to any type of articulated figure.

**Table 6.1** Selected Terms from Anatomy

| | |
|---|---|
| **Sagittal plane** | Perpendicular to the ground and divides the body into right and left halves |
| **Coronal plane** | Perpendicular to the ground and divides the body into front and back halves |
| **Transverse plane** | Parallel to the ground and divides the body into top and bottom halves |
| **Distal** | Away from the attachment of the limb |
| **Proximal** | Toward the attachment of the limb |
| **Flexion** | Movement of the joint that decreases the angle between two bones |
| **Extension** | Movement of the joint that increases the angle between two bones |

Table 6.1 provides the definitions of the anatomical terms used here. Particularly noteworthy for this discussion are the terms that name planes relative to the human figure: *sagittal, coronal,* and *transverse*.

## 6.1 Reaching and Grasping

One of the most common human figure animation tasks involves movement of the upper limbs. A synthetic figure may be required to reach and operate a control, raise a coffee cup from a table up to his mouth to drink, or turn a complex object over and over to examine it. It is computationally simpler to consider the arm as an appendage that moves independently of the rest of the body. In some cases, this can result in unnatural-looking motion. To produce more realistic motion, the user often involves additional joints of the body in executing the motion. In this section, the arm is considered in isolation. It is assumed that additional joints, if needed, can be added to the reaching motion in a preprocessing step that positions the figure and readies it for independently considered arm motion.

### 6.1.1 Modeling the Arm

The basic model of the human arm, ignoring the joints of the hand for now, can be most simply represented as a seven-degrees-of-freedom (DOF) manipulator (Figure 6.1): three DOFs are at the shoulder joint, one at the elbow, and three at the wrist. See Chapter 4 for an explanation of joint representation, forward kinematics, and inverse kinematics. A *configuration* or *pose* for the arm is a set of seven joint angles, one for each of the seven DOFs of the model.

Forearm rotation presents a problem. In Figure 6.1, the forearm rotation is associated with the wrist. However, in reality, the forearm rotation is not associated with a localized joint like most of the other DOFs of the human figure but rather

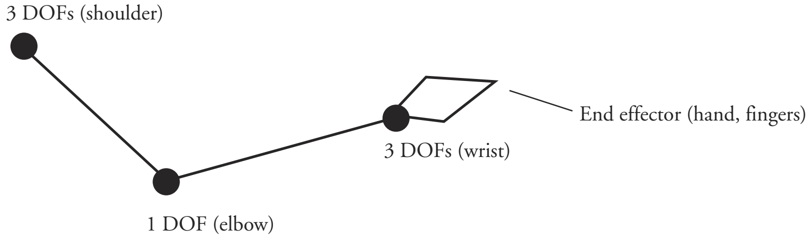


**Figure 6.1** Basic model of the human arm

is distributed along the forearm itself as the two forearm bones (radius and ulna) rotate around each other. Sometimes this rotation is associated with the elbow instead; other implementations create a "virtual" joint midway along the forearm to handle forearm rotation.

Of course, the joints of the human arm have limits. For example, the elbow can flex to approximately 20 degrees and extend to as much as 160 degrees. Allowing a figure's limbs to exceed the joint limits would certainly contribute to an unnatural look. Most joints are positioned with least strain somewhere in the middle of their range and rarely attain the boundaries of joint rotation unless forced. More subtly, joint limits may vary with the position of other joints, and further limits are imposed on joints to avoid intersection of appendages with other parts of the body. For example, if the arm is moved in a large circle parallel to and to the side of the torso, the muscle strain causes the arm to distort the circle at the back. As another example, tendons make it more difficult to fully extend the knee when one is bending at the hip (the motion used to touch one's toes).

If joint limits are enforced, some general motions can be successfully obtained by using forward kinematics. Even if an object is to be carried by the hand, forward kinematics in conjunction with attaching the object to the end effector creates a fairly convincing motion. But if the arm/hand must operate relative to a fixed object, such as a knob, inverse kinematics is necessary. Unfortunately, the normal methods of inverse kinematics, using the pseudo inverse of the Jacobian, is not guaranteed to give humanlike motion. As explained in Chapter 4, in some orientations a singularity may exist in which a degree of freedom is "lost" in Cartesian space. For example, the motion can be hard to control in cases in which the arm is fully extended.

According to the model shown in Figure 6.1, if only the desired end effector position is given, then the solution space is underconstrained. In this case, multiple solutions exist, and inverse kinematic methods may result in configurations that do not look natural. As noted in Chapter 4, there are methods for biasing the solution toward desired joint angles. This helps to avoid violating joint limits and produces more humanlike motion but still lacks any anatomical basis for producing humanlike configurations.

It is often useful to specify the goal position of the wrist instead of the fingers to better control the configurations produced. But even if the wrist is fixed (i.e., treated as the end effector) at a desired location, and the shoulder is similarly fixed, there are still a large number of positions that might be adopted that satisfy both the constraints and the joint limits. Biasing the joint angles to orientations preferable for certain tasks reduces the multiple-solution-problem somewhat.

To more precisely control the movement, the user can specify intermediate positions and orientations for the end effector as well as for intermediate joints. Essentially, this establishes key poses for the linkage. Inverse kinematics can then be used to step from one pose to the next so that the arm is still guided along the path. This affords some of the savings of using inverse kinematics while giving the animator more control over the final motion.

The formal inverse Jacobian approach can be replaced with a more procedural approach based on the same principles to produce more humanlike motion. In human motion, the joints farther away from the end effector (the hand) have the most effect on it. The joints closer to the hand change angles in order to perform the fine orientation changes necessary for final alignment. This can be implemented procedurally by computing the effect of each DOF on the end effector by taking the cross product of the axis of rotation, $\omega_1$, with the vector from the joint to the end effector, $V_1$ (Figure 6.2). In addition, since the arm contains a 1-DOF angle (elbow), a plane between the shoulder, the elbow, and the wrist is formed, and the arm's preferred positions dictate a relatively limited rotation range for that plane. Once the plane is fixed, the shoulder and elbow angles are easy to calculate and can be easily adjusted on that plane (Figure 6.3). Some animation packages (e.g., Maya$^{\text{TM}}$) allow the animator to specify an inverse kinematic solution based on such a plane and to rotate the plane as desired.

Some neurological studies, notably those by Lacquaniti and Soechting [42] and Soechting and Flanders [66], suggest that the arm's posture is determined from the desired location of the end effector (roughly "fixing the wrist's orientation"), and then the final wrist orientation is tweaked for the nature of the object and the task. The model developed by Kondo [41] for this computation makes use of a spherical coordinate system. A set of angles for the shoulder and elbow is calculated from the desired hand and shoulder position and then adjusted if joint limitations are violated. Finally, a wrist orientation is calculated separately. The method is described, along with a manipulation planner for trajectories of cooperating arms, by Koga et al. [40].

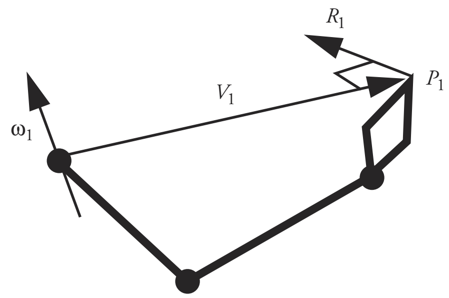


**Figure 6.2** Effect of the first DOF on the end effector: $R_1 = \omega_1 \times V_1$

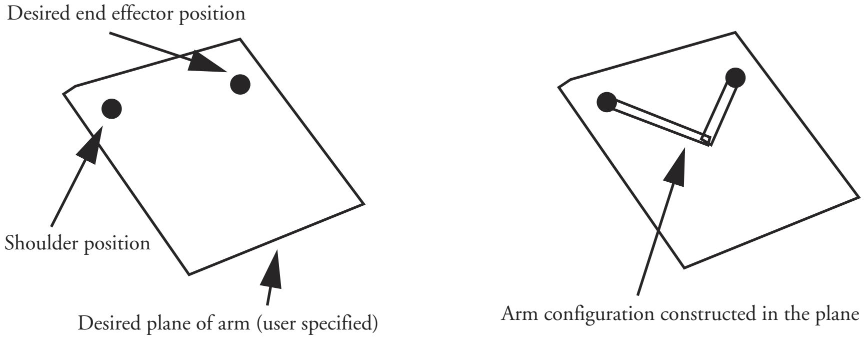

Desired end effector position
Shoulder position
Desired plane of arm (user specified)
Arm configuration constructed in the plane

**Figure 6.3** Constructing the arm in a user-specified plane

## 6.1.2 The Shoulder Joint

The shoulder joint requires special consideration. It is commonly modeled as a ball joint with three coincident degrees of freedom. The human shoulder system is actually more complex. Scheepers [62] describes a more realistic model of the clavicle and scapula along with a shoulder joint, in which three separate joints with limited range provide very realistic-looking arm and shoulder motion. Scheepers also provides a solution to the forearm rotation problem using a radioulnar (mid-forearm) joint. See Figure 6.4.

## 6.1.3 The Hand

To include a fully articulated hand in the arm model, one must introduce many more joints (and thus DOFs). A simple hand configuration may consist of a palm, four fingers, and a thumb, with joints and DOFs as shown in Figure 6.5.

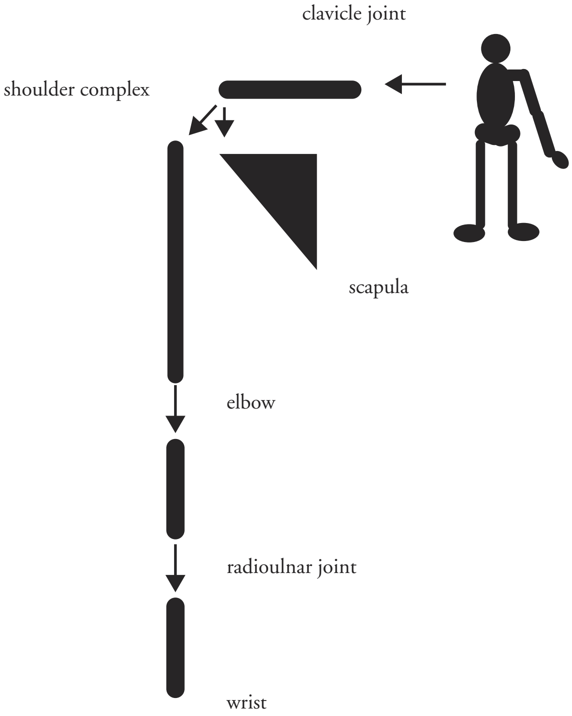


**Figure 6.4** Conceptual model of the upper limb

A model similar to Figure 6.5 is used by Rijpkema and Girard [59] in their work on grasping. Scheepers [62] uses twenty-seven bones, but only sixteen joints are movable parts. Others use models with subtler joints inside the palm area in order to get humanlike action.

If the hand is to be animated in detail, the designer must pay attention to types of grasp and how the grasps are to be used. The opposable thumb provides humans with great manual dexterity: the ability to point, grasp objects of many shapes, and exert force such as that needed to open a large jar of pickles or a small jewelry clasp. This requires carefully designed skeletal systems. Studies of grasping show at least sixteen different categories of grasp, most involving the thumb and one or more fingers. For a given task, the problem of choosing which grasp is best (most efficient and/or most credible) adds much more complexity to the mere ability to form the grasp.

Simpler models combine the four fingers into one surface and may eliminate the thumb (Figure 6.6). This reduces both the display complexity and the motion control complexity. Display complexity, and therefore image quality, can be maintained by using the full-detail hand model but coordinating the movement of all

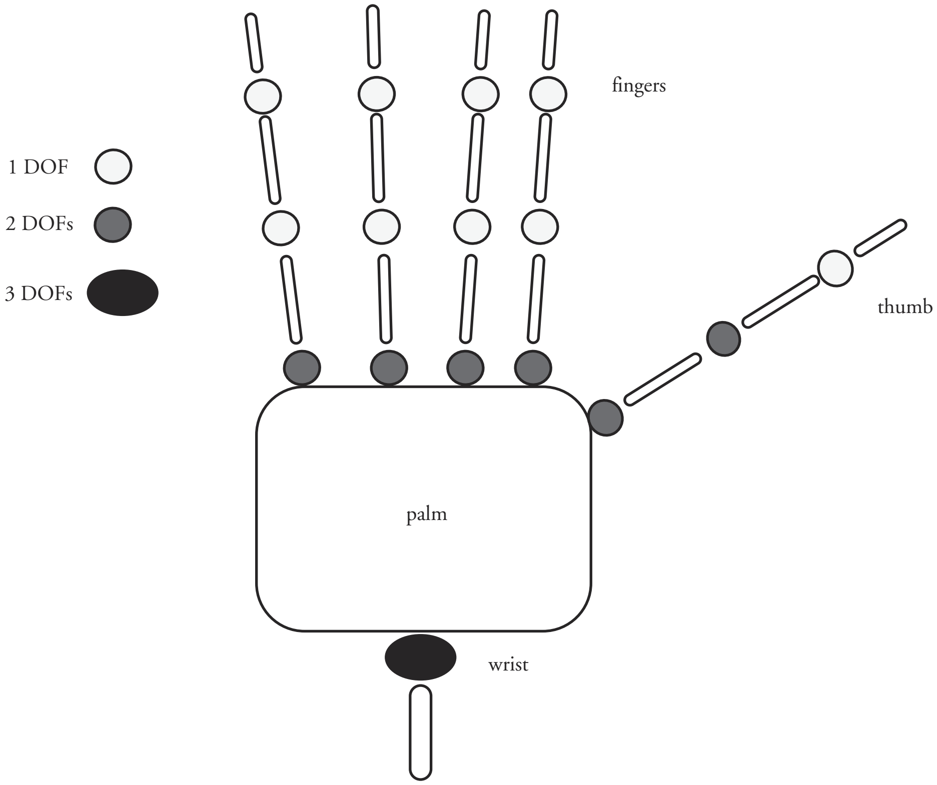


**Figure 6.5** Simple model of hands and fingers

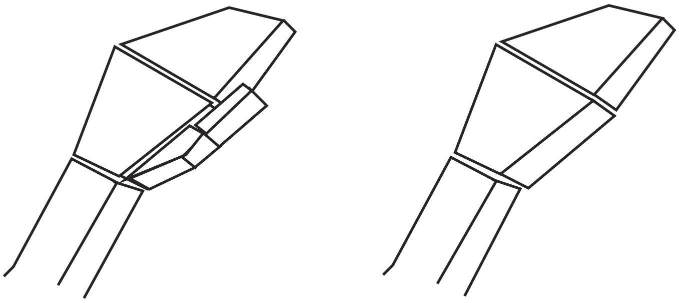


**Figure 6.6** Simplified hands

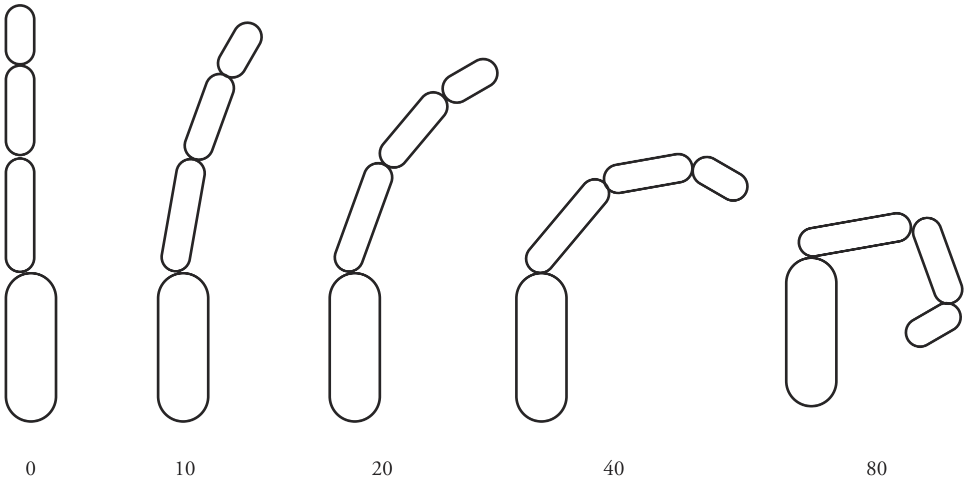


**Figure 6.7** Finger flexion controlled by single parameter; the increase in joint angle (degrees) per joint is shown

the joints of the four fingers with one "grasping" parameter (Figure 6.7), even though this only approximates real grasping action.

### 6.1.4 Coordinated Movement

Adding to the difficulties of modeling and controlling differentiated parts of the upper limb is the difficulty of interjoint cooperation in a movement and assigning any animation to a particular joint. It is easy to demonstrate this difficulty. Stretch out your arm to the side and turn the palm of the hand so it is first facing up; then rotate the hand so it is facing down and try to continue rotating it all the way around so that the palm faces up again. Try to do this motion by involving first only the hand/wrist/forearm and then the upper arm/shoulder. Adding motion of the torso including the clavicle and spine, which involves more DOFs, makes this task simpler, but it also makes the specification of angles to joints more complex. It is difficult to determine exactly what rotation should be assigned to which joints at what time in order to realistically model this motion.

Interaction between body parts is a concern beyond the determination of which joints to use in a particular motion. While viewing the arm and hand as a separate, independent system simplifies the control strategy, its relation to the rest of the body must be taken into account for a more robust treatment of reaching. Repositioning, twisting, and bending of the torso, reactive motions by the other arm, and even counterbalancing by the legs are often part of movements that only appear to belong to a single arm. It is nearly impossible for a person reaching for an object to keep the rest of the body in a fixed position. Rather than extend joints to the edges of their limits and induce stress, other body parts may cooperate to relieve muscle strain or maintain equilibrium.

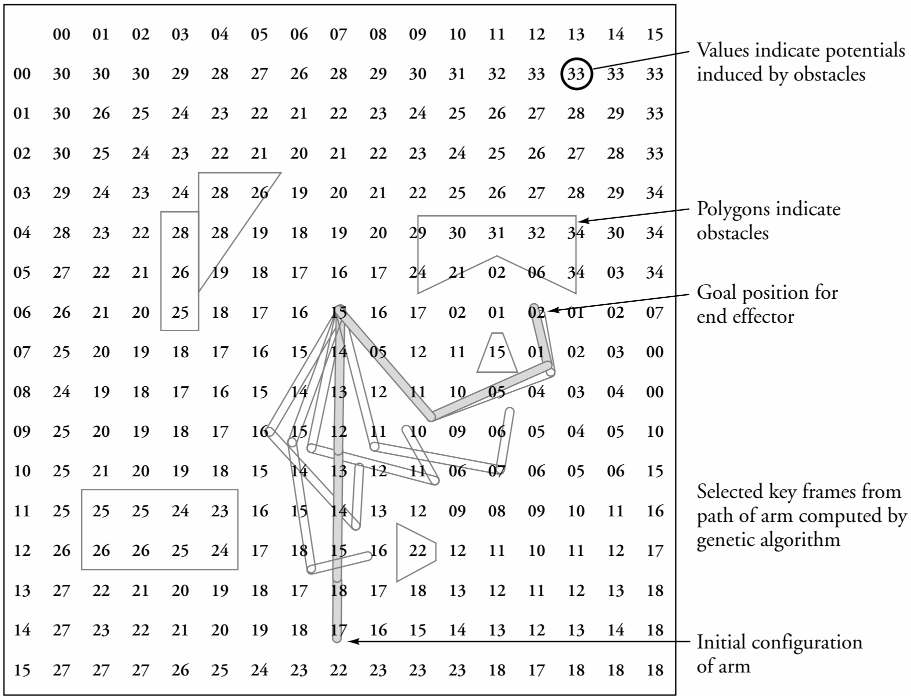
By the same token, arm manipulation is used in many different full-body movements. Even walking, which is often modeled as an activity of the legs only, involves the torso, the arms, and even the head. The arm often seems like a simple and rewarding place to begin modeling human figure animation, but it is difficult to keep the tasks at a simple level.

### 6.1.5 Reaching Around Obstacles

To further complicate the specification and control of reaching motion, there may be obstacles in the environment that must be avoided. Of course, it is not enough to merely plan a collision-free path for the end effector. The entire limb sweeps out a volume of space during reach that must be completely devoid of other objects to avoid collisions. For sparse environments, simple reasoning strategies can be used to determine the best way to avoid obstacles.

As more obstacles populate the environment, more complex search strategies might be employed to determine the path. Various path-planning strategies have been proposed. For example, given an environment with obstacles, an artificial potential field can be constructed as a function of the local geometry. Obstacles impart a high potential field that attenuates based on distance. Similarly, the goal position imparts a low potential into the field. The gradient of the field suggests a direction of travel for the end effector and directs the entire linkage away from collisions (Figure 6.8). Such approaches are susceptible to local minima traps, which various strategies have been used to overcome. Genetic algorithms, for example, have been used to search the space for a global minimum [50]. The genetic fitness function can be tailored to find an optimal path in terms of one of several criteria such as shortest end effector distance traveled, minimum torque, and minimum angular acceleration.

Such optimizations, however, produce paths that would not necessarily be considered humanlike. Optimized paths will typically come as close as possible to critical objects in the path in order to minimize the fitness function. Humans seldom generate such paths in their reaching motions. The complexity of human motion is further complicated by the effect of vision on obstacle avoidance. If the figure “knows” there is an object to avoid but is not looking directly at it, then the reaching motion will incorporate more leeway in the path than it would if the obstacle were directly in the field of view. Furthermore, the cost of collision can influence the resulting path: it costs more to collide with a barbed-wire fence than a towel.



| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 30 | 30 | 30 | 29 | 28 | 27 | 26 | 28 | 29 | 30 | 31 | 32 | 33 | 33 | 33 | 33 |
| 01 | 30 | 26 | 25 | 24 | 23 | 22 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 33 |
| 02 | 30 | 25 | 24 | 23 | 22 | 21 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 33 |
| 03 | 29 | 24 | 23 | 24 | 28 | 26 | 19 | 20 | 21 | 22 | 25 | 26 | 27 | 28 | 29 | 34 |
| 04 | 28 | 23 | 22 | 28 | 28 | 19 | 18 | 19 | 20 | 29 | 30 | 31 | 32 | 34 | 30 | 34 |
| 05 | 27 | 22 | 21 | 26 | 19 | 18 | 17 | 16 | 17 | 24 | 21 | 02 | 06 | 34 | 03 | 34 |
| 06 | 26 | 21 | 20 | 25 | 18 | 17 | 16 | 15 | 16 | 17 | 02 | 01 | 02 | 01 | 02 | 07 |
| 07 | 25 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 05 | 12 | 11 | 15 | 01 | 02 | 03 | 00 |
| 08 | 24 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 05 | 04 | 03 | 04 | 00 |
| 09 | 25 | 20 | 19 | 18 | 17 | 16 | 15 | 12 | 11 | 10 | 09 | 06 | 05 | 04 | 05 | 10 |
| 10 | 25 | 21 | 20 | 19 | 18 | 15 | 14 | 13 | 12 | 11 | 06 | 07 | 06 | 05 | 06 | 15 |
| 11 | 25 | 25 | 25 | 24 | 23 | 16 | 15 | 14 | 13 | 12 | 09 | 08 | 09 | 10 | 11 | 16 |
| 12 | 26 | 26 | 26 | 25 | 24 | 17 | 18 | 15 | 16 | 22 | 12 | 11 | 10 | 11 | 12 | 17 |
| 13 | 27 | 22 | 21 | 20 | 19 | 18 | 17 | 18 | 17 | 18 | 13 | 12 | 11 | 12 | 13 | 18 |
| 14 | 27 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 13 | 14 | 18 |
| 15 | 27 | 27 | 27 | 26 | 25 | 24 | 23 | 22 | 23 | 23 | 23 | 18 | 17 | 18 | 18 | 18 |

Values indicate potentials induced by obstacles

Polygons indicate obstacles

Goal position for end effector

Selected key frames from path of arm computed by genetic algorithm

Initial configuration of arm

**Figure 6.8** Path planning result [50]

## 6.1.6 Strength

As anyone who has ever changed a spark plug in a car knows, avoiding all the obstacles and getting the wrench on the plug is only half the battle. Once in position, the arm and hand must be in a configuration in which there is enough strength available to actually dislodge the plug. To simulate more realistic motions, users incorporate strength criteria into the task planning [44]. As previously noted, typical reaching motion problems are usually underconstrained, allowing for multiple solutions. The solution space can be searched for a specific motion that is acceptable in terms of the amount of strain it places on the figure.

When a specific motion is proposed by a kinematic planner, it can be evaluated according to the strain it places on the body. The strain is determined by computing the torque necessary at each joint to carry out the motion and rating the torque requirements according to desirability. Given the current pose for the figure, the required joint accelerations, and any external forces, the torque required at each joint can be calculated. For each joint, the maximum possible torque for both flexion and extension is given as a function of the joint's angle as well as that of neighboring joints. A *comfort* metric can be formed as the ratio of currently requested torque and maximum possible torque. The *comfort level* for the figure is computed by finding the maximum torque ratio for the entire body. The most desirable motions are those that minimize the maximum torque ratio over the duration of the motion.

Once a motion has been determined to be unacceptable, it must be modified in order to bring its comfort level back to within acceptable ranges. This can be done by initiating one or more strategies that reduce the strain. Assume that a particular joint has been identified that exceeds the accepted comfort range. If other joints in the linkage can be identified that produce a motion in the end effector similar to that of the problem joint and that have excess torque available, then increasing the torques at these joints can compensate for reduced torque at the problem joint. It may also be possible to include more joints in the linkage, such as the spine in a reaching motion, to reformulate the inverse kinematic problem in the hope of reducing the torque at the problem joint. See Lee et al. [44] for details.

## 6.2 Walking

Walking, along with reaching, is one of the most common activities in which the human form engages. It is a complex activity that for humans is learned only after an extended trial-and-error process. An aspect that differentiates walking from typical reaching motions, besides the fact that it uses the legs instead of the arms, is that it is basically cyclic. While its cyclic nature provides some uniformity, acyclic components such as turning and tripping occur periodically. In addition, walking is responsible for transporting the figure from one place to another and is simultaneously responsible for maintaining balance. Thus dynamics plays a much more integral role in the formation of the walking motion than it does in reaching.

An aspect of walking that complicates its analysis and generation is that it is dynamically but not statically stable. This means that if a figure engaged in walking behavior suddenly freezes, the figure is not necessarily in a balanced state and might fall to the ground. For animation purposes, this means that the walking motion cannot be frozen in time and statically analyzed to determine the correct forces and torques that produce the motion. As a result, knowledge of the walking motion, in the form of either empirically gathered data [32] [37] or a set of parameters adjustable by the animator, is typically used as the global control mechanism for walking behavior. Attributes such as stride length, hip rotation, and foot placement can be used to specify what a particular walk should look like. A state transition diagram, or its equivalent, is typically used to transition from phase to phase of the gait [11] [12] [28] [35] [57]. Calculation of forces and torques can then be added, if desired, to make the nuances of the motion more physically accurate and more visually satisfying. Kinematics can be used to entirely control the legs, while the forces implied by the movement of the legs are used to affect the motion of the upper body [28] [67]. Alternatively, kinematics can be used to establish constraints on leg motion such as leg swing duration and foot placement. Then the forces and torques necessary to satisfy the constraints can be used to resolve the remaining degrees of freedom of the legs [11] [35] [57]. In some cases, forward dynamic control can be used after determining the forces and torques necessary to drive the legs from state to state [48].

## 6.2.1 The Mechanics of Locomotion

Understanding the interaction of the various joints involved in locomotion is the first step in understanding and modeling locomotion. The walking and running cycles are presented first. Then the walk cycle is broken down in more detail, showing the complex movements involved.

### Walk Cycle

The walk cycle can be broken down into various phases [11] based on the relation of the feet to their points of contact with the ground. See Figure 6.9. The *stride* is defined by the sequence of motions between two consecutive repetitions of a body configuration [37]. The *left stance* phase of a stride is initiated with the right foot on the ground and the left heel just starting to strike the ground. During this phase, the body is supported by both feet until the right foot pivots up and the right toe leaves the ground. The left stance phase continues as the right foot leaves the ground and starts swinging forward and as the right heel strikes the ground and both feet are once again on the ground. The left toe leaving the ground terminates the left stance phase. The *right swing phase* is the period in which the right toe leaves the ground, the right leg swings forward, and the right heel strikes the ground. Notice that the right swing phase is a subinterval of the left stance phase. The end of the right swing phase initiates the right stance phase, and analogous phases now proceed with the roles of the left leg and the right leg switched. The walking cycle is characterized by alternating periods of single and double support.

### Run Cycle

The run cycle can also be described as a sequence of phases. It differs from the walk cycle in that both feet are off the ground at one time and at no time are both feet on the ground. As in the walk cycle, the *stance* is the duration that a foot is on the ground. Thus, the *left stance,* defined by the left heel strike and left toe lift, has the right foot off the ground. This is followed by a period of *flight,* during which both feet are off the ground, with the right foot swinging forward. The flight is terminated by the right heel strike, which starts the *right stance.* See Figure 6.10.

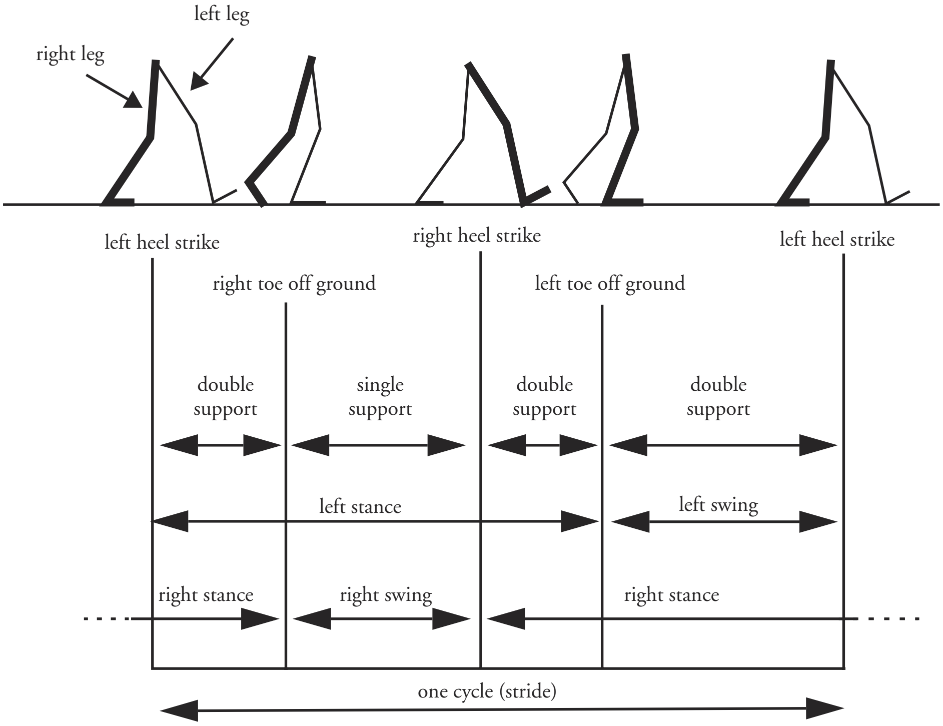


**Figure 6.9** Walk cycle [37]

Notice that the left and right stances do not overlap and are separated by periods of flight.

### Pelvic Transport

For present purposes, let the pelvis represent the mass of the upper body being transported by the legs. Using a simplified representation for the legs, Figure 6.11 shows how the pelvis is supported by the stance leg at various points during the stance phase of the walk cycle. Figure 6.12 shows these positions superposed during a full stride and illustrates the abutting of two-dimensional circular arcs describing the basic path of the pelvis as it is transported by the legs.

### Pelvic Rotation

The pelvis represents the connection between the legs and the structure that separates the legs in the third dimension. Figure 6.13 shows the position of the pelvis during various points in the walking cycle, as viewed from above. The pelvis rotates about a vertical axis centered at the stance leg, helping to lengthen the stride as the swing leg stretches out for its new foot placement. This rotation of the

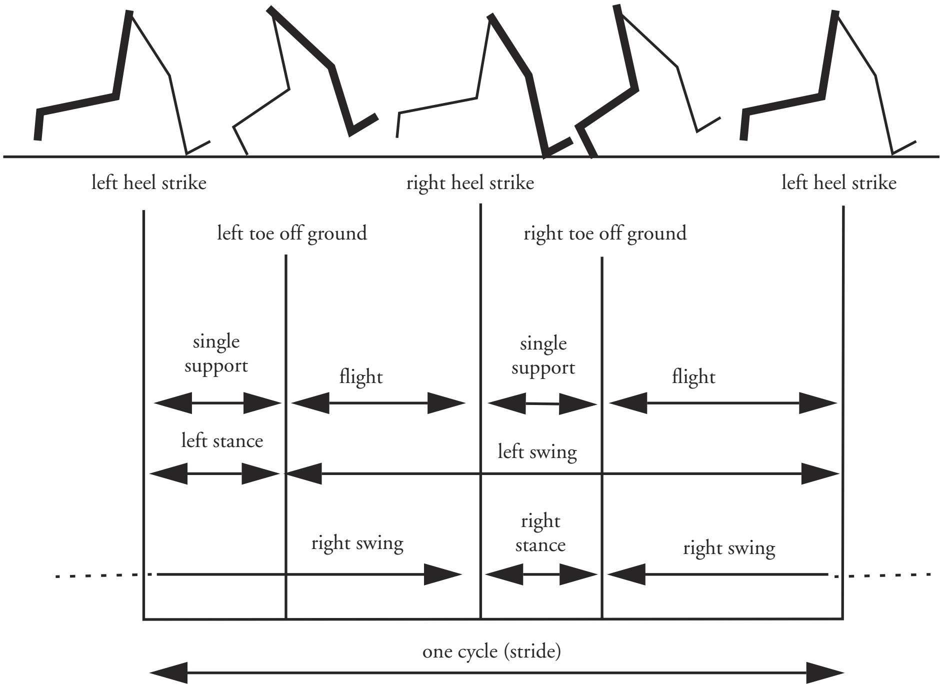
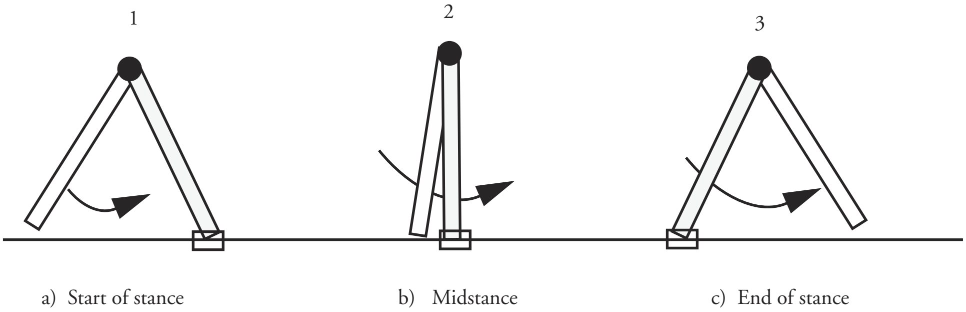


**Figure 6.10** Run cycle [12]



a) Start of stance b) Midstance c) End of stance

**Figure 6.11** Position of pelvis during stance phase (sagittal plane); box indicates supporting contact with floor

pelvis above the stance leg means that the center of the pelvis follows a circular arc relative to the top of that leg. The top of the stance leg is rotating above the point of contact with the floor (Figure 6.13), so the path of the center of the pelvis resembles a sinusoidal curve (Figure 6.14).

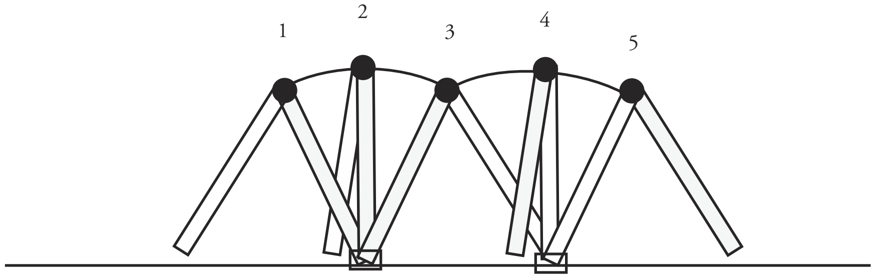


**Figure 6.12** Transport of pelvis by intersecting circular arcs (sagittal plane)

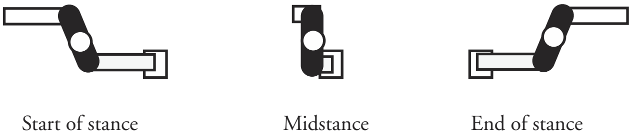


**Figure 6.13** Pelvic orientation during stance phase (transverse plane)

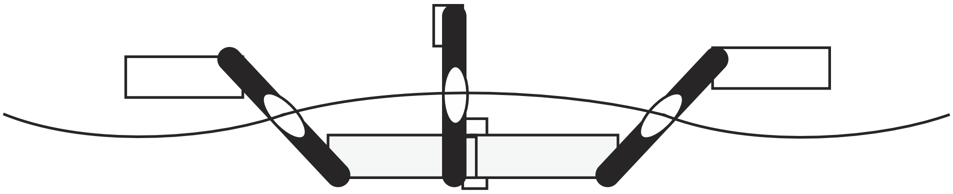


**Figure 6.14** Path of the pelvic center from above (transverse plane), exaggerated for illustrative purposes

## Pelvic List

The transport of the pelvis requires the legs to lift the weight of the body as the pelvis rotates above the point of contact with the floor (Figure 6.15). To reduce the amount of lift, the pelvis lists by rotating in the coronal plane.

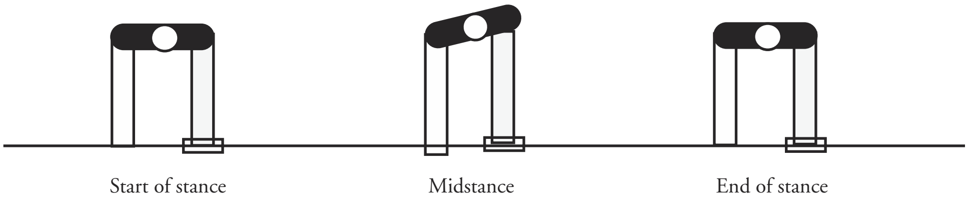


**Figure 6.15** Pelvic list to reduce the amount of lift (coronal plane)

### Knee Flexion

As shown in Figure 6.15, in a pelvic list with one-piece legs, the swing leg would penetrate the floor. Bending at the knee joint (flexion) allows the swing leg to safely pass over the floor and avoid contact (Figure 6.16). Flexion at the knee of the stance leg also produces some leveling out of the pelvic arcs produced by the rotation of the pelvis over the point of contact with the floor. In addition, extension just before contact with the floor followed by flexion of the new stance leg at impact provides a degree of shock absorption.

### Ankle and Toe Joints

The final part of the puzzle to the walking motion is the foot complex, consisting of the ankle, the toes, and the foot itself. This complex comprises several bones and many degrees of freedom and can be simply modeled as two hinge joints per foot (Figure 6.17). The ankle and toe joints serve to further flatten out the rotation of the pelvis above the foot as well as to absorb some shock.

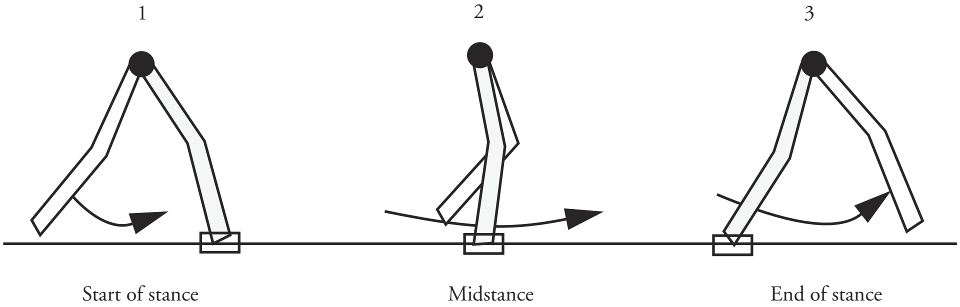


**Figure 6.16** Knee flexion allowing for the swing leg to avoid penetrating the floor, leveling the path of the pelvis over the point of contact, and providing some shock absorption (sagittal plane)

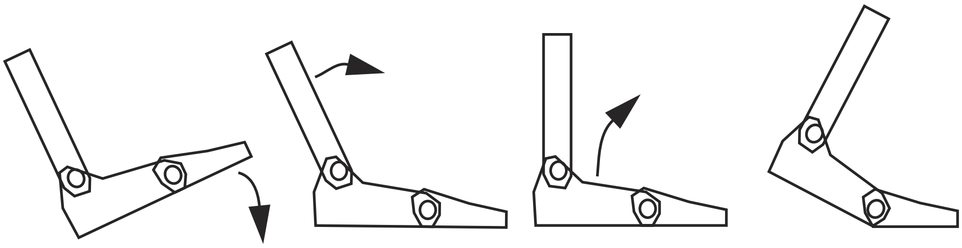


**Figure 6.17** Rotation due to ankle-toe joints

## 6.2.2 The Kinematics of the Walk

Animation of the leg can be performed by appropriate control of the joint angles. As previously mentioned, a leg's walk cycle is composed of a stance phase and a swing phase. The stance phase duration is the time from heel strike to toe lift. The swing phase duration is the time between contact with the ground—from toe lift to heel strike. The most basic approach to generating the walking motion is for the animator to specify a list of joint angle values for each degree of freedom involved in the walk. There are various sources for empirical data describing the kinematics of various walks at various speeds. Figures 6.18 through 6.22, from Inman, Ralson, and Todd [37], graph the angles over time for the various joints involved in the walk cycle, as well as giving values for the lateral displacement of the pelvis [37].

Specifying all the joint angles, either on a frame-by-frame basis or by interpolation of values between key frames, is an onerous task for the animator. In addition, it takes a skilled artist to design values that create unique walks that deviate in any way from precisely collected clinical data. When creating new walks, the animator

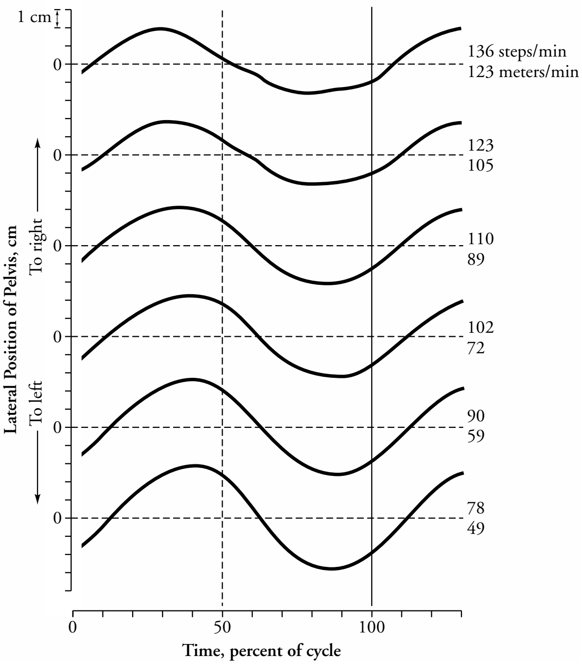


**Figure 6.18** Lateral displacement of pelvis [37]

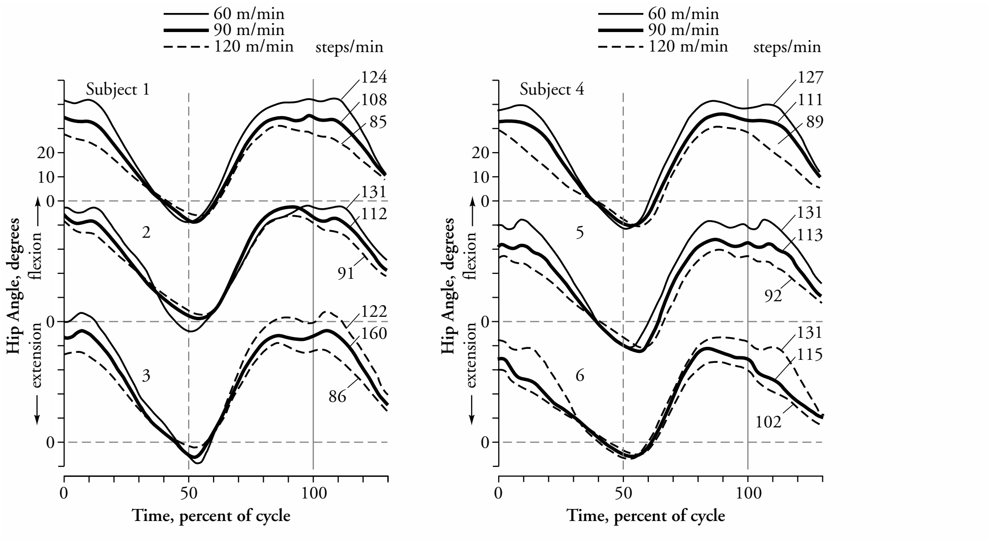


**Figure 6.19** Hip angles [37]

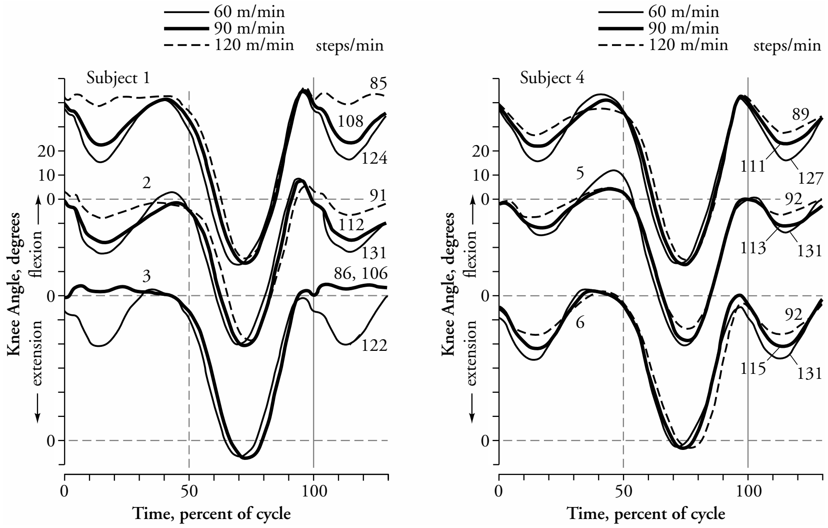


**Figure 6.20** Knee angles [37]

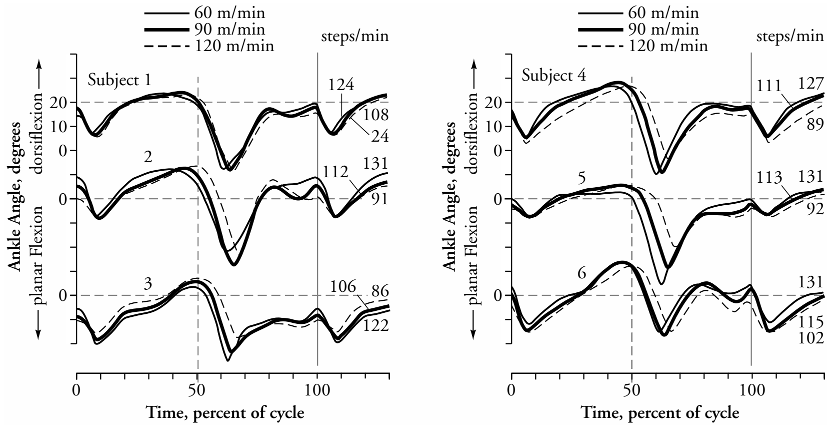


**Figure 6.21** Ankle angles [37]

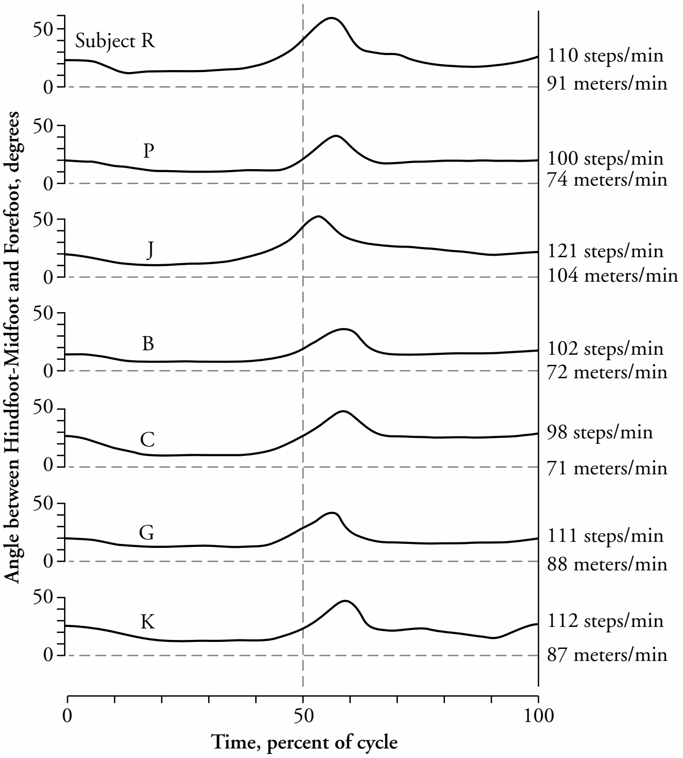


**Figure 6.22** Toe angles [37]

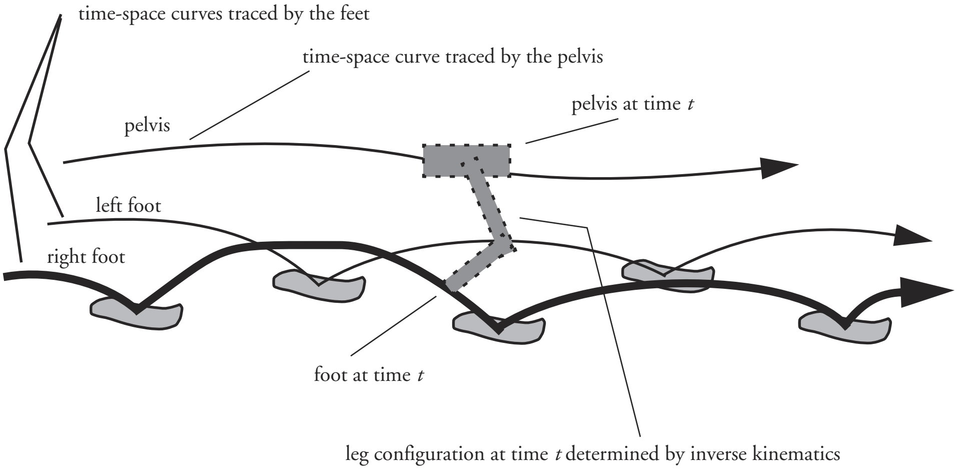

time-space curves traced by the feet
time-space curve traced by the pelvis
pelvis at time *t*
pelvis
left foot
right foot
foot at time *t*
leg configuration at time *t* determined by inverse kinematics

**Figure 6.23** Pelvis and feet constraints satisfied by inverse kinematics

can specify kinematic values such as pelvic movement, foot placement, and foot trajectories. Inverse kinematics can be used to determine the angles of the intermediate joints [12]. By constructing the time-space curves traced by the pelvis and each foot, the user can determine the position of each for a given frame of the animation. Each leg can then be positioned by considering the pelvis fixed and the leg a linked appendage whose desired end effector position is the corresponding position on the foot trajectory curve (Figure 6.23). Sensitivity to segment lengths can cause even clinical data to produce configurations that fail to keep the feet in solid contact with the floor during walking. Inverse kinematics is also useful for forcing clinical data to maintain proper foot placement.

### 6.2.3 Using Dynamics to Help Produce Realistic Motion

Dynamic simulation can be used to map specified actions and constraints to make the movement more accurate physically. However, as Girard and Maciejewski [28] point out, an animator who wants a particular look for a behavior often wants more control over the motion than a total physical simulation provides (Girard and Maciejewski discuss this in relation to walking, but it obviously applies in many situations where physically reasonable, yet artistically controlled, motion is desired). Dynamics must be intelligently applied so that it aids the animator and does not become an obstacle that the animator must work around. In addition, to make the computations tractable, the animator almost always simplifies the dynamics. There are several common types of simplifications: (1) some dynamic effects are ignored, such as the effect of the swing leg on balance; (2) relatively small temporal variations are ignored and a force is considered constant over some time interval, such as the upward push of the stance leg; (3) a complex structure, such as the 7-DOF leg, is replaced for purposes of the dynamic computations by a simplified but somewhat dynamically equivalent structure, such as a 1-DOF telescoping leg; and (4) computing arbitrarily complex dynamic effects is replaced by computing decoupled dynamic components, such as separate horizontal and vertical components, which are computed independently of each other and then summed.

In achieving the proper motion of the upper body, the legs are used to impart forces to the mass of the upper body as carried by the pelvis. An upward force can be imparted by the stance leg on the mass of the upper body at the point of connection between the legs and the torso, that is, the hips [28] [67]. To achieve support of the upper body, the total upward acceleration due to the support of the legs has to cancel to total downward acceleration due to gravity. As simplifying assumptions, the effect of each leg can be considered independent of the other(s), and the upward force of a leg on the upper body during the leg's stance phase can be considered constant. Similarly, horizontal acceleration can be computed independently for each leg and can be considered constant during the leg's stance phase (Figure 6.24). The horizontal force of the legs can be adjusted automatically to produce the average velocity for the body as specified by the animator, but the fluctuations in instantaneous velocity over time help to create visually more appealing motion than constant velocity alone would. The temporal variations in upward and forward acceleration at each hip due to the alternating stance phases can be used to compute pelvic rotation and pelvic list to produce even more realistic motion.

More physics can be introduced into the lower body by modeling the leg dynamics with a telescoping joint (implemented as a parallel spring-damper system) during the stance phase. The upward force of the leg during the stance phase

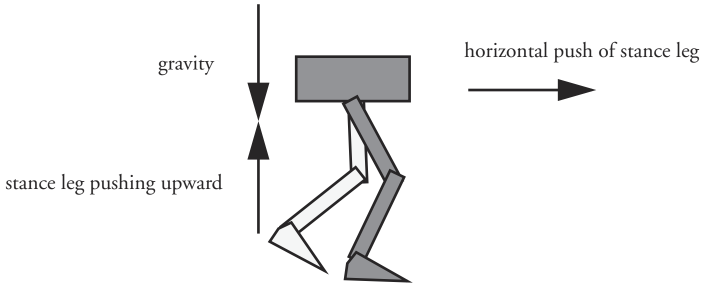


**Figure 6.24** Horizontal and vertical dynamics of stance leg: gravity and the vertical component must cancel over the duration of the cycle; the horizontal push must account for the average forward motion of the figure over the duration of the cycle

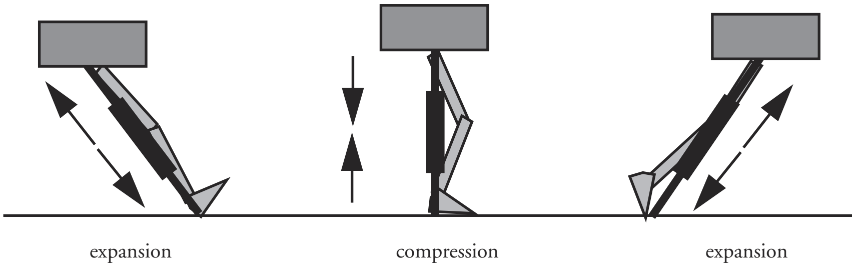


**Figure 6.25** Telescoping joint with kinematically fit leg complex

becomes time-varying as the telescoping joint compresses under the weight of the upper body and expands under the restoring forces of the leg complex (upper leg, lower leg, foot, and associated joints). The telescoping mechanism simulates the shock-absorbing effect of the knee-ankle-toe joints. The leg complex is then fit to the length of the telescoping joint by inverse kinematics (Figure 6.25). During the swing phase, the leg is typically controlled kinematically and does not enter into any dynamic considerations.

Incorporating more physics into the model, the kinematic control information for the figure can be used to guide the motion (as opposed to constraining it). A simple inverse dynamics computation is then used to try to match the behavior of the system with the kinematic control values. Desired joint angles are computed based on high-level, animator-supplied parameters such as speed and the number of steps per unit distance. Joint torques are computed based on proportional-derivative servos (Equation 6.1). The difference between the desired angle, denoted by the underbar, and the current angle at each joint is used to compute the torque to be applied at the next time step. The angular velocities are treated similarly. These torque values are smoothed to prevent abrupt changes in the computed motion. However, choosing good values for the gains ($k_s$, $k_v$) can be difficult and usually requires a trial-and-error approach.

$$\tau = k_s \cdot (\underline{\theta}_i - \theta_i) + k_v \cdot (\dot{\underline{\theta}}_i - \dot{\theta}_i) \qquad \text{(Eq. 6.1)}$$

## 6.2.4 Forward Dynamic Control

In some cases, forward dynamic control instead of kinematic control can be effectively used. Kinematics still plays a role. Certain kinematic states, such as maximum forward extension of a leg, trigger activation of forces and torques. These forces and torques move the legs to a new kinematic state, such as maximum backward extension of a leg, which triggers a different sequence of forces and torques [48]. The difficulty with this approach is in designing the forces and torques necessary to produce a reasonable walk cycle (or other movement, for that matter). In some cases it may be possible to use empirical data found in the biomechanics literature as the appropriate force and torque sequence.

### 6.2.5 Summary

Implementations of algorithms for procedural animation of walking are widely available in commercial graphics packages. However, none of these could be considered to completely solve the locomotion problem, and many issues remain for ongoing or future research. Walking over uneven terrain and walking around arbitrarily complex obstacles are difficult problems to solve in the most general case. The coordinated movement required for climbing is especially difficult. A recurring theme of this chapter is that developing general, robust computational models of human motion is difficult, to say the least.

## 6.3 Facial Animation

Realistic facial animation is one of the most difficult tasks facing computer animators. The face is a very familiar structure that also exhibits a well-defined underlying structure but allows for a large amount of variation between individuals. It is a complex, deformable shape having distinct parts that articulate and is an important component in modeling a figure because it is the main instrument for communication and for defining a person's character. There is also a need to realistically animate the movement of the lips and surrounding face during speech (lip-synching). A good facial model must be capable of geometrically representing a specific person (called *conformation* by Parke [55], *static* by others [60]).

Facial models can be used for cartooning, for realistic character animation, for telecommunications to reduce bandwidth, and for human-computer interaction (HCI). In cartooning, facial animation has to be able to convey expression and personality. In realistic character animation, the geometry and movement of the face must adhere to the constraints of realistic human anatomy. Telecommunications and HCI have the added requirement that the facial model and motion must be computationally efficient. In some applications, the model must correspond closely to a specific target individual.

In addition to the issues addressed by other animation tasks, facial animation often has the constraint of precise timing with respect to a sound track because of lip-synching. Despite its name, lip-synching involves more than just the lips; the rigid articulation of the jaw and the muscle deformation of the tongue must also be considered.

If a cartoon type of animation is desired, a simple geometric shape for the head (such as a sphere) coupled with the use of animated texture maps often suffices for facial animation. The eyes and mouth can be animated using a series of texture maps applied to a simple head shape. See Figure 6.26 (Plate 7). The nose and ears may be part of the head geometry, or, simpler still, they may be incorporated into the texture map. Stylized models of the head may also be used that mimic the basic mechanical motions of the human face, using only a pivot jaw and rotating spheres for eyeballs. Eyelids can be skin-colored hemispheres that rotate to enclose the visible portion of the eyeball. The mouth can be a separate geometry positioned on the surface of the face geometry, and it can be animated independently or sequentially replaced in its entirety by a series of mouth shapes to simulate motion of deformable lips. See Figure 6.27 (Plate 8). These approaches are analogous to techniques used in conventional hand-drawn and stop-motion animation.

For more realistic facial animation, more complex facial models are used whose surface geometry more closely corresponds to that of a real face. And the animation of these models is correspondingly more complex. For an excellent in-depth presentation of facial animation see Parke and Waters [56]. An overview is given here.

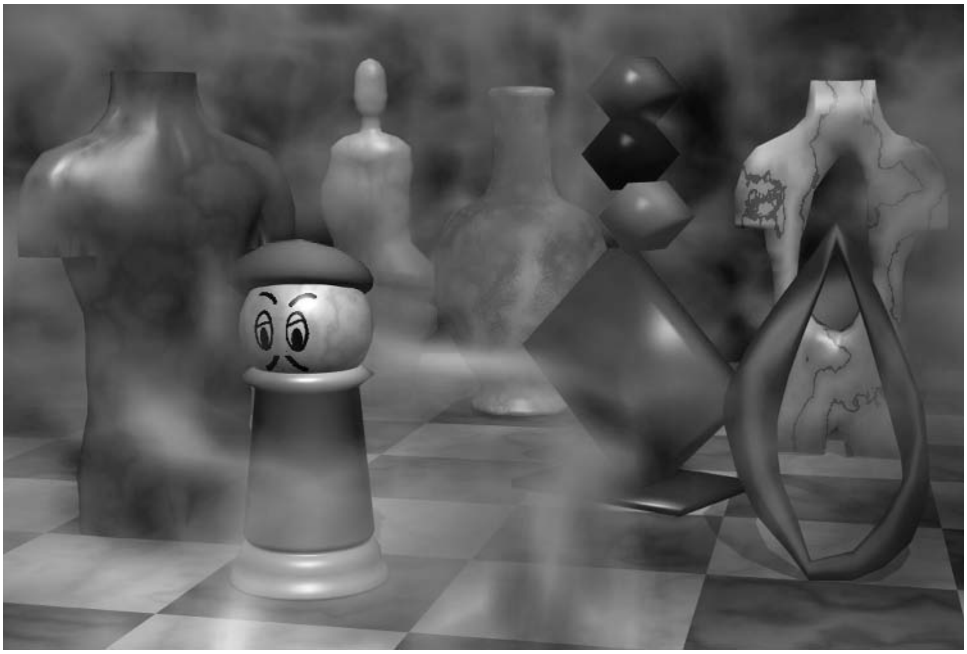


**Figure 6.26** Texture-mapped facial animation from *Getting into Art* (Copyright 1990 David S. Ebert [21])

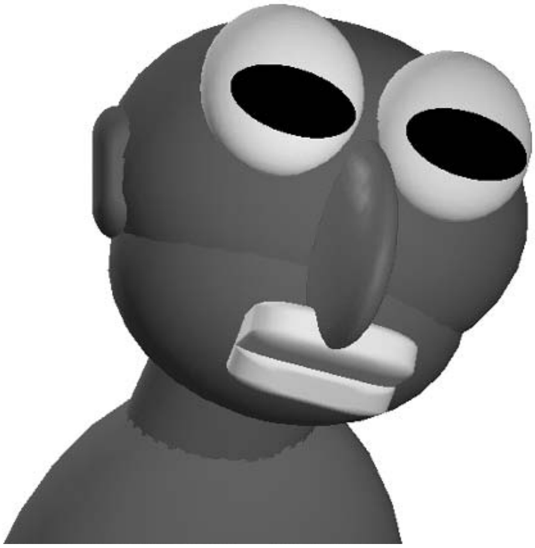


**Figure 6.27** Cartoon face

## 6.3.1 Types of Facial Models

The first problem confronting an animator in facial animation is creating the geometry of the facial model to make it suitable for animation. This in itself can be very difficult. Facial animation models vary widely, from simple geometry to anatomy based. Generally, the complexity is dictated by the intended use. When deciding on the construction of the model, important factors are geometry data acquisition method, motion control and corresponding data acquisition method, rendering quality, and motion quality. The first factor concerns the method by which the actual geometry of the subject's or character's head is obtained. The second factor concerns the method by which the data describing changes to the geometry are obtained. The quality of the rendered image with respect to smoothness and surface attributes is the third concern. The final concern is the corresponding quality of the computed motion.

The model can be discussed in terms of its static properties and its dynamic properties. The statics deal with the geometry of the model in its neutral form, while the dynamics deal with the deformation of the geometry of the model during animation. Three main methods have been used to deal with the geometry of the model. Polygonal models are used most often for their simplicity (e.g., [55] [56] [70] [71]); splines are chosen when a smooth surface is desired. Subdivision surfaces have also been used recently with some success.

Polygonal models are relatively easy to create and can be deformed easily. However, the smoothness of the surface is directly related to the complexity of the model, and polygonal models are visually inferior to other methods of modeling the facial surface. Currently, data acquisition methods sample only the surface, producing discrete data, and surface fitting techniques are subsequently applied.

Spline models typically use bicubic, quadrilateral surface patches, such as Bezier or B-spline, to represent the face. While surface patches offer the advantage of low data complexity in comparison to polygonal techniques when generating smooth surfaces, they have several disadvantages when it comes to modeling an object such as the human face. With standard surface patch technology, a rectangular grid of control points is used to model the entire object. As a result, it is difficult to maintain low data complexity while incorporating small details and sharp localized features, because entire rows and/or entire columns of control information must be modified. Thus, a small addition to one local area of the surface to better represent a facial feature means that information has to be added across the entire surface.

Hierarchical B-splines, introduced by Forsey and Bartels [27], are a mechanism by which local detail can be added to a B-spline surface while avoiding the global modifications required by standard B-splines. Finer resolution control points are carefully laid over the coarser surface while continuity is carefully maintained. In this way, local detail can be added to a surface. The organization is hierarchical, so finer and finer detail can be added. The detail is defined relative to the coarser surface so that editing can take place at any level.

Subdivision surfaces (e.g., [19]) use a polygonal control mesh that is refined, in the limit, to a smooth surface. The refinement can be terminated at an intermediate resolution and rendered as a polygonal mesh. Subdivision surfaces have the advantage of being able to create local complexity without global complexity. They provide an easy-to-use, intuitive interface for developing new models, and provisions for discontinuity of arbitrary order can be accommodated [19]. However, they are difficult to interpolate to a specific data set, which makes modeling a specific face problematic.

Implicitly defined surfaces have also been used to model faces, but such models typically become increasingly complex when the animator tries to deal with small details and sharp features. Where photorealism is not the objective and a caricature or a more "cartoony" model is desired, implicits show promise as a modeling technique. However, the animation of such models remains a challenge.

### 6.3.2 Creating the Model

Creating a model of a human head from scratch is not easy. Not only must the correct shape be generated, but when facial animation is the objective, the geometric elements (vertices, edges) must be placed appropriately so that the motion of the surface can be controlled precisely. If the model is dense in the number of geometric elements used, then the placement becomes less of a concern, but in relatively low resolution models it can be an issue. Of course, one approach is to use a CAD system and let the user construct the model. This is useful when the model to be constructed is a fanciful creature or a caricature or must meet some aesthetic design criteria. While this approach gives an artist the most freedom in creating a model, it requires the most skill.

Besides the CAD approach, there are two main methods for creating facial models: digitization using some physical reference and modification of an existing model. The former is useful when the model of a particular person is desired; the latter is useful when animation control is already built into a generic model.

As with any model, a physical sculpture of the desired object can be generated with clay, wood, or plaster and then digitized, most often using a mechanical or magnetic digitizing device. A 2D surface-based coordinate grid can be drawn on the physical model, and the polygons can be digitized on a polygon-by-polygon basis. Postprocessing can identify unique vertices, and a polygonal mesh can be easily generated. The digitization process can be fairly labor intensive when large numbers of polygons are involved, which makes using an actual person's face a bit problematic. If a model is used, this approach still requires some artistic talent to generate the physical model, but it is easy to implement at a relatively low cost if small mechanical digitizers are used.

Laser scanners use a laser to calculate distance to a model surface and can create very accurate models. They have the advantage of being able to directly digitize a person's face. The scanners sample the surface at regular intervals to create an unorganized set of surface points. The facial model can be constructed in a variety of ways. Polygonal models are most commonly generated. Scanners have the added advantage of being able to capture color information that can be used to generate a texture map. This is particularly important with facial animation: a texture map can often cover flaws in the model and motion. Laser scanners also have drawbacks; they are expensive, bulky, and require a physical model.

Muraki [51] presents a method for fitting a blobby model (implicitly defined surface formed by summed, spherical density functions) to range data by minimizing an energy function that measures the difference between the isosurface and the range data. By splitting primitives and modifying parameters, the user can refine the isosurface to improve the fit.

Models can also be generated from photographs. This has the advantage of not requiring the presence of the physical model once the photograph has been taken, and it has applications for video conferencing and compression. While most of the photographic approaches modify an existing model by locating feature points, a common method of generating a model from scratch is to take front and side

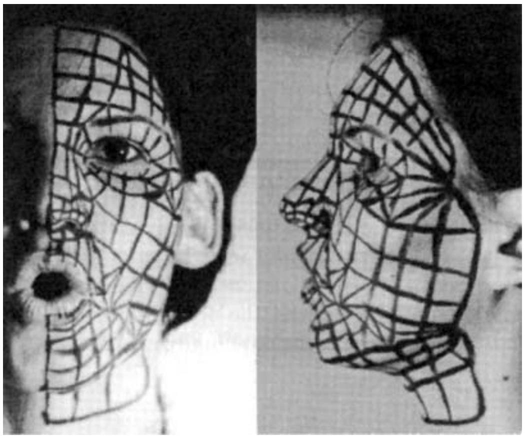


**Figure 6.28** Photographs from which a face may be digitized [56]

images of a face on which grid lines have been drawn (Figure 6.28). Point correspondences can be established between the two images either interactively or by locating common features automatically, and the grid in three-space can be reconstructed. Symmetry is usually assumed for the face, so only one side view is needed and only half of the front view is considered.

Modifying an existing model is a popular technique for generating a face model. Of course, someone had to first generate a generic model. But once this is done, if it is created as a parameterized model and the parameters were well designed, the model can be used to try to match a particular face, to design a face, or to generate a family of faces. In addition, the animation controls can be built into the model so that they require little or no modification of the generic model for particular instances.

One of the most often used approaches to facial animation employs a parameterized model originally created by Parke [54] [55].The parameters for his model of the human face are divided into two categories: *conformational* and *expressive*. The conformational parameters are those that distinguish one individual's head and face from another's. The expressive parameters are those concerned with animation of an individual's face; these are discussed later. There are twenty-two conformational parameters in Parke's model. Again, symmetry between the sides of the face is assumed. Five parameters control the shape of the forehead, cheekbone, cheek hollow, chin, and neck. There are thirteen scale distances between facial features:[1] head *x,y,z;* chin to mouth and chin to eye; eye to forehead; eye *x* and *y;*

[1] In Parke's model, the *z*-axis is up, the *x*-axis is oriented from the back of the head toward the front, and the *y*-axis is from the middle of the head out to the left side.

widths of the jaw, cheeks, nose bridge, and nose nostril. Five parameters translate features of the face: chin in *x* and *z;* end of nose *x* and *z;* eyebrow *z.* Even these are not enough to generate all possible faces, although they can be used to generate a wide variety.

Parke's model was not developed based on any anatomical principles but from the intuitions from artistic renderings of the human face. Facial anthropomorphic statistics and proportions can be used to constrain the facial surface to generate realistic geometries of a human head [18]. Variational techniques can then be used to create realistic facial geometry from a deformed prototype that fits the constraints. This approach is useful for generating heads for a crowd scene or a background character. It may also be useful as a possible starting point for some other character; however, the result will be influenced heavily by the prototype used.

The MPEG-4 standard proposes tools for efficient encoding of multimedia scenes. It includes a set of *Facial Definition Parameters* (*FDPs*) [26] that are devoted mainly to facial animation for purposes of video teleconferencing. Figure 6.29 shows the feature points defined by the standard. Once the model is defined in this way, it can be animated by an associated set of *Facial Animation Parameters* (*FAP*s) [25], also defined in the MPEG-4 standard. MPEG-4 defines sixty-eight FAPs. The FAPs control rigid rotation of the head, eyeballs, eyelids, and mandible. Other low-level parameters indicate the translation of a corresponding feature point, with respect to its position in the neutral face, along one of the coordinate axes [17].

One other interesting approach to generating a model of a face from a generic model is fitting it to images in a video sequence [18]. While not a technique developed for animation applications, it is useful for generating a model of a face of a specific individual. A parameterized model of a face is set up in a 3D viewing configuration closely matching that of the camera that produced the video images. Feature points are located on the image of the face in the video and are also located on the 3D synthetic model. Camera parameters and face model parameters are then modified to more closely match the video by using the pseudo inverse of the Jacobian. (The Jacobian is the matrix of partial derivatives that relates changes in parameters to changes in measurements.) By computing the difference in the measurements between the feature points in the image and the projected feature points from the synthetic setup, the pseudo inverse of the Jacobian indicates how to change the parametric values to reduce the measurement differences.

### 6.3.3 Textures

Texture maps are very important in facial animation. Most objects created by computer graphics techniques have a plastic or metallic look, which, in the case of facial animation, seriously detracts from the believability of the image. Texture

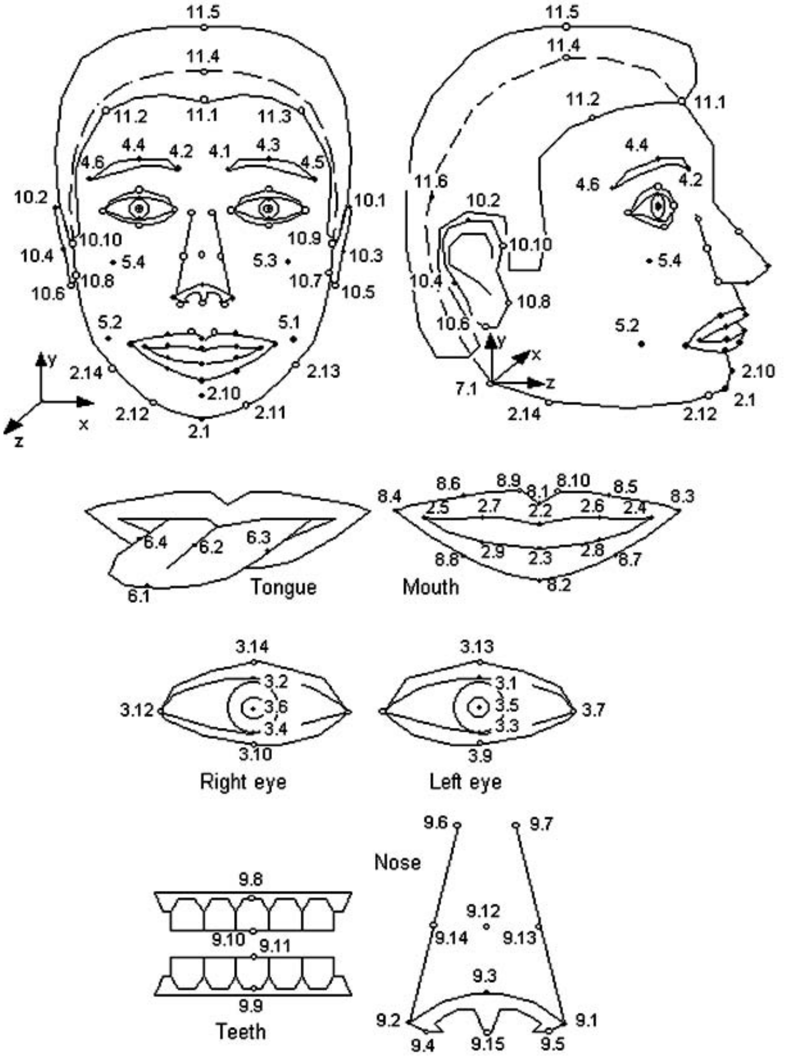


**Figure 6.29** Feature points corresponding to the MPEG-4 Facial Definition Parameters [26]

maps can give a facial model a much more organic look and can give the observer more visual cues when interacting with the images. The texture map can be taken directly from a person's head; however, it must be registered with the geometry. The lighting situation during digitization of the texture must also be considered.

Laser scanners are capable of collecting information on intensity as well as depth, resulting in a high-resolution surface with a matching high-resolution texture. However, once the face deforms, the texture no longer matches exactly. Since the scanner revolves around the model, the texture resolution is evenly spread over the head. However, places are missed where the head is self-occluding (at the ears and maybe the chin) and at the top of the head.

Texture maps can also be created from photographs by simply combining top and side views using pixel blending where the textures overlap [1]. Lighting effects must be taken into consideration, and, because the model is not captured in the same process as the texture map, registration with a model is an issue. Using a sequence of images from a video can improve the process.

## 6.3.4 Approaches to Animating the Face

The simplest approach to facial animation is to define a set of key poses. Facial animation is produced by selecting two of the key poses and interpolating between the positions of their corresponding vertices in the two poses. This restricts the available motions to be the interpolation from one key pose to another. To generalize this a bit more, a weighted sum of two or more key poses can be used in which the weights sum to one. Each vertex position is then computed as a linear combination of its corresponding position in each of the poses whose weight is nonzero. This can be used to produce facial poses not directly represented by the keys. However, this is still fairly restrictive because the various parts of the facial model are not individually controllable by the animator. The animation is still restricted to those poses represented as a linear combination of the keys. If the animator allows for a wide variety of facial motions, the key poses quickly increase to an unmanageable number. This raises the questions: What are the primitive motions of the face? And how many degrees of freedom are there in the face?

### The Facial Action Coding System

The Facial Action Coding System (FACS) is the result of research conducted by the psychologists Ekman and Friesen [22] with the objective of deconstructing all facial expressions into a set of basic facial movements. These movements, called Action Units, or AUs, when considered in combinations, can be used to describe all facial expressions.

Forty-six AUs are identified in the study, and they provide a clinical basis from which to build a facial animation system. Examples of AUs are brow lowerer, inner brow raiser, wink, cheek raiser, upper lip raiser, and jaw drop. See Figure 6.30 for an example of diagrammed AUs. Given the AUs, an animator can build a facial model that is parameterized according to the motions of the AUs. A facial animation system can be built by giving a user access to a set of variables that are in one-to-one correspondence with the AUs. A parametric value controls the amount of the facial motion that results from the associated AU. By setting a variable for each

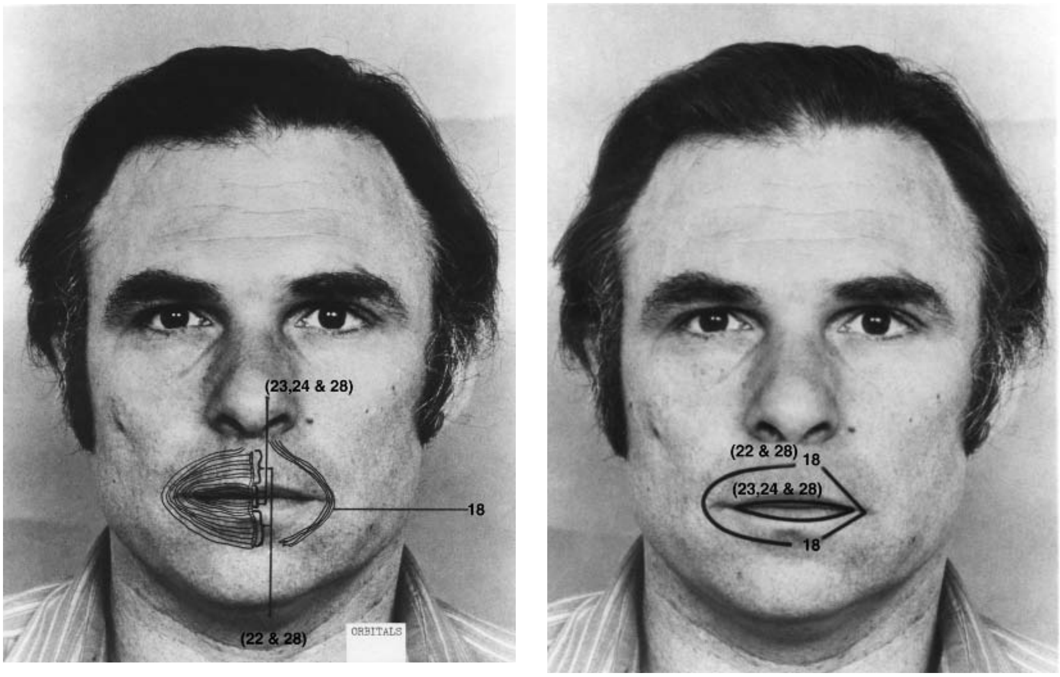


**Figure 6.30** Three Action Units of the lower face [22]

AU, the user can generate all the facial expressions analyzed by Ekman and Friesen. By using the value of the variables to interpolate the degree to which the motion is realized and by interpolating their value over time, the user can then animate the facial model. By combining the AUs in nonstandard ways, the user can also generate many truly strange expressions.

While this work is impressive and is certainly relevant to facial animation, two of its characteristics should be noted before it is used as a basis for such a system. First, the FACS is meant to be descriptive of an expression, not generative. The FACS is not time based, and facial movements are analyzed only relative to a neutral pose. This means that the AUs were not designed to animate a facial model in all the ways that an animator may want control. Second, the FACS describes facial expressions, not speech. The movements for forming individual phonemes, the basic units of speech, were not specifically incorporated into the system. While the AUs provide a good start for describing the basic motions that must be in a facial animation system, they were never intended for this purpose.

### Parameterized Models

As introduced in the discussion of the FACS, parameterizing the facial model according to primitive actions and then controlling the values of the parameters over time is one of the most common ways to implement facial animation. Abstractly, any possible or imaginable facial contortion can be considered as a point in an *n*-dimensional space of all possible facial poses. Any parameterization of a space should have complete coverage and be easy to use. Complete coverage means that the space reachable by (linear) combinations of the parameters includes all (at least most) of the interesting points in that space. Of course, the definition of the word *interesting* may vary from application to application, so a generally useful parameterization includes as much of the space as possible. For a parameterization to be easy to use, the set of parameters should be as small as possible, the effect of each parameter should be independent of the effect of any other parameter, and the effect of each parameter should be intuitive. Of course, in something as complex as facial animation, attaining all of these objectives is probably not possible, so determining appropriate trade-offs is an important activity in designing a parameterization. Animation brings an additional requirement to the table: the animator should be able to generate common, important, or interesting motions through the space by manipulating one or just a few parameters.

The most popular parameterized facial model is credited to Parke [54] [55] [56] and has already been discussed in terms of creating facial models based on the so-called *conformational parameters* of a generic facial model. In addition to the conformational parameters, there are *expression parameters*. Examples of expression parameters are upper-lip position, eye gaze, jaw rotation, and eyebrow separation. Figure 6.31 shows a diagram of the parameter set with the (interpolated) expres-

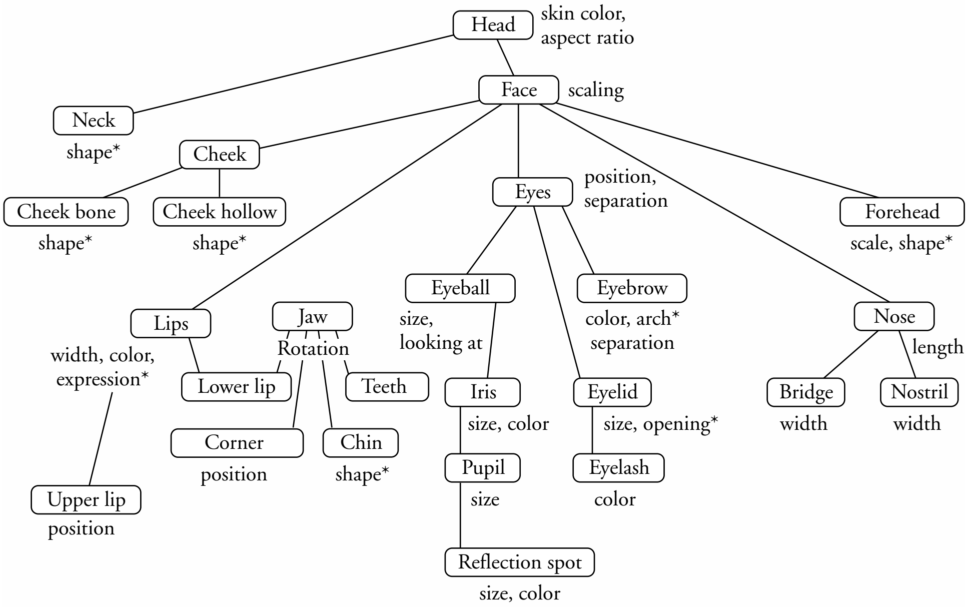


**Figure 6.31** Parke model; * indicates interpolated parameters [56]

sion parameters identified. Most of the parameters are concerned with the eyes and the mouth, where most facial expression takes place. With something as complex as the face, it is usually possible to animate interesting expressions with a single parameter. Experience with the parameter set is necessary for understanding the relationship between a parameter and the facial model. Higher-level abstractions can be used to aid in animating common motions.

### Muscle Models

Parametric models encode geometric displacement of the skin in terms of an arbitrary parametric value. Muscle-based models are more sophisticated, although there is wide variation in the reliance on a physical basis for the models. There are typically three types of muscles that need to be modeled for the face: linear, sheet, and sphincter. The linear model is a muscle that contracts and pulls one point (the *point of insertion*) toward another (the *point of attachment*). The sheet muscle acts as a parallel array of muscles and has a line of attachment at each of its two ends rather than a single point of attachment as in the linear model. The sphincter muscle contracts radially toward an imaginary center. The user, either directly or indirectly, specifies muscle activity to which the facial model reacts. Three aspects differentiate one muscle-based model from another: the geometry of the muscle-skin arrangement, the skin model used, and the muscle model used.

The main distinguishing feature in the geometric arrangement of the muscles is whether they are modeled on the surface of the face or whether they are attached to a structural layer beneath the skin (e.g., bone). The former case is simpler in that only the surface model of the face is needed for the animation system (Figure 6.32). The latter case is more anatomically correct and thus promises more accurate results, but it requires much more geometric structure in the model and is therefore much more difficult to construct (Figure 6.33).

The model used for the skin will dictate how the area around the point of insertion of a (linear) muscle reacts when that muscle is activated; the point of insertion will move an amount determined by the muscle. How the deformation propagates along the skin as a result of that muscle determines how rubbery or how plastic the

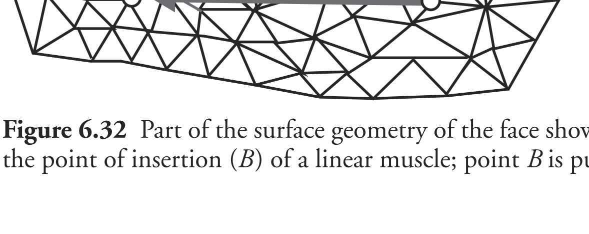


**Figure 6.32** Part of the surface geometry of the face showing the point of attachment (*A*) and the point of insertion (*B*) of a linear muscle; point *B* is pulled toward point *A*.

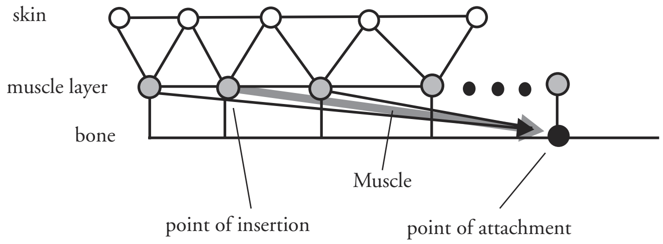


**Figure 6.33** Cross section of the trilayer muscle as presented by Parke and Waters [56]; the muscle only directly affects nodes in the muscle layer.

surface will appear. The simplest model to use is based on geometric distance from the point and deviation from the muscle vector. For example, the effect of the muscle may attenuate based on the distance a given point is from the point of insertion and on the angle of deviation from the displacement vector of the insertion point. See Figure 6.34 for sample calculations. A slightly more sophisticated skin model might model each edge of the skin geometry as a spring and control the propagation of the deformation based on spring constants. The insertion point is moved by the action of the muscle, and this displacement creates restoring forces in the springs attached to the insertion point, which moves the adjacent vertices, which in turn moves the vertices attached to them, and so on. See Figure 6.35. The more complicated Voight model treats the skin as a viscoelastic element by

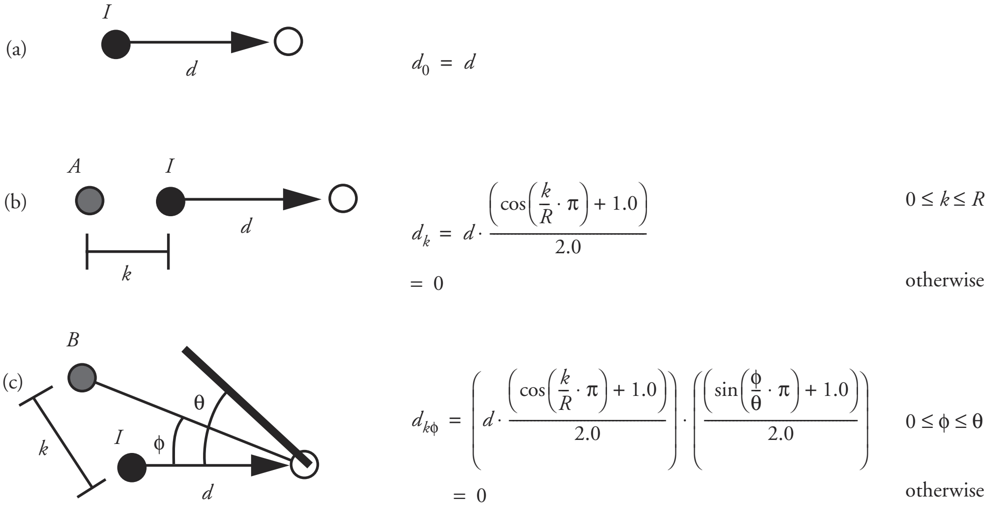
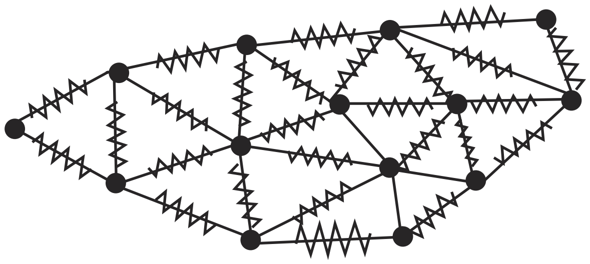
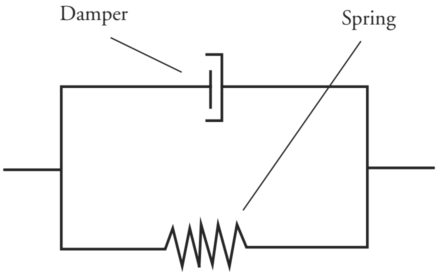

(a)
$I$
$d$
$d_0 = d$
(b)
$A$
$I$
$d$
$k$
$d_k = d \cdot \dfrac{\left(\cos\left(\frac{k}{R} \cdot \pi\right) + 1.0\right)}{2.0}$
$0 \le k \le R$
$= 0$
otherwise
(c)
$B$
$\theta$
$I$
$\phi$
$k$
$d$
$d_{k\phi} = \left(d \cdot \dfrac{\left(\cos\left(\frac{k}{R} \cdot \pi\right) + 1.0\right)}{2.0}\right) \cdot \left(\dfrac{\left(\sin\left(\frac{\phi}{\theta} \cdot \pi\right) + 1.0\right)}{2.0}\right)$
$0 \le \phi \le \theta$
$= 0$
otherwise

**Figure 6.34** Sample attenuation: (a) insertion point $I$ is moved $d$ by muscle; (b) point $A$ is moved $d_k$ based on linear distance from the insertion point; and (c) point $B$ is moved $d_{k\phi}$ based on the linear distance and the deviation from the insertion point displacement vector



$F_i = k_i \cdot (|V_i^1 - V_i^2| - d_i)$ for the $i$th spring

**Figure 6.35** Spring mesh as skin model; the displacement of the insertion point propagates through the mesh according to the forces imparted by the springs



$F_i = k_i \cdot (|V_i^1 - V_i^2| - d_i) + n_i \cdot \frac{d}{dt}(|V_i^1 - V_i^2|)$

**Figure 6.36** Voight viscoelastic model; the motion induced by the spring forces is damped; variables *k* and *n* are spring and damper constants, respectively; and $d_i$ is the rest length for the spring

combining a spring and a damper in parallel (Figure 6.36). The movement induced by the spring is damped as a function of the change in length of the edge.

The muscle model determines the function used to compute the contraction of the muscle. The alternatives for the muscle model are similar to those for the skin, with the distinction that the muscles are active elements, whereas the skin is composed of passive elements. Using a linear muscle as an example, the displacement of the insertion point is produced as a result of muscle activation. Simple models for the muscle will simply specify a displacement of the insertion point based on activation amount. More physically accurate muscle models will compute the effect of muscular forces. The simplest dynamic model uses a spring to represent the muscle. Activating the muscle results in a change of its rest length so as to induce a force at the point of insertion. More sophisticated muscle models include

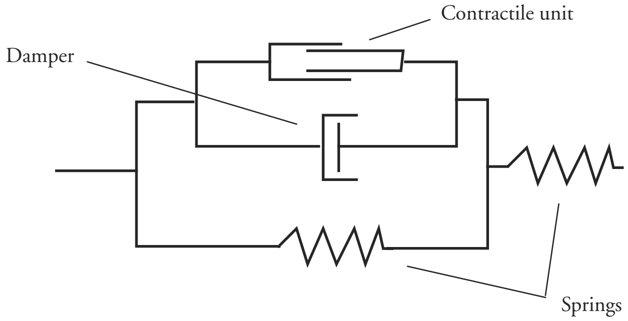


**Figure 6.37** Hill's model for the muscle

damping effects. A muscle model developed by clinical observation is shown in Figure 6.37.

A wide range of approaches can be used to model and animate the face. Which to use depends greatly on how realistic the result is meant to be and what kind of control the animator is provided. Results vary from cartoon faces to parameterized surface models to skull-muscle-skin simulations. Realistic facial animation remains one of the interesting challenges in computer animation.

## 6.4 Overview of Virtual Human Representation

One of the most difficult challenges in computer graphics is the creation of virtual humans. Efforts to establish standard representations of articulated bodies are now emerging [17] [36]. Representing human figures successfully requires solutions to several very different problems. The visible geometry of the skin can be created by a variety of techniques, differing primarily in the skin detail and the degree of representation of underlying internal structures such as bone and muscle. Geometry for hair and clothing can be simulated with a clear trade-off between accuracy and computational complexity. The way in which light interacts with skin, clothing, and hair can also be calculated with varying degrees of correctness, depending on visual requirements and available resources.

Techniques for simulating virtual humans have been created that allow for extremely modest visual results but that can be computed in real time (i.e., 60 Hz). Other approaches may give extremely realistic results by simulating individual hairs, muscles, wrinkles, or threads. These methods may consequently also require days of computation time to generate a single frame of animation. This discussion focuses solely on the representation of virtual humans. It touches only on animation issues where necessary to discuss how the animation techniques affect the figure's creation.

## 6.4.1 Representing Body Geometry

Many methods have been developed for creating and representing the geometry of a virtual human's body. They vary primarily in visual quality and computational complexity. Usually these two measures are inversely proportionate.

The vast majority of human figures are modeled using a boundary representation constructed from either polygons (often triangles) or patches (usually NURBS). These boundary shell models are usually modeled manually in one of the common off-the-shelf modeling packages (e.g., [2] [5] [39]). The purpose of the model being produced dictates the technique used to create it. If the figure is constructed for real-time display in a game on a low-end PC or gaming console, usually it will be assembled from a relatively low number of triangular polygons, which, while giving a chunky appearance to the model, can be rendered quickly. If the figure will be used in an animation that will be rendered off-line by a high-end rendering package, it might be modeled with NURBS patch data, to obtain smooth curved contours. Factors such as viewing distance and the importance of the figure to the scene can be used to select from various levels of detail at which to model the figure for a particular sequence of frames.

### Polygonal Representations

Polygonal models usually consist of a set of vertices and a set of faces. Polygonal human figures can be constructed out of multiple objects (frequently referred to as segments), or they can consist of a single polygonal mesh. When multiple objects are used, they are generally arranged in a hierarchy of joints and rigid segments. Rotating a joint rotates all of that joint's children (e.g., rotating a hip joint rotates all of the child's leg segments and joints around the hip). If a single mesh is used, then rotating a joint must deform the vertices surrounding that joint, as well as rotate the vertices in the affected limb.

Various constraints may be placed on the polygonal mesh's topology depending on the software that will be displaying the human. Many real-time rendering engines require polygonal figures to be constructed from triangles. Some modeling programs require that the object remain closed.

Polygonal representations are primarily used either when rendering speed is of the essence, as is the case in real-time systems such as games, or when topological flexibility is required. The primary problem with using polygons as a modeling primitive is that it takes far too many of them to represent a smoothly curving surface. It might require hundreds or thousands of polygons to achieve the same visual quality as could be obtained with a single NURBS patch.

### Patch Representations

Virtual humans constructed with an emphasis on visual quality are frequently built from a network of cubic patches, usually NURBS. The control points defining these patches are manipulated to sculpt the surfaces of the figure. Smooth continuity must be maintained at the edges of the patches, which often proves challenging. Complex topologies also can cause difficulties, given the rectangular nature of the patches. While patches can easily provide much smoother surfaces than polygons in general, it is more challenging to add localized detail to a figure without adding a great deal more global data. Hierarchical splines provide a partial solution to this problem [27].

### Other Representations

Several other methods have been used for representing virtual human figures. However, they are used more infrequently because of a lack of modeling tools or because of their computational complexity.

Subdivision surfaces combine the topological flexibility of polygonal objects with the resultant smoothness of patch data. They transform a low-resolution polygonal model into a smooth form by recursively subdividing the polygons as necessary [19] [23].

Implicit surfaces (frequently the term "metaballs" is used) can be employed as sculpting material for building virtual humans. Metaballs resemble clay in their ability to blend with other nearby primitives. While computationally expensive, they provide an excellent organic look that is perfect for representing skin stretched over underlying tissue [39] [43].

Probably the most computationally demanding representation method is volumetric modeling. While all of the above techniques merely store information about the surface qualities of a virtual human, volumetric models store information about the entire interior space as well. Because of its extreme computational requirements, this technique is limited almost exclusively to the medical research domain, where knowledge of the interior of a virtual human is crucial.

As computers continue to become more powerful, more attempts are being made to more accurately model the interior of humans to get more realistic results on the visible surfaces. There have been several "layered" approaches, where some attempt has been made to model underlying bone and/or muscle and its effect on the skin.

Chen and Zeltzer [15] use a finite element approach to accurately model a human knee, representing each underlying muscle precisely, based on medical data. Several authors have attempted to create visually reasonable muscles attached to bones and then generate skin over the top of the muscle (e.g., [62] [73]). Thalmann's lab takes the interesting hybrid approach of modeling muscles with metaballs, producing cross sections of these metaballs along the body's segments, and then lofting polygons between the cross sections to produce the final surface geometry [10]. Chadwick, Haumann, and Parent [14] use FFDs to produce artist-driven muscle bulging and skin folding.

### 6.4.2 Geometry Data Acquisition

Geometric data can be acquired by a number of means. By far the most common method is to have an artist create the figure using interactive software tools. The quality of the data obtained by these means is of course completely dependent on the artist's skills and experience. Another method of obtaining data, digitizing real humans, is becoming more prevalent as the necessary hardware becomes more affordable. Data representing a specific individual are often captured using a laser scanner or by performing image processing on video images [34] [38] [45].

There have also been various parametric approaches to human figure data generation. Virtual human software intended for use in ergonomics simulation tends to use parameters with a strong anthropometric basis [6]. Software with an artistic or entertainment orientation allows for more free-form parametric control of the generated body data [39] [64]. A few projects are just starting to scratch the surface of using exemplar-based models to allow for data generation by mixing known attributes [9] [45].

### 6.4.3 Geometry Deformation

For a user to animate a virtual human figure, the figure's subparts must be able to be deformed. The method of deformation used is largely determined by the way the figure is being represented. Very simple figures, usually used for real-time display, are often broken into multiple rigid subparts, such as a forearm, a thigh, or a head. These parts are arranged in a hierarchy of joints and segments such that rotating a joint rotates all of the child segments and joints beneath it in the hierarchy [36]. While this method is quick, it yields terrible visual results at the joints, particularly if the body is textured.

A single skin is more commonly used for polygonal figures. When a joint is rotated, the appropriate vertices are deformed to simulate rotation around the joint. Several different methods can be used for this, and, as with most of these techniques, they involve trade-offs of realism and speed. The simplest and fastest method is to bind each vertex to exactly one bone. When a bone rotates, the vertices move along with it [7]. Better results can be obtained, at the cost of additional computation, if the vertices surrounding a joint are weighted so that their position is affected by multiple bones [43]. While weighting the effects of bone rotations on vertices results in smoother skin around joints, severe problems can still occur with extreme joint bends. Free-form deformations have been used in this case to simulate the skin-on-skin collision and the accompanying squashing and bulging that occur when a joint such as the elbow is fully bent [14]. The precise placement of joints within the body greatly affects the realism of the surrounding deformations. Joints must be placed strictly according to anthropometric data, or unrealistic bulges will result [33] [58].

Some joints require more complicated methods for realistic deformation. Using only a simple, single three-degrees-of-freedom rotation for a shoulder or vertebrae can yield very poor results. A few systems have attempted to construct more complex, anatomically based joints [24] [53]. The hands in particular can require significantly more advanced deformation techniques to animate realistically [30]. Surface deformations that would be caused by changes in underlying material, such as muscle and fat in a real human, can be produced in a virtual human by a number of means. Methods range from those that simply allow an animator to specify muscular deformations [14] to those that require complex dynamics simulations of the various tissue layers and densities [62]. A great deal of muscle simulation research has been conducted for facial animation. See the facial animation section of this chapter for more details. Finally, deformations resulting from interaction with the environment have been simulated both with traditional dynamics systems and with implicit surfaces [65].

### 6.4.4 Clothing

It is the rare application or animation that calls for totally nude figures. Simulating clothing and its interaction with the surfaces of the figure is probably the most computationally intensive part of representing virtual humans. As a result, virtual humans to date usually appear to be sporting rigid geometric clothing or tight-fitting spandex implemented as texture maps. In some real-time applications, rigid segments of cloth have been jointed together and animated via key frames [64]. High-end off-line animation systems are starting to offer advanced cloth simulation modules that attempt to calculate the effects of gravity as well as cloth-cloth and cloth-body collisions by using mass-spring networks [2] [8] [52] [61]. See Section 6.6 for more discussion of cloth and clothing.

### 6.4.5 Hair

The most significant hurdle for making virtual humans that are indistinguishable from real ones is the accurate simulation of a full head of hair. Convincingly emulating the complex interactions of thousands of hairs has proved exceedingly difficult. The most common, visually poor, but computationally inexpensive, technique has been to merely construct a rigid piece of geometry in the rough shape of the volume of hair and attach it to the top of the head, like a helmet. Texture maps with transparency information are sometimes used to improve this approach. The next best option is similar to one of the simpler cloth techniques: animate a chain of rigid hair segments. This technique is often seen with animated ponytails [64]. While real-time virtual humans usually employ one of the previous two techniques for simulating hair, off-line hair modeling is beginning to employ either small geometric tubes [4] [48] or particle trails [2] to generate individual strands. Correctly lighting individual hair strands efficiently remains an active research problem.

### 6.4.6 Surface Detail

After the geometry for a virtual figure has been constructed, its surface properties must also be specified. As with the figure's geometry, surface properties can be produced by an artist, scanned from real life, or procedurally generated. Not only color but also specular, diffuse, bump, and displacement maps may be generated. Accurately positioning the resulting textures requires the generation of texture coordinates [61]. The skin may be simulated using complex physically based simulations [31] [46]. Wrinkles may also be simulated by different means [48] [65].

## 6.5 Layered Approach to Human Figure Modeling

A common approach to animating the human figure is to construct the figure in layers consisting of skeleton, muscles, and skin. The skeletal layer is responsible for the articulation of the form. The muscle layer is responsible for deforming the shape as a result of skeletal articulation. The skin is responsible for carrying the appearance of the figure.

Chadwick, Haumann, and Parent [14] introduce the layered approach to figure animation by incorporating an articulated skeleton, surface geometry representing the skin, and an intermediate muscle layer that ties the two together. The muscle layer is not anatomically based, and its only function is to deform the surface geometry as a function of joint articulation. The muscle layer implements a system of free-form deformation lattices in which the surface geometry is embedded. The lattice is organized with two planes on each end that are responsible for maintaining continuity with adjoining lattices, and the interior planes are responsible for deforming the skin to simulate muscle bulging (Figure 6.38). As the joint flexes, the interior planes elongate perpendicular to the axis of the link. The elongation is usually not symmetrical about the axis and is designed by the animator. For example, the upper-arm FFD lattice elongates as the elbow flexes. Typically, the FFD

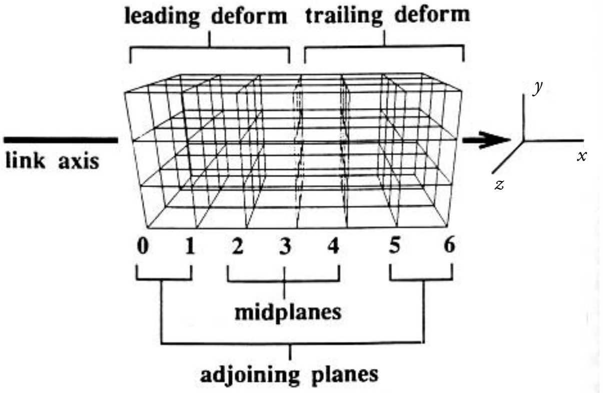


**Figure 6.38** The basic FFD lattice [14]

deformation for the upper arm is designed to produce the majority of the skin deformation in the region of the biceps. A pair of FFD lattices are used on either side of each joint to isolate the FFDs responsible for muscle bulging from the more rigid area around the joint. In addition, the joint FFDs are designed to maintain continuity on the outside of the joint and create the skin crease on the inside of the joint. See Figure 6.39

Recently, there has been more anatomically accurate modeling of the human figure [62] [63] [73]. Scheepers [62] and Scheepers et al. [63] use artistic anatomy to guide analysis of the human form. Bones, muscles, tendons, and fatty tissue are modeled in order to occupy the appropriate volumes. Scheepers identifies the types of muscles sufficient for modeling the upper torso of the human figure: linear muscles, sheet muscles, and bendable sheet muscles. Tendons are modeled as part of the muscles and attach to the skeleton. Muscles deform according to the articulation of the skeleton. See Figure 6.40 (Plate 9) for an example. These muscles populate the skeletal figure in the same manner that actual muscles are arranged in the human body (Figure 6.41 [Plate 10]). To deform the skin according to the underlying structure (muscles, tendons, fatty tissue), the user defines implicit functions so that the densities occupy the volume of the corresponding anatomy. Ellipsoids are used for muscles, cylinders for tendons, and flattened ellipsoids for fat pads. The implicits are summed to smooth the surface and further model underlying tissue. The skin, modeled as a B-spline surface, is defined by floating the control points of the B-spline patches to the isosurface of the summed implicits. This allows the skin to distort as the underlying skeletal structure articulates and muscles deform (Figure 6.42).

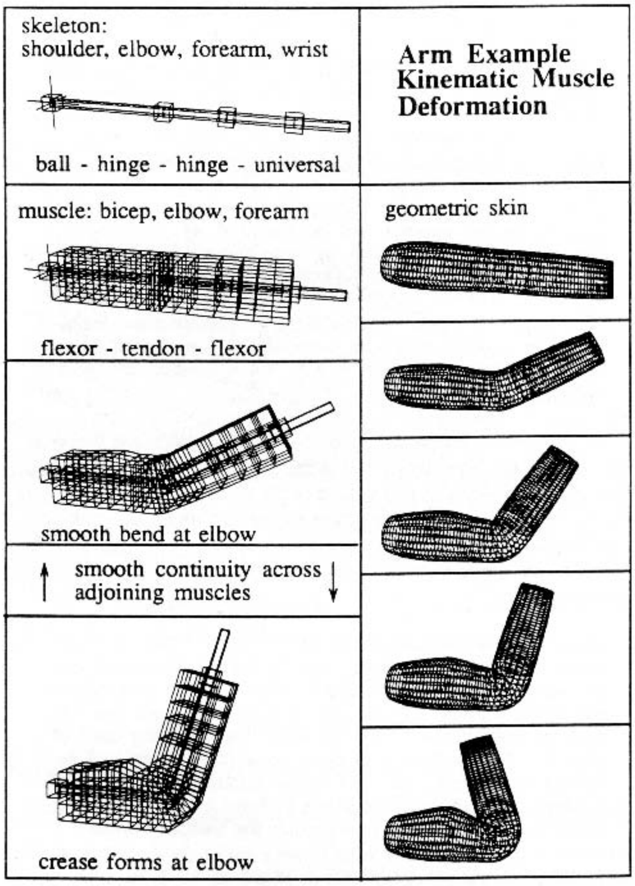


**Figure 6.39** Deformation induced by FFDs as a result of joint articulation [14]

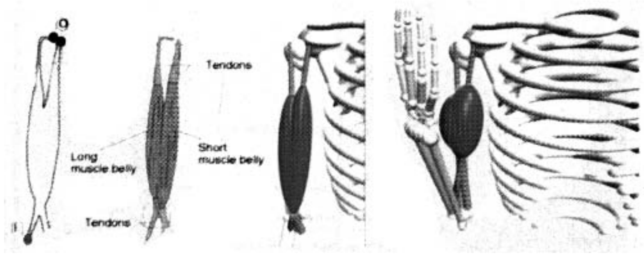


**Figure 6.40** Linear muscle model [62]

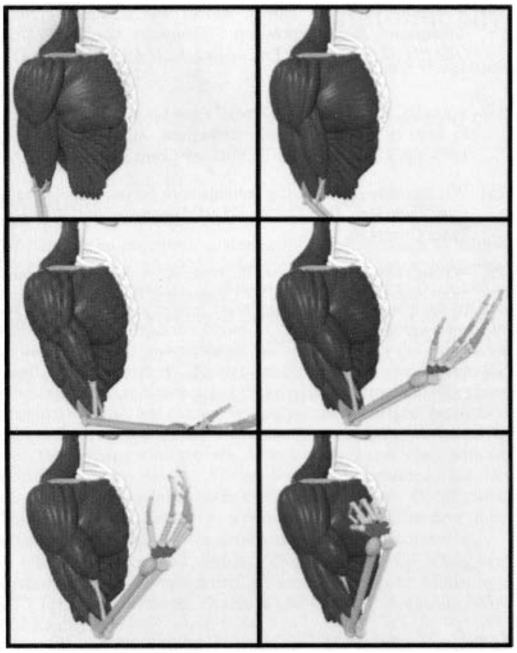


**Figure 6.41** Muscles of upper torso [62]

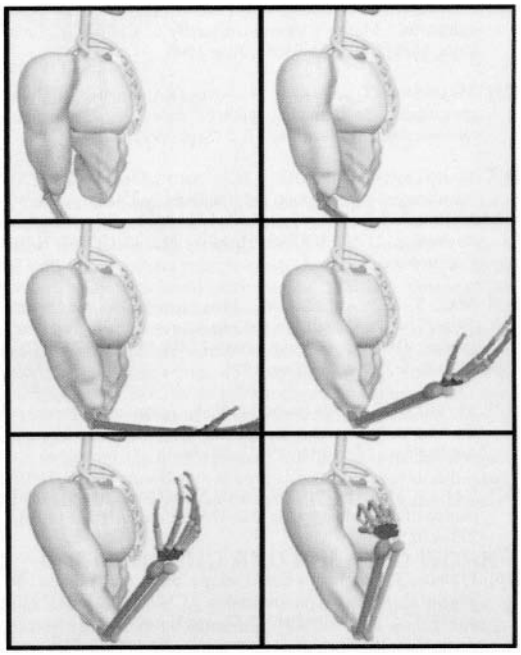


**Figure 6.42** Skin model over muscles, tendons, and fatty tissue [62]

## 6.6 Cloth and Clothing

The clothes that protect and decorate the body contribute importantly to the appearance of a human figure. For most human figures in most situations, cloth covers the majority of the body. As in real life, cloth provides important visual qualities for the synthetic figure and imparts certain attributes and characteristics to it. The way in which cloth drapes over and highlights or hides aspects of the figure can make the figure more or less attractive. For an animated figure, the clothes provide important visual cues that indicate the type and speed of its motion. For example, the swirling of a skirt or the bouncing of a shirttail indicates the pace or smoothness of a walk.

One of the simplest cases of modeling the behavior of cloth is the static draping of a piece of material. Very specific ridges in the material will be created when supported at discrete points. Animating these points, in turn, animates the cloth. This model for cloth is limited in that it does not address the folding, wrinkling, and bending of material. A robust model for cloth must consider the constant state of collision between the material and the figure being clothed. The more robust implementations of cloth typically use a mesh of triangles as the underlying model, in which collisions are tested for continually. The simple draping model does, however, help to illustrate the difference between levels of representation possible when employing a model for animation. Simple draping identifies the macro-features of cloth and models these directly, whereas triangulated polygonal meshes model cloth at a much finer level of detail. With realistic modeling of these fine-detail elements, the macro-features of the cloth will emerge from the simulation without explicitly being modeled. In the discussion that follows, simple draping of cloth is discussed first, followed by the considerations involved in forming a more robust model for clothing on a figure.

### 6.6.1 Simple Draping

In the special case of a cloth supported at a fixed number of constrained points, the cloth will drape along well-defined lines. Weil [72] presents a geometric method for hanging cloth from a fixed number of points. The cloth is represented as a two-dimensional grid of points located in three-space with certain of the grid points constrained to a fixed position.

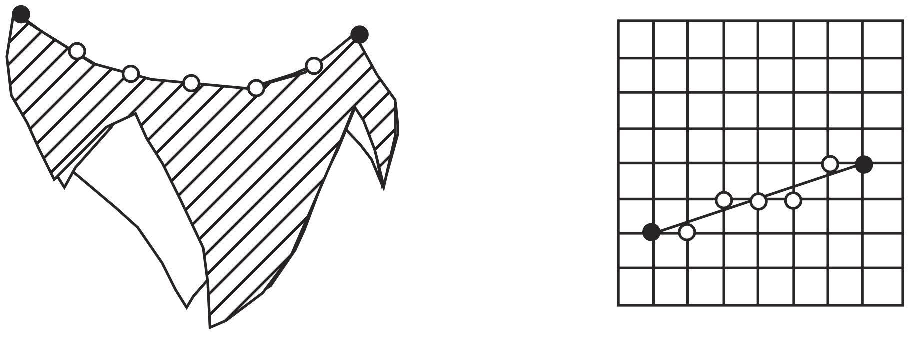
The procedure takes place in two stages. In the first stage, an approximation is made to the draped surface within the convex hull of the constrained points. The second stage is a relaxation process that continues until the maximum displacement during a given pass falls below a user-defined tolerance.

Vertices on the cloth grid are classified as either *interior* or *exterior,* depending on whether they are inside or outside the convex hull of the constrained points. This inside/outside determination is based on two-dimensional grid coordinates. The first step is to position the vertices that lie along the line between constrained vertices. The curve of a thread hanging between two vertices is called a *catenary curve* and has the form shown in Equation 6.2.

$$y = c - \left(a \cdot \cosh\left(\frac{x-b}{a}\right)\right) \tag{Eq. 6.2}$$

Catenary curves traced between paired constrained points are generated using vertices of the cloth that lie along the line between constrained vertices. The vertices between the constrained vertices are identified in the same way that a line is drawn between points on a raster display (Figure 6.43).

If two catenary curves cross each other in grid space (Figure 6.44), then the lower of the two curves is simply removed. The reason for this is that the catenary curves essentially support the vertices along the curve. If a vertex is supported by



Cloth supported at two constrained points

Constrained points in grid space

**Figure 6.43** Constrained cloth and grid coordinate space

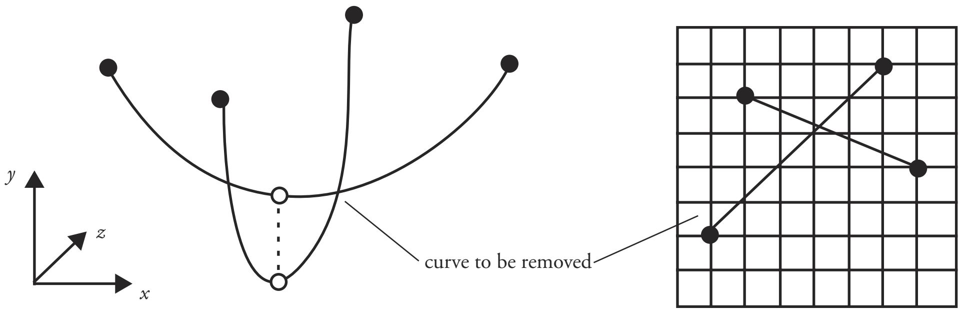


**Figure 6.44** Two catenary curves supporting the same point

two curves, but one curve supports it at a higher position, then the higher curve takes precedence.

A catenary curve is traced between each pair of constrained vertices. After the lower of two crossing catenary curves is removed, a triangulation is produced in the grid space of the constrained vertices (Figure 6.45). The vertices of the grid that fall on the lines of triangulation are positioned in three-space according to the catenary equations. To find the catenary equation between two vertices ($x_1$, $y_1$), ($x_2$, $y_2$), see Equation 6.3 [72]. Each of these triangles is repeatedly subdivided by constructing a catenary from one of the vertices to the midpoint of the opposite edge on the triangle. This is done for all three vertices of the triangle. The highest of the three catenaries so formed is kept and the others are discarded. This breaks the triangle into two new triangles (Figure 6.46). This continues until all interior vertices have been positioned.

$$y_1 = c + a \cdot \cosh\left[\frac{(x_1 - b)}{a}\right]$$

$$y_2 = c + a \cdot \cosh\left[\frac{(x_2 - b)}{a}\right]$$

$$L = a \cdot \sinh\left[\frac{(x_2 - b)}{a}\right] - a \cdot \sinh\left[\frac{(x_1 - b)}{a}\right]$$

$$\sqrt{L^2 - (y_2 - y_1)^2} = 2 \cdot a \cdot \sinh\left[\frac{(x_2 - x_1)}{2 \cdot a}\right] \quad \text{(}a\text{ can be solved for numerically at this point)}$$

$$M = \sinh\left(\frac{x_2}{a}\right) - \sinh\left(\frac{x_1}{a}\right)$$

$$N = \cosh\left(\frac{x_2}{a}\right) - \cosh\left(\frac{x_1}{a}\right)$$

$$\text{if } N > M \qquad \mu = \tanh^{-1}\left(\frac{M}{N}\right)$$

$$Q = \frac{M}{\sinh(\mu)} = \frac{N}{\cosh(\mu)}$$

$$b = a \cdot \left[\mu - \sinh^{-1}\left(\frac{L}{Q \cdot a}\right)\right]$$

$$\text{if } M > N \qquad \mu = \tanh^{-1}\left(\frac{N}{M}\right)$$

$$Q = \frac{N}{\sinh(\mu)} = \frac{M}{\cosh(\mu)}$$

$$b = a \cdot \left[\mu - \cosh^{-1}\left(\frac{L}{Q \cdot a}\right)\right] \qquad \textbf{(Eq. 6.3)}$$

A relaxation process is used as the second and final step in positioning the vertices. The effect of gravity is ignored; it has been used implicitly in the formation of the catenary curves using the interior vertices. The exterior vertices are initially positioned so as to effect a downward pull. The relaxation procedure repositions each vertex to satisfy unit distance from each of its neighbors. For a given vertex, displacement vectors are formed to each of its neighbors. The vectors are added to determine the direction of the repositioning. The magnitude of the repositioning is determined by taking the square root of the squares of the distance to each of the

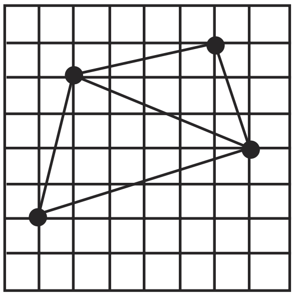


**Figure 6.45** Triangulation of constrained points in grid coordinates

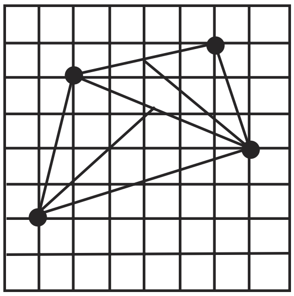


**Figure 6.46** Subdividing triangles

neighbors. Self-intersection of the cloth is ignored. To model material stiffness, the user can add dihedral angle springs to bias the shape of the material.

Animation of the cloth is produced by animating the existence and/or position of constrained points. In this case, the relaxation process can be made more efficient by updating the position of points based on positions in the last time step. Modeling cloth this way, however, is not appropriate for clothes because of the large number of closely packed contact points that clothing material has with the body. The formation of catenary curves is not an important aspect of clothing, and there is no provision for the wrinkling of material above contact points. For clothing, a more physically based approach is needed.

## 6.6.2 Getting into Clothes

To simulate clothing, the user must incorporate the dynamic aspect of cloth into the model to produce the wrinkling and bulging that naturally occur when cloth is worn by an articulated figure. In order for the figure to affect the shape of the cloth, extensive collision detection and response must be calculated as the cloth collides with the figure and with itself almost constantly. The level of detail at which clothes must be modeled to create realistic wrinkles and bulges requires relatively small triangles. Therefore, it takes a large number of geometric elements to clothe a human figure. As a result, one must attend to the efficiency of the methods used to implement the dynamic simulation and collision handling of the clothing.

### Modeling Dynamics

To achieve the realistic reaction of the cloth to the movements of the supporting figure, the user must incorporate the characteristics of the cloth threads into the model (e.g., [69]); these characteristics must be adjustable to model differences among types of cloth. The characteristics of cloth are its ability to stretch, bend, and skew. These tendencies can be modeled as springs that impart forces or as energy functions that contribute to the overall energy of a given configuration. Forces are used to induce accelerations on system masses. The energy functions can be minimized to find optimal configurations or differentiated to find local gradients. Whether forces or energies are used, the functional forms are similar and are usually based on the amount some metrics deviate from rest values.

Woven cloth is formed by the warp (parallel threads in the lengthwise principal direction of the material) and weft (parallel filler threads in the direction orthogonal to the warp threads) pattern of the threads, which creates a quadrilateral mesh. Thus, cloth is suitably modeled by a quadrilateral mesh in which the vertices of any given quadrilateral element are not necessarily planar.

Stretching is usually modeled by simply measuring the amount that the length of an edge deviates from its rest length. Equation 6.4 shows a commonly used form for the force equation of the corresponding spring. $v1^*$ and $v2^*$ are the rest positions of the vertices defining the edge; $v1$ and $v2$ are the current positions of the edge vertices. Notice that the force is a vector equation, whereas the energy equation is a scalar. The analogous energy function is given in Equation 6.5. The metrics have been normalized by the rest length ($|v1^* - v2^*|$).

$$F_s = k_s \cdot \left(\frac{|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right) \cdot \frac{v1 - v2}{|v1 - v2|} \qquad \text{(Eq. 6.4)}$$

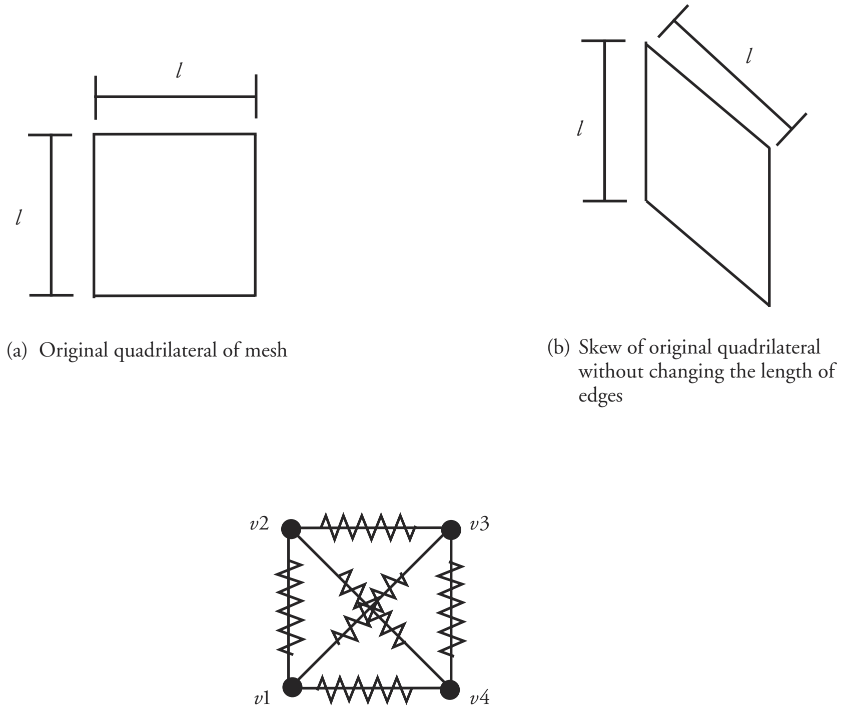
$$E_s = k_s \cdot \frac{1}{2} \cdot \left(\frac{|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right)^2 \qquad \text{(Eq. 6.5)}$$

Restricting the stretching of edges only controls changes to the surface area of the mesh. Skew is in-plane distortion of the mesh, which still maintains the length of the original edges (Figure 6.47a, b). To control such distortion (when using forces), one may employ diagonal springs (Figure 6.47c). The energy function suggested by DeRose, Kass, and Truong [19] to control skew distortion is given in Equation 6.6.

$$S(v1,v2) = \left(\frac{1}{2}\right) \cdot \left(\frac{|v1 - v2| - |v1^* - v2^*|}{|v1^* - v2^*|}\right)^2$$

$$E_w = k_w \cdot S(v1,v3) \cdot S(v2,v4) \qquad \text{(Eq. 6.6)}$$

Edge and diagonal springs (energy functions) control in-plane distortions, but out-of-plane distortions are still possible. These include the bending and folding of the mesh along an edge that does not change the length of the edges or diagonal



(a) Original quadrilateral of mesh

(b) Skew of original quadrilateral without changing the length of edges

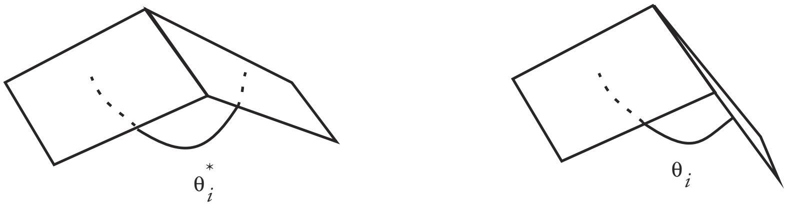
(c) Diagonal springs to control skew

**Figure 6.47** Original quadrilateral, skewed quadrilateral, and controlling skew with diagonal springs

measurements. Bending can be controlled by either restricting the dihedral angles (the angle between adjacent quadrilaterals) or controlling the separation among a number of adjacent vertices. A spring-type equation based on deviation from the rest angle can be used to control the dihedral angle: $F_b = k_b \cdot (\theta_i - \theta_i^*)$ (Figure 6.48). Bending can also be controlled by considering the separation of adjacent vertices in the direction of the warp and weft of the material (Figure 6.49). See Equation 6.7 for an example.

$$F_b = k_b \cdot \left( \frac{l3}{(l1 + l2)} - \frac{l3^*}{(l1^* + l2^*)} \right) \qquad \textbf{(Eq. 6.7)}$$

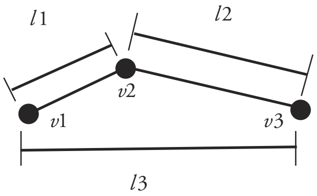
Whether springlike forces or energy functions are used, the constants $k_s$, $k_w$, and $k_b$ can be globally and locally manipulated to impart characteristics to the mesh.



Original dihedral angle

Bending along the edge that changes dihedral angle

**Figure 6.48** Control of bending by dihedral angle



$l1 = |v2 - v1|$

$l2 = |v3 - v2|$

$l3 = |v3 - v1|$

**Figure 6.49** Control of bending by separation of adjacent vertices

## Collision Detection and Response

Collisions with cloth are handled much like collisions with elements of any complex environment. That is, they must be handled efficiently. One of the most common techniques for handling collisions in a complex environment is to organize data in a hierarchy of bounding boxes. As suggested by DeRose, Kass, and Truong [19], such a hierarchy can be built from the bottom up by first forming bounding boxes for each of the faces of the cloth. Each face is then logically merged with an adjacent face, forming the combined bounding box and establishing a node one level up in the hierarchy. This can be done repeatedly to form a complete hierarchy of bounding boxes for the cloth faces. Care should be taken to balance the resulting hierarchy, as by making sure all faces are used when forming nodes at each level of the hierarchy. Because the cloth is not a rigid structure, additional information can be included in the data structure to facilitate updating the bounding boxes each time a vertex is moved. Hierarchies can be formed for rigid and nonrigid structures in the environment. To test a vertex for possible collision with geometric elements, compare it with all the object bounding box hierarchies, including the hierarchy of the object to which it belongs.

Collisions are an almost constant occurrence for a figure wearing clothes. Calculation of impulse forces that result from such collisions is costly for an animated figure. Whenever a vertex is identified that violates a face in the environment, a transient spring can be introduced that imparts a force to the vertex so that it is restored to an acceptable position.

## 6.7 Motion Capture

*Motion capture* of an object involves sensing, digitizing, and recording that object in motion [49]. As used here, it specifically refers to capturing human motion (or the motion of other life forms), but some very creative animation can be accomplished by capturing the motion of structures other than biological life forms. In motion capture, the subject's movement is recorded. It should be noted that locating and extracting a figure's motion directly from raw (unadulterated) video is extremely difficult and is the subject of current research. As a result, the subject of the motion capture is typically instrumented in some way so that positions of key feature points can be easily detected and recorded. Usually, these positions are in two-space. The $x$-, $y$-, $z$-coordinates of joints and other strategic positions of the figure are computed from the instrumented positions. An appropriate synthetic figure can then be fitted to these positions and animated as the positions of the figure are made to follow the motion of the computed positions.

There are primarily two approaches to this instrumentation:[2] *electromagnetic sensors* and *optical markers*. Electromagnetic tracking, also simply called *magnetic tracking*, uses sensors placed at the joints that transmit their positions and orientations back to a central processor to record their movements. While accurate theoretically, magnetic tracking systems require an environment devoid of magnetic field distortions. To transmit their information, the sensors have to use either cables or wireless transmission to communicate with the central processor. The former requires that the subject be "tethered" with some kind of cabling harness. The latter requires money. The advantage of electromagnetic sensors is that the three-dimensional position and orientation of each sensor can be recorded and displayed in real time (with some latency). The drawbacks relate to the range and accuracy of the magnetic field. Optical markers, on the other hand, have much larger range, and the performers only have to wear reflective markers on their clothes (see Figure 6.50). The optical approach does not provide real-time feedback, however, and the data from optical systems is error-prone and noisy. Optical markers use video technology to record images of the subject in motion. Because orientation information is not directly generated, more markers are required than with magnetic trackers. Some combination of joint and midsegment markers is

[2] There are several other technologies used to capture motion, including electromechanical suits, fiber-optic sensors, and digital armatures [49]. However, electromagnetic sensors and (passive) optical markers are by far the most commonly used technologies for capturing full-body motion.

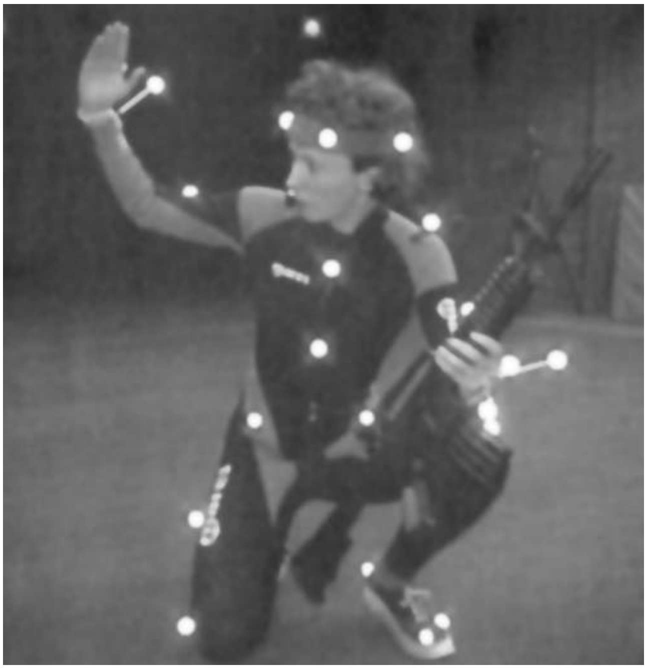


**Figure 6.50** Image from optical motion capture session [20]

used. While industrial strength systems may use infrared cameras, eight or more cameras, and fast (up to 120 frames per second) cameras, basic optical motion control can be implemented with consumer-grade video technology. Because the optical approach can be low cost, at least in limited situations, this is the approach that is discussed here.

The objective of motion control is to reconstruct the three-dimensional motion of a physical object and apply it to a synthetic model. With optical systems, three major tasks need to be undertaken. First, the images need to be processed so that the animator can locate, identify, and correlate the markers in the 2D video images. Second, the 3D locations of the markers need to be reconstructed from their 2D locations. Third, the 3D marker locations must be constrained to a model of the physical system whose motion is being captured (e.g., a stick figure model of the performer). The first requires some basic image-processing techniques, simple logic, and sometimes some luck. The second requires camera calibration and enough care to overcome numerical inaccuracies. The third requires satisfying constraints between relative marker positions.

### 6.7.1 Processing the Images

Optical markers can be fashioned from Ping-Pong balls and coated to make them stand out in the video imagery. They can be attached to the figure using Velcro strips or some other suitable method. Colored tape can also be used. The markers are usually positioned at the joints since these are the structures of interest in animating a figure. The difference between the position of the marker and that of the joint is one source of error in motion capture systems. This is further complicated if the marker moves relative to the joint during the motion of the figure. Once the video is digitized, it is simply a matter of scanning each video image for evidence of the optical markers. If the background image is static, then it can be subtracted out to simplify the processing. Finding the position of a single marker in a video frame in which the marker is visible is the easy part. This step gets messier the more markers there are, the more often some of the markers are occluded, the more often the markers overlap in an image, and the more the markers change position relative to one another. With multimarker systems, the task is not only to isolate the occurrence of a marker in a video frame but also to track a specific marker over a number of frames even when the marker may not always be visible.

Once all of the visible markers have been extracted from the video, each individual marker must be tracked over the video sequence. Sometimes this can be done with simple domain knowledge. For example, if the motion is constrained to be normal walking, then the ankle markers (or foot markers, if present) can be assumed to always be the lowest markers and they can be assumed to always be within some small distance from the bottom of the image. Frame-to-frame coherence can be employed to track markers by making use of the position of a marker in a previous frame and knowing something about the limits of its velocity and acceleration. For example, knowing that the markers are on a walking figure and knowing something about the camera position relative to the figure, one can estimate the maximum number of pixels that a marker might travel from one frame to the next and thus help track it over time.

Unfortunately, one of the realities of optical motion capture systems is that periodically one or more of the markers are occluded. In situations in which several markers are used, this can cause problems in successfully tracking a marker throughout the sequence. Some simple heuristics can be used to track markers that drop from view for a few frames and that do not change their velocity much over that time. But these heuristics are not foolproof (this is why they are called heuristics). The result of failed heuristics can be markers swapping paths in mid-sequence or the introduction of a new marker when, in fact, a marker is simply reappearing again. Marker swapping can happen even when markers are not occluded. If markers pass within a small distance of each other they may swap paths because of numerical inaccuracies of the system. Sometimes these problems can be resolved when the three-dimensional positions of markers are constructed. At other times user intervention is required to resolve ambiguous cases.

As a side note, there are optical systems that use active markers. The markers are LEDs that flash their own unique identification code. There is no chance of marker swapping in this case, but this system has its own limitations. The LEDs are not very bright and cannot be used in bright environments. Because each marker has to take the time to flash its own ID, the system captures the motion at a relatively slow rate. Finally, there is a certain delay between the measurements of markers, so the positions of each marker are not recorded at exactly the same moment, which may present problems in animating the synthetic model.

A constant problem with motion capture systems is noise. Noise can arise from the physical system; the markers can move relative to their initial positioning, and the faster the performer moves, the more the markers can swing and reposition themselves. Noise also arises from the sampling process; the markers are being sampled in time and space, and errors can be introduced in all dimensions. A typical error might result in inaccurate positioning of a feature point by half a centimeter. For some animations, this can be a significant error.

To deal with the noise, the user can condition the data before they are used in the reconstruction process. Data points that are radically inconsistent with typical values can be thrown out, and the rest can be filtered. The objective is to smooth the data without removing any useful features. A simple weighted average of adjacent values can be used to smooth the curve. The number of adjacent values to use and their weights are a function of the desired smoothness. Generally, this must be selected by the user.

### 6.7.2 Camera Calibration

Before the three-dimensional position of a marker can be reconstructed, it is necessary to know the locations and orientations of cameras in world space as well as the intrinsic properties of the cameras such as focal length, image center, and aspect ratio [68].

A simple pinhole camera model is used for the calibration. This is an idealized model that does not accurately represent certain optical effects often present in real cameras, but it is usually sufficient for computer graphics and image-processing applications. The pinhole model defines the basic projective geometry used to describe the imaging of a point in three-space. For example, the camera's coordinate system is defined with the origin at the center of projection and the plane of projection at a focal-length distance along the positive $z$-axis, which is pointed toward the camera's center of interest (Figure 6.51). Equivalently, the projection plane could be a focal length along the negative $z$-axis on the other side of the cen-

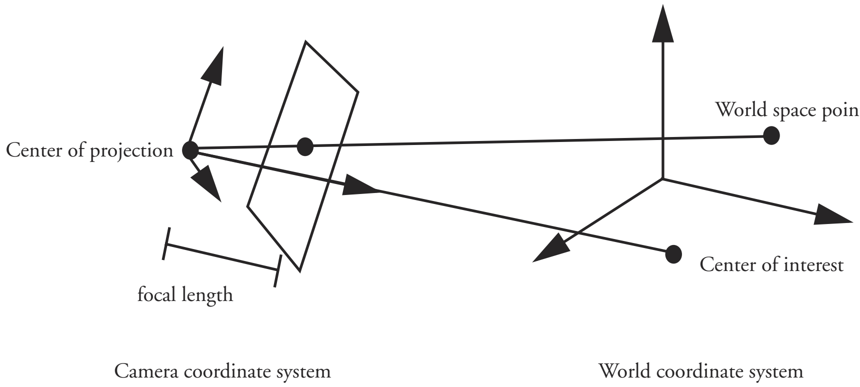


**Figure 6.51** Camera model

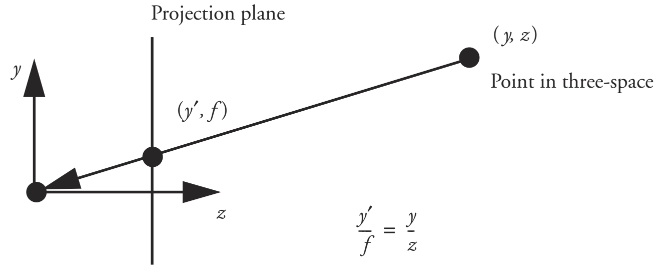


**Figure 6.52** *Y-Z* projection of a world space point onto the image plane in the camera coordinate system

ter of projection from the center of interest; this would produce an inverted image, but the mathematics would be the same.

The image of a point is formed by projecting a ray from the point to the center of projection (Figure 6.52). The image of the point is formed on the image (projection) plane where this ray intersects the plane. The equations for this point, as should be familiar to the reader, are formed by similar triangles. Camera calibration is performed by recording a number of image space points whose world space locations are known. These pairs can be used to create an overdetermined set of linear equations that can be solved using a least-squares solution method. See Appendix B for further discussion.

### 6.7.3 3D Position Reconstruction

To reconstruct the three-dimensional coordinates of a marker, the user must locate its position in at least two views relative to known camera positions. In the simplest case, this requires two cameras to be set up to record marker positions (Figure 6.53). The greater the orthogonality of the two views, the better the chance for an accurate reconstruction.

If the position and orientation of each camera are known with respect to the global coordinate system, along with the position of the image plane relative to the camera, then the images of the point to be reconstructed ($I_1$, $I_2$) can be used to locate the point, $P$, in three-space (Figure 6.53). Using the location of a camera, the relative location of the image plane, and a given pixel location on the image plane, the user can compute the position of that pixel in world coordinates. Once that is known, a vector from the camera through the pixel can be constructed in world space for each camera (Equation 6.8, Equation 6.9).

$$C_1 + k_1 \cdot (I_1 - C_1) = P \tag{Eq. 6.8}$$

$$C_2 + k_2 \cdot (I_2 - C_2) = P \tag{Eq. 6.9}$$

By setting these equations equal to each other, $C_1 + k_1 \cdot (I_1 - C_1) = C_2 + k_2 \cdot (I_2 - C_2)$. This represents three equations with two unknowns that can be easily solved—in an ideal world. Noise tends to complicate the ideal world. In practice, these two equations will not exactly intersect, although if the noise in the system is not excessive, they will come close. So, in practice, the points of closest encounter must be found on each line. This requires that a $P_1$ and a $P_2$ be found such that $P_1$

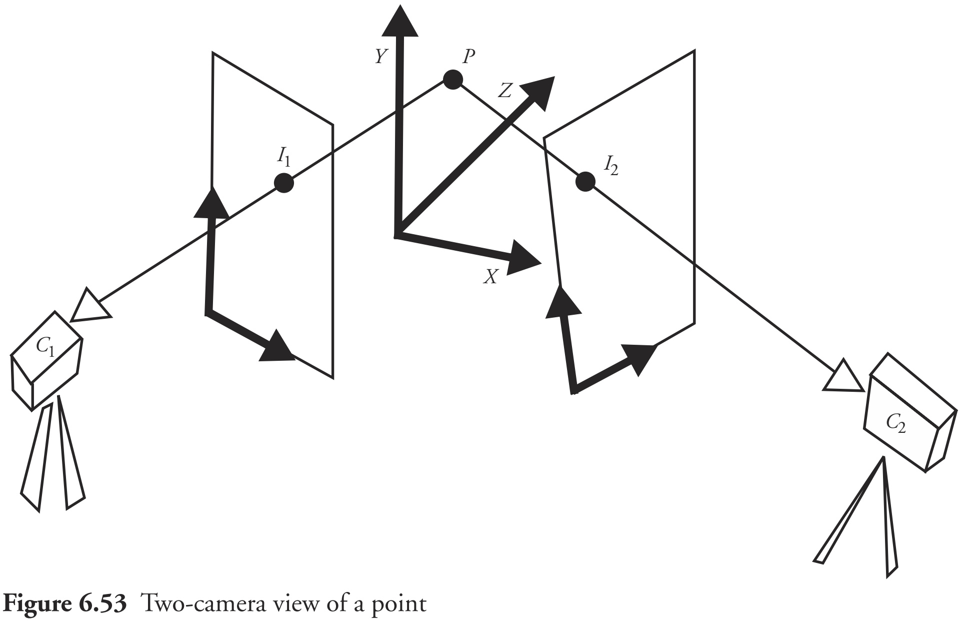


**Figure 6.53** Two-camera view of a point

is on the line from Camera 1, $P_2$ is on the line from Camera 2, and $P_2 - P_1$ is perpendicular to each of the two lines (Equation 6.10, Equation 6.11).

$$(P_2 - P_1) \bullet (I_1 - C_1) = 0 \tag{Eq. 6.10}$$

$$(P_2 - P_1) \bullet (I_2 - C_2) = 0 \tag{Eq. 6.11}$$

See Appendix B on solving for $P_1$ and $P_2$. Once the points $P_1$ and $P_2$ have been calculated, the midpoint of the chord between the two points can be used as the location of the reconstructed point. In the case of multiple markers in which marker identification and tracking have not been fully established for all the markers in all the images, the distance between $P_1$ and $P_2$ can be used as a test for correlation between $I_1$ and $I_2$. If the distance between $P_1$ and $P_2$ is too great, then this indicates that $I_1$ and $I_2$ are probably not images of the same marker and a different pairing needs to be tried. Smoothing can also be performed on the reconstructed three-dimensional paths of the markers to further reduce the effects of noise on the system.

### Multiple Markers

The number and positioning of markers on a human figure depend on the intended use of the captured motion. A simple configuration of markers for digitizing gross human figure motion might require only fourteen markers (Figure 6.54). For more accurate recordings of human motion, markers need to be added to the elbows, knees, chest, hands, toes, ankles, and spine. Figure 6.55 shows thirty-one unique markers. Menache [49] suggests an addition of three per foot for some applications.

### Multiple Cameras

As the number of markers increases and the complexity of the motion becomes more involved, there is greater chance for marker occlusion. To reconstruct the

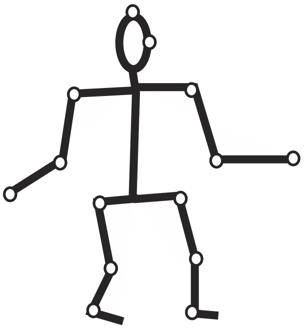


**Figure 6.54** Sample marker sets

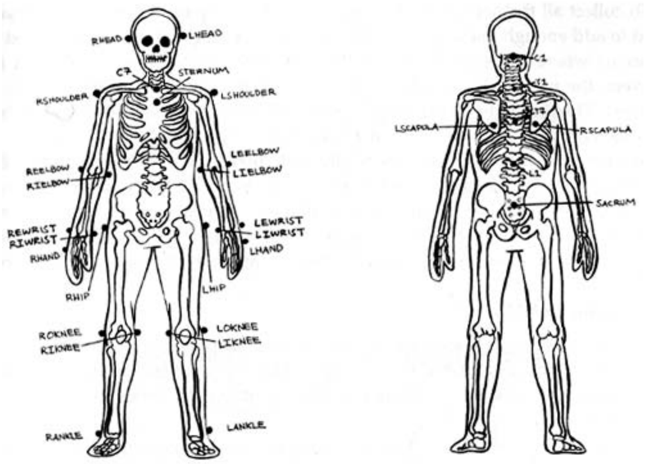


**Figure 6.55** Complete marker set [49]

three-dimensional position of a marker, the user must see and identify it in at least two images. As a result, a typical system may have eight cameras simultaneously taking images. These sequences need to be synchronized either automatically or manually. This can be done manually, for example, by noting the frame in each sequence when a particular heel strike occurs. However, with manually synchronized cameras the images could be a half a frame off from each other.

### 6.7.4 Fitting to the Skeleton

Once the motion of the individual markers looks smooth and reasonable, the next step is to attach them to the underlying skeletal structure that is to be controlled by the digitized motion. In a straightforward approach, the position of each marker in each frame is used to absolutely position a specific joint of the skeleton. As a first approximation to controlling the motion of the skeleton, this works fine. However, on closer inspection, a problem often exists. The problem with using the markers to directly specify position is that, because of noise, smoothing, and general inaccuracies, distances between joints of the skeleton will not be precisely maintained over time. This change in bone length can be significant. Length changes of 10 to 20 percent are common. In many cases, this is not acceptable. For example, this can cause *foot-sliding* of the skeleton (also known as *skating*). Inverse kinematics used to lock the foot to the ground can counteract this skating.

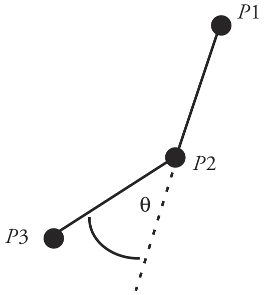
One source of the problem is that the markers are located, not at the joints of the performers, but outside the joints at the surface. This means that the point being digitized is displaced from the joint itself. While a constant distance is maintained on the performer, for example, between the knee joint and the hip joint, it is not the case that a constant distance is maintained between a point to the side of the knee joint and a point to the side of the hip joint.

The one obvious correction that can be made is logically relocating the digitized positions so that they correspond more accurately to the positions of the joints. This can be done by using marker information to calculate the joint position. The distance from the marker to the actual joint can be determined easily by visual inspection. However, the problem with locating the joint relative to a marker is that there is no orientation information directly associated with the marker. This means that, given a marker location and the relative distance from the marker to the joint, the user does not know in which direction to apply the displacement in order to locate the joint.

One solution is to put markers on both sides of the joint. With two marker locations, the joint can be interpolated as the midpoint of the chord between the two markers. While effective for joints that lend themselves to this approach, the approach does not work for joints that are complex or more inaccessible (such as the shoulder and spine), and it doubles the number of markers that must be processed.

A little geometry can be used to calculate the displacement of the joint from the marker. A plane formed by three markers can be used to calculate a normal to a plane of three consecutive joints, and this normal can be used to displace the joint location. Consider the elbow. If there are two markers at the wrist (commonly used to digitize the forearm rotation) the position of the true wrist can be interpolated between them. Then the wrist-elbow-shoulder markers can be used to calculate a normal to the plane formed by those markers. Then the true elbow position is calculated by offsetting from the elbow marker in the direction of the normal by the amount measured from the performer. By recalculating the normal every frame, the user can easily maintain an accurate elbow position throughout the performance. In most cases, this technique is very effective. A problem with the technique is that when the arm straightens out the wrist-elbow-shoulder become (nearly) colinear. Usually, the normal can be interpolated during these periods of congruity from accurately computed normals on either side of the interval. This approach keeps limb lengths much closer to being constant.

Now that the digitized joint positions are more consistent with the skeleton to be articulated, they can be used to control the skeleton. To avoid absolute positioning in space and further limb-length changes, one typically uses the digitized positions to calculate joint rotations. For example, in a skeletal hierarchy, if the positions of three consecutive joints have been recorded for a specific frame, then the third of the points in the hierarchy is used to calculate the rotation of that limb relative to the limb represented by the first two points (Figure 6.56).



$$\cos(\theta) = \frac{(P3 - P2) \bullet (P2 - P1)}{|P3 - P2| \cdot |P2 - P1|}$$

**Figure 6.56** One-degree-of-freedom rotation joint

After posing the model using the calculated joint angles, it might still be the case that, because of inaccuracies in the digitization process, feature points of the model violate certain constraints such as avoiding floor penetration. The potential for problems is particularly high for end effectors, such as the hands or feet, which must be precisely positioned. Often, these must be independently positioned, and then the joints higher up the hierarchy (e.g., knee and hip) must be adjusted to compensate for any change to the end effector position.

## 6.7.5 Modifying Motion Capture

There has been recent work (e.g., [13] [29] [74]) on modifying motion capture data to fit individuals of various dimensions or to blend one motion into another. This holds the promise of developing libraries of common digitized motions that can be retrieved, modified, adapted, and combined to produce any desired motion for any target figure. However, this remains an area of active research. Most often the motion is recaptured if the original motion does not fit with its intended purpose.

## 6.7.6 Summary

Motion capture is a very powerful and useful tool. It will never replace the results produced by a skilled animator, but its role in animation will expand and increase as motion libraries are built and the techniques to modify, combine, and adapt motion capture data become more sophisticated. Current research involves extracting motion capture data from markerless video. This has the potential to free the subject from instrumentation constraints and make motion capture even more useful.

## 6.8 Chapter Summary

The human figure is an interesting and complex form. It has a uniform structure but contains infinite variety. As an articulated rigid structure, it contains many degrees of freedom, but its surface is deformable. The pliable nature of the face presents an immense animation challenge by itself. Modeling and animating hair in any detail is also enormously complex. Moreover, the constant collision and sliding of cloth on the surface of the body represents significant computational complexity.

One of the things that make human motion so challenging is that no single motion can be identified as correct human motion. Human motion varies from individual to individual for a number of reasons, but it is still recognizable as reasonable human motion, and slight variations can seem odd or awkward. Research in computer animation is just starting to delve into the nuances of human motion. Many of these nuances vary from person to person but, at the same time, are very consistent for a particular person, for a particular ethnic group, for a particular age group, for a particular weight group, for a particular emotional state, and so on. Computer animation is only beginning to analyze, record, and synthesize these important qualities of movement. Most of the work has focused on modeling changes in motion resulting from an individual's emotional state (e.g., [3] [16]).

Human figure animation remains a challenge for computer animators and fertile ground for graphics researchers. Developing a synthetic actor indistinguishable from a human counterpart remains the Holy Grail of researchers in computer animation.

## References

1. T. Akimoto and Y. Suenaga, "3D Facial Model Creation Using Generic Model and Front and Side View of Face," *IEICE Transactions on Information and Systems,* E75-D(2):191–197, March 1992.
2. Alias/Wavefront. Maya. 1998. http://www.aliaswavefront.com.
3. K. Amaya, A. Bruderlin, and T. Calvert, "Emotion from Motion," *Graphics Interface '96,* pp. 222–229 (May 1996). Canadian Human-Computer Communications Society. Edited by Wayne A. Davis and Richard Bartels. ISBN 0-9695338-5-3.
4. K. Anjyo, Y. Usami, and T. Kurihara, "A Simple Method for Extracting the Natural Beauty of Hair," *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 111–120 (August 1988, Atlanta, Ga.). Edited by John Dill.
5. Avid Technology, Inc., SOFTIMAGE|3D, http://www.softimage.com/.

6. F. Azulola, N. Badler, Teo K. Hoon, and Susanna Wei, "Sass v.2.1 Anthropometric Spreadsheet and Database for the Iris," Technical Report, Dept. of Computer and Information Science, University of Pennsylvania, 1993. MS-CIS-93-63.
7. K. Baca, "Poor Man's Skinning," *Game Developer,* pp. 48–51 (July 1998).
8. D. Baraff and A. Witkin, "Large Steps in Cloth Simulation," *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 43–54 (August 1988, Atlanta, Ga.). Edited by John Dill.
9. V. Blanz and T. Vetter, "A Morphable Model for the Synthesis of 3D Faces," Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, pp. 187–194 (August 1999, Los Angeles, Calif.). Addison-Wesley Longman. Edited by Alyn Rockwood. ISBN 0-20148-560-5.
10. R. Boulic, T. Capin, Z. Huang, P. Kalra, B. Linterrnann, N. Magnenat-Thalmann, L. Moccozet, T. Molet, L. Pandzic, K. Saar, A. Schmitt, J. Shen, and D. Thalmann, "The HUMANOID Environment for Interactive Animation of Multiple Deformable Human Characters," *Computer Graphics Forum,* 14 (3), pp. 337–348 (August 1995). Blackwell Publishers. Edited by Frits Post and Martin Göbel. ISSN 1067-7055.
11. A. Bruderlin and T. Calvert, "Goal-Directed, Dynamic Animation of Human Walking," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 233–242 (July 1989, Boston, Mass.). Edited by Jeffrey Lane.
12. A. Bruderlin and T. Calvert, "Knowledge-Driven, Interactive Animation of Human Running," *Graphics Interface '96,* pp. 213–221 (May 1996). Canadian Human-Computer Communications Society. Edited by Wayne A. Davis and Richard Bartels. ISBN 0-9695338-5-3.
13. A. Bruderlin and L. Williams, "Motion Signal Processing," Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 97–104 (August 1995, Los Angeles, Calif.). Addison-Wesley. Edited by Robert Cook. ISBN 0-201-84776-0.
14. J. Chadwick, D. Haumann, and R. Parent, "Layered Construction for Deformable Animated Characters," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 243–252 (July 1989, Boston, Mass.). Edited by Jeffrey Lane.
15. D. Chen and D. Zeltzer, "Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle Using the Finite Element Method," *Computer Graphics* (Proceedings of SIGGRAPH 92), 26 (2), pp. 89–98 (July 1992, Chicago, Ill.). Edited by Edwin E. Catmull. ISBN 0-201-51585-7.
16. D. Chi, M. Costa, L. Zhao, and N. Badler, "The EMOTE Model for Effort and Shape," Proceedings of SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, pp. 173–182 (July 2000). ACM Press/ACM SIGGRAPH/Addison-Wesley Longman. Edited by Kurt Akeley. ISBN 1-58113-208-5.
17. COVEN, http://coven.lancs.ac.uk/mpeg4/index.html, January 2001.
18. D. DeCarlo, D. Metaxas, and M. Stone, "An Anthropometric Face Model Using Variational Techniques," Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 67–74 (July 1998, Orlando, Fla.). Addison-Wesley. Edited by Michael Cohen. ISBN 0-89791-999-8.
19. T. DeRose, M. Kass, and T. Truong, "Subdivision Surfaces for Character Animation," Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 85–94 (July 1998, Orlando, Fla.). Addison-Wesley. Edited by Michael Cohen. ISBN 0-89791-999-8.

20. Digital Domain, *Digital Domain Demo Tape,* 1999.
21. D. Ebert, J. Ebert, and K. Boyer, "Getting into Art" (animation), CIS Department, Ohio State University, May 1990.
22. P. Ekman and W. Friesen, *Facial Action Coding System,* Consulting Psychologists Press, Palo Alto, Calif., 1978.
23. M. Elson, "Displacement Animation: Development and Application," Course #10: Character Animation by Computer, SIGGRAPH 1990 (August 1990, Dallas, Tex.).
24. A. Engin and R. Peindl, "On the Biomechanics of Human Shoulder Complex—I: Kinematics for Determination of the Shoulder Complex Sinus," *Journal of Biomechanics,* 20 (2), pp. 103–117, 1987.
25. FAP Specifications, http://www-dsp.com.dist.unige.it/~pok/RESEARCH/MPEGfapspec.htm, January 2001.
26. FDP Specifications, http://www-dsp.com.dist.unige.it/~pok/RESEARCH/MPEGfdpspec.htm, January 2001.
27. D. Forsey and R. Bartels, "Hierarchical B-Spline Refinement," *Computer Graphics* (Proceedings of SIGGRAPH 88), 22 (4), pp. 205–212 (August 1988, Atlanta, Ga.). Edited by John Dill.
28. M. Girard and A. Maciejewski, "Computational Modeling for the Computer Animation of Legged Figures," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 263–270 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
29. M. Gleicher, "Retargeting Motion to New Characters," Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, pp. 33–42 (July 1998, Orlando, Fla.). Addison-Wesley. Edited by Michael Cohen. ISBN 0-89791-999-8.
30. J.-P. Gourret, N. Magnenat-Thalmann, and D. Thalmann. "Simulation of Object and Human Skin Deformations in a Grasping Task," *Computer Graphics* (Proceedings of SIGGRAPH 89), 23 (3), pp. 21–30 (July 1989, Boston, Mass.). Edited by Jeffrey Lane.
31. P. Hanrahan and K. Wolfgang, "Reflection from Layered Surfaces due to Subsurface Scattering," Proceedings of SIGGRAPH 93, Computer Graphics Proceedings, Annual Conference Series, pp. 165–174 (August 1993, Anaheim, Calif.). Edited by James T. Kajiya. ISBN 0-201-58889-7.
32. G. Harris and P. Smith, eds., *Human Motion Analysis,* IEEE Press, Piscataway, N.J., 1996.
33. S. Henry-Biskup, "Anatomically Correct Character Modeling," Gamasutra, 2 (45), November 13, 1998, http://www.gamasutra.com/features/visual_arts/19981113/charmod_01.htm.
34. A. Hilton, D. Beresford, T. Gentils, R. Smith, and W. Sun, "Virtual People: Capturing Human Models to Populate Virtual Worlds," http://www.ee.surrey.ac.uk/Research/VSSP/3DVision/VirtualPeople/.
35. J. Hodgins, W. Wooten, D. Brogan, and J. O'Brien, "Animating Human Athletics," Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 71–78 (August 1995, Los Angeles, Calif.). Addison-Wesley. Edited by Robert Cook. ISBN 0-201-84776-0.
36. Humanoid Animation Working Group, "Specification for a Standard Humanoid," http://ece.uwaterloo.ca/~h-anim/spec1.1/, 1999.
37. V. Inman, H. Ralston, and F. Todd, *Human Walking,* Williams & Wilkins, Baltimore, Md., 1981.

38. I. Kakadiaris and D. Metaxas, "3D Human Body Model Acquisition from Multiple Views," Proceedings of the Fifth International Conference on Computer Vision, Boston, Mass., June 20–23, 1995.
39. Kinetix, 3D Studio MAX, http://www.ktx.com.
40. Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe, "Planning Motions with Intentions," Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, pp. 395–408 (July 1994, Orlando, Fla.). ACM Press. Edited by Andrew Glassner. ISBN 0-89791-667-0.
41. K. Kondo, *Inverse Kinematics of a Human Arm,* Technical Report STAN-CS-TR-94-1508, Stanford University, 1994.
42. F. Lacquaniti and J. F. Soechting, "Coordination of Arm and Wrist Motion during a Reaching Task," *Journal of Neuroscience,* 2 (2), pp. 399–408 (1982).
43. J. Lander, "On Creating Cool Real-Time 3D," Gamasutra, October 17, 1997, Vol. 1: Issue 8, http://www.gamasutra.com/features/visual_arts/101797/rt3d_01.htm.
44. P. Lee, S. Wei, J. Zhao, and N. Badler, "Strength Guided Motion," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 253–262 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
45. W.-S. Lee and N. Magnenat-Thalmann, "From Real Faces to Virtual Faces: Problems and Solutions," Proc. 3IA'98, Limoges (France), 1998.
46. J. P. Lewis, M. Cordner, and N. Fong, "Pose Space Deformations: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation," Proceedings of SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, pp. 165–172 (July 2000). ACM Press/ACM SIGGRAPH/Addison-Wesley Longman. Edited by Kurt Akeley. ISBN 1-58113-208-5.
47. N. Magnenat-Thalmann, S. Carion, M. Courchesne, P. Volino, and Y. Wu, "Virtual Clothes, Hair and Skin for Beautiful Top Models," *Computer Graphics International 1996* (1996). IEEE Computer Society.
48. M. McKenna and D. Zeltzer, "Dynamic Simulation of Autonomous Legged Locomotion," *Computer Graphics* (Proceedings of SIGGRAPH 90), 24 (4), pp. 29–38 (August 1990, Dallas, Tex.). Edited by Forest Baskett. ISBN 0-201-50933-4.
49. A. Menache, *Understanding Motion Capture for Computer Animation and Video Games,* Morgan Kaufmann, New York, 2000.
50. D. Miller, "The Generation of Human-Like Reaching Motion for an Arm in an Obstacle-Filled 3-D Static Environment," Ph.D. dissertation, Ohio State University, 1993.
51. S. Muraki, "Volumetric Shape Description of Range Data Using *Blobby Model*," *Computer Graphics* (Proceedings of SIGGRAPH 91), 25 (4), pp. 227–235 (July 1991, Las Vegas, Nev.). Edited by Thomas W. Sederberg. ISBN 0-201-56291-X.
52. L. Nedel and D. Thalmann, "Modeling and Deformation of the Human Body Using an Anatomically-Based Approach," *Computer Animation '98* (June 1998, Philadelphia, Pa.). IEEE Computer Society.
53. I. Pandzic, T. Capin, N. Magnenat-Thalmann, and D. Thalmann, "Developing Simulation Techniques for an Interactive Clothing System," *Proceedings of VSMM '97,* Geneva, Switzerland, 1997, pp. 109–118.
54. F. Parke, "Computer-Generated Animation of Faces," *Proceedings of the ACM Annual Conference* (August 1972).

55. F. Parke, "A Parametric Model for Human Faces," Ph.D. dissertation, University of Utah, 1974.
56. F. Parke and K. Waters, *Computer Facial Animation,* A. K. Peters, Wellesley, Mass., 1996. ISBN 1-56881-014-8.
57. M. Raibert and J. Hodgins, "Animation of Dynamic Legged Locomotion," *Computer Graphics* (Proceedings of SIGGRAPH 91), 25 (4), pp. 349–358 (July 1991, Las Vegas, Nev.). Edited by Thomas W. Sederberg. ISBN 0-201-56291-X.
58. REM Infogr·fica, "MetaReyes and ClothReyes," http://www.infografica.com/.
59. H. Rijpkema and M. Girard, "Computer Animation of Knowledge-Based Human Grasping," *Computer Graphics* (Proceedings of SIGGRAPH 91), 25 (4), pp. 339–348 (July 1991, Las Vegas, Nev.). Edited by Thomas W. Sederberg. ISBN 0-201-56291-X.
60. M. Rydfalk, "CANDIDE: A Parameterized Face," Technical Report LiTH-ISY-I-0866, Linköping University, Sweden, 1987.
61. G. Sannier and N. Magnenat-Thalmann, "A User-Friendly Texture-Fitting Methodology for Virtual Humans," *Computer Graphics International 1997* (June 1997, Hasselt/Diepenbeek, Belgium). IEEE Computer Society.
62. F. Scheepers, "Anatomy-Based Surface Generation for Articulated Models of Human Figures," Ph.D. dissertation, Ohio State University, 1996.
63. F. Scheepers, R. Parent, W. Carlson, and S. May, "Anatomy-Based Modeling of the Human Musculature," Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 163–172. Edited by Turner Whitted. ISBN 0-89791-896-7.
64. Sega, Virtua Fighter 3, http://www.sega.com/games/games_vf3.html.
65. K. Singh, "Realistic Human Figure Synthesis and Animation for VR Applications," Ph.D. dissertation, Ohio State University, 1995.
66. J. Soechting and M. Flanders, "Errors in Pointing Are due to Approximations in Sensorimotor Transformations," *Journal of Neurophysiology,* 62 (2), pp. 595–608 (August 1989).
67. N. Torkos and M. van de Panne, "Footprint-Based Quadruped Motion Synthesis," *Graphics Interface '98,* pp. 151–160 (June 1998). Edited by Kellogg Booth and Alain Fournier. ISBN 0-9695338-6-1.
68. M. Tuceryan, D. Greer, R. Whitaker, D. Breen, C. Crampton, E. Rose, and K. Ahlers, "Calibration Requirements and Procedures for a Monitor-Based Augmented Reality System," *IEEE Transactions on Visualization and Computer Graphics,* 1 (3), pp. 255–273 (September 1995). ISSN 1077-2626.
69. P. Volino, M. Courchesne, and N. Magnenat-Thalmann, "Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects," Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 137–144 (August 1995, Los Angeles, Calif.). Addison-Wesley. Edited by Robert Cook. ISBN 0-201-84776-0.
70. K. Waters, "A Physical Model of Facial Tissue and Muscle Articulation Derived from Computer Tomography Data," *SPIE Visualization in Biomedical Computing,* 1808, pp. 574–583, 1992.
71. K. Waters and D. Terzopoulos, "Modeling and Animating Faces Using Scanned Data," *Journal of Visualization and Computer Animation,* 2 (4), pp. 123–128, 1991.
72. J. Weil, "The Synthesis of Cloth Objects," *Computer Graphics* (Proceedings of SIGGRAPH 86), 20 (4), pp. 49–54 (August 1986, Dallas, Tex.). Edited by David C. Evans and Russell J. Athay.

73. J. Wilhelms and A. van Gelder, “Anatomically Based Modeling,” Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 173–180 (August 1997, Los Angeles, Calif.). Addison-Wesley. Edited by Turner Whitted. ISBN 0-89791-896-7.
74. A. Witkin and Z. Popovic, “Motion Warping,” Proceedings of SIGGRAPH 95, Computer Graphics Proceedings, Annual Conference Series, pp. 105–108 (August 1995, Los Angeles, Calif.). Addison-Wesley. Edited by Robert Cook. ISBN 0-201-84776-0.

APPENDIX A

# Rendering Issues

This appendix presents rendering techniques for computing a series of images that are played back as an animation sequence. It is assumed that the reader has a solid background in rendering techniques and issues, namely, the use of frame buffers, the *z*-buffer display algorithm, and aliasing. The techniques presented here concern smoothly displaying a sequence of images on a computer monitor (*double buffering*), efficiently computing images of an animated sequence (*compositing, drop shadows*), and effectively rendering moving objects (*motion blur*). An understanding of this material is not necessary for understanding the techniques and algorithms covered in the rest of the book, but computer animators should be familiar with these techniques when considering the trade-offs involved in rendering images for animation.

## A.1 Double Buffering

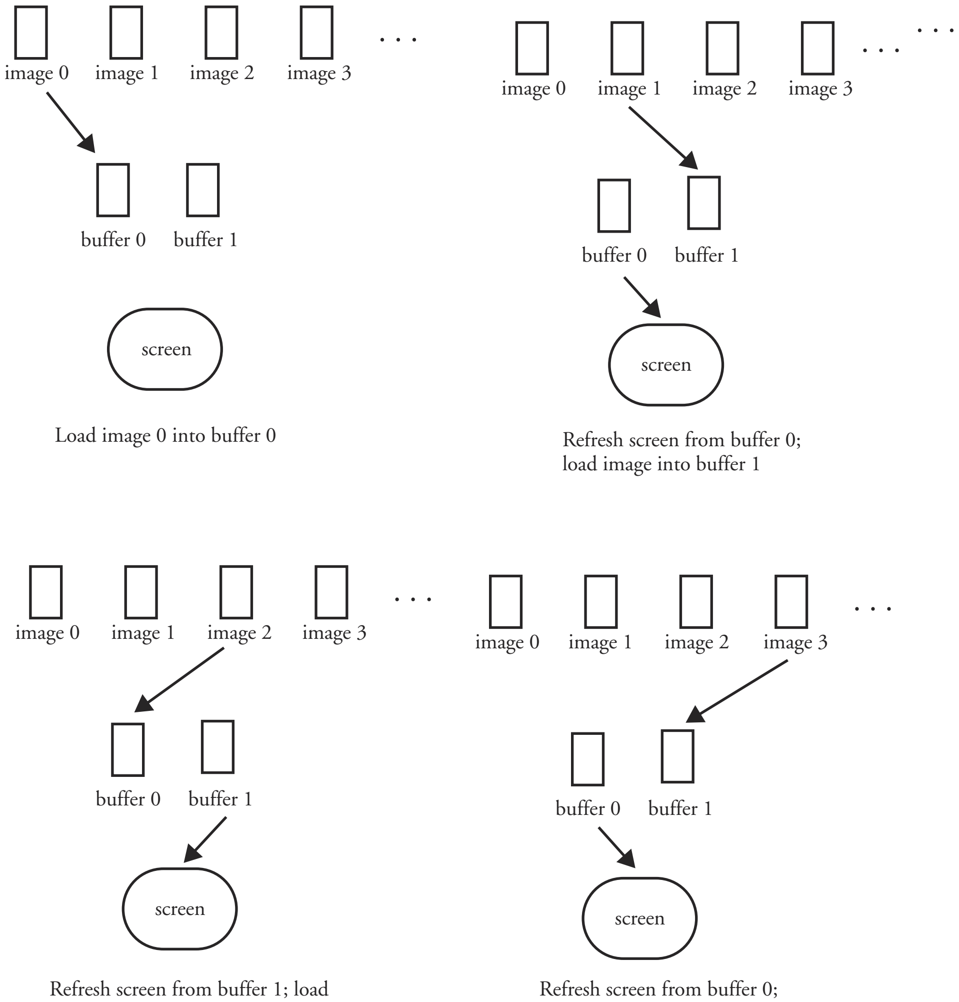
Not all computer animation is first recorded onto film or video for later viewing. In many cases, image sequences are displayed in real time on the computer monitor. Computer games are a prime example of this, as is web animation. Real-time display also occurs in simulators and for previewing animation for later high-quality recording. In some of these cases, the motion is computed and images are rendered as the display is updated; sometimes precalculated images are read from the disk and loaded into the frame buffer. In either case, the time it takes to paint an image on a computer screen can be significant (for various reasons). To avoid waiting for the image to update, animators often paint the new image into an off-screen buffer. Then a quick operation is performed to change the offscreen buffer to on-screen (often with hardware display support); the previous on-screen buffer becomes offscreen. This is called *double buffering*.

In double buffering, two (or more) buffers are used. One buffer is used to refresh the computer display, while another is used to assemble the next image. When the next image is complete, the two buffers switch roles. The second buffer is used to refresh the display, while the first buffer is used to assemble the next image. The buffers continue exchanging roles to ensure that the buffer used for display is not actively involved in the update operations (see Figure A.1).

Double buffering is often supported by the hardware of a computer graphics system. For example, the display system may have two built-in frame buffers, both of which are accessible to the user's program and are program selectable for driving the display. Alternatively, the graphics system may be designed so that the screen can be refreshed from an arbitrary section of main memory. Double buffering is also effective when implemented completely in software, as is popular in JAVA applets. Pseudocode of simple double buffering using two frame buffers is shown in Figure A.2.

## A.2 Compositing

Compositing is the act of combining separate image layers into a single picture. It allows the animator, or compositor, to combine elements obtained from different sources to form a single image. It allows artistic decisions to be deferred until all elements are brought together. Individual elements can be selectively manipulated without the user having to regenerate the scene as a whole. As frames of computer animation become more complex, the technique allows layers to be calculated separately and then composited after each has been finalized. Compositing provides great flexibility and many advantages to the animation process. Figure A.3 (Plate 11) shows a frame from *James and the Giant Peach* in which computer graphics, live action, and miniatures were combined. Before digital imagery, compositing was performed optically in much the same way that a multiplane camera is used in conventional animation. With digital technology, compositing operates on digital image representations.



Load image 0 into buffer 0

Refresh screen from buffer 0; load image into buffer 1

Refresh screen from buffer 1; load image into buffer 0

Refresh screen from buffer 0; load image into buffer 1

**Figure A.1** Double buffering

```
open_window (w)
i=0;                               // start using buffer 0
j=0;                               // start with image 0
do {
   load_image(buffer[i],image[j]); // load the jth image into the ith frame buffer
   display_in_window(w,buffer[i]); // display the ith buffer
   i=1-i;                          // swap which buffer to load image into
   j= j+1;                         // advance to the next image
} until (done);
```

**Figure A.2** Double buffering pseudocode

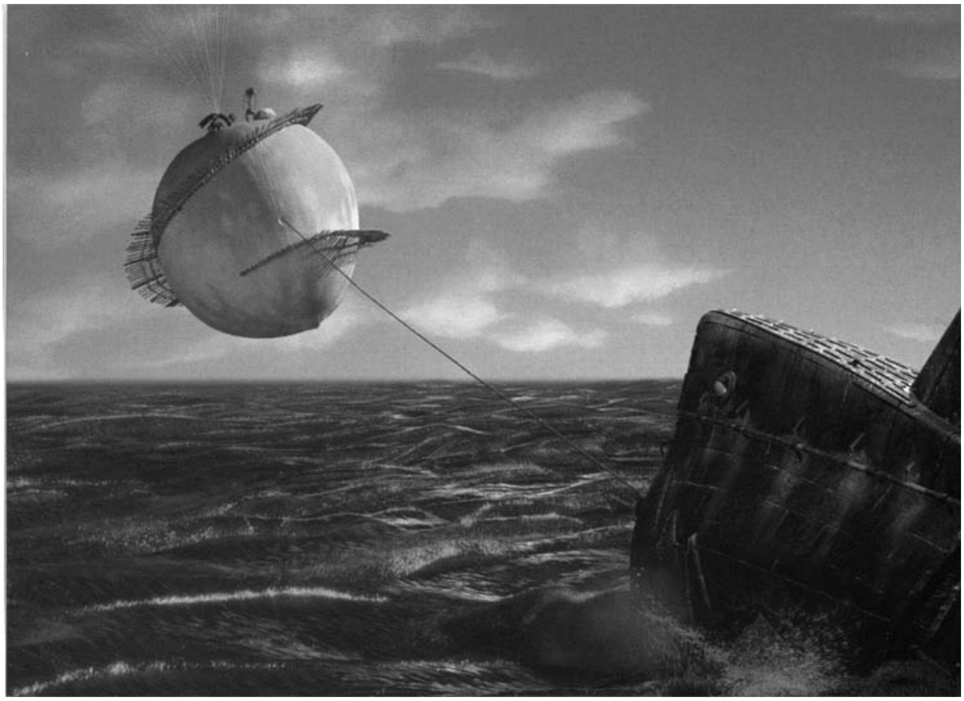


**Figure A.3** Composited image from *James and the Giant Peach*

One common and extremely important use of compositing is that of computing the foreground animation separately from the static background image. If the background remains static it need only be computed and rendered once. Then, as each frame of the foreground image is computed, it is composited with the background to form the final image. The foreground can be further segmented into multiple layers so that each object, or even each object part, has its own layer. This technique allows the user to modify just one of the object parts, rerender it, and then recompose it with the previously rendered layers, thus potentially saving a great deal of rendering time and expense. This is very similar to the 2½D approach taken in conventional hand-drawn cell animation.

Compositing offers a more interesting advantage when digital frame buffers are used to store depth information in the form of $z$-values, along with the color information at each pixel. To a certain extent, this approach allows the animator to composite layers that actually interleave each other in depth. The initial discussion focuses on compositing without pixel depth information, which effectively mimics the approach of the multiplane camera. Following this, compositing that takes the $z$-values into account is explored.

### A.2.1 Compositing without Pixel Depth Information

Digital compositing attempts to combine multiple two-dimensional images of three-dimensional scenes so as to approximate the visibility of the combined scenes. Compositing will combine two images at a time with the desire of maintaining the equilibrium shown in Equation A.1.

$$composite(render(scene1),\ render(scene2)) = render(merge(scene1,\ scene2)) \quad \textbf{(Eq. A.1)}$$

The *composite* operation on the left side of the equation refers to compositing the two rendered images; the *merge* operation on the right side of the equation refers to combining the geometries of the two scenes into one. The *render* function represents the process of rendering an image based on the input scene geometries. In the case in which the composite operator is used, the render function must tag pixels not covered by the scene as transparent. The equality will hold if the scenes are disjoint in depth from the observer and the composite operator gives precedence to the image closer to the observer. The visibility between elements from the different scenes can be accurately represented in 2½D. The visibility between disjoint planes can be accurately resolved by assigning a single visibility priority to all elements within a single plane.

The **over** operator places one image on top of the other. In order for **over** to operate, there must be some assumption or some additional information indicating which part of the closer image (also referred to as the *overlay plane* or *foreground image*) occludes the image behind it (the *background image*). In the simplest case, all of the foreground image occludes the background image. This is useful for the restricted situation in which the foreground image is smaller than the background image. In this case, the smaller foreground image is often referred to as a *sprite*. There are usually 2D coordinates associated with the sprite that locate it relative to the background image (see Figure A.4).

However, for most cases, additional information in the form of an *occlusion mask* (also referred to as a *matte* or *key*) is provided along with the overlay image. A one-bit matte can be used to indicate which pixels of the foreground image should occlude the background during the compositing process (see Figure A.5). In frame buffer displays, this technique is often used to overlay text or a cursor on top of an image.

Compositing is a binary operation, combining two images into a single image. However, any number of images can be composited to form a final image. The images must be ordered by depth and are operated on two at a time. Each operation replaces the two input images with the output image. Images can be composited in any order as long as the two composited during any one step are adjacent in their depth ordering.

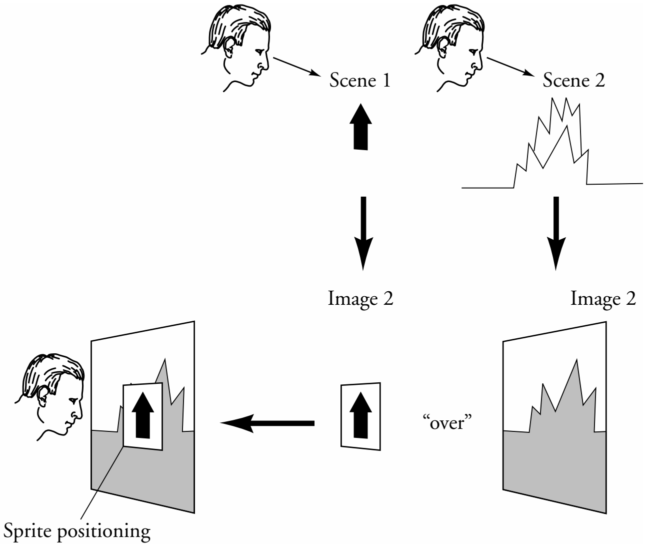
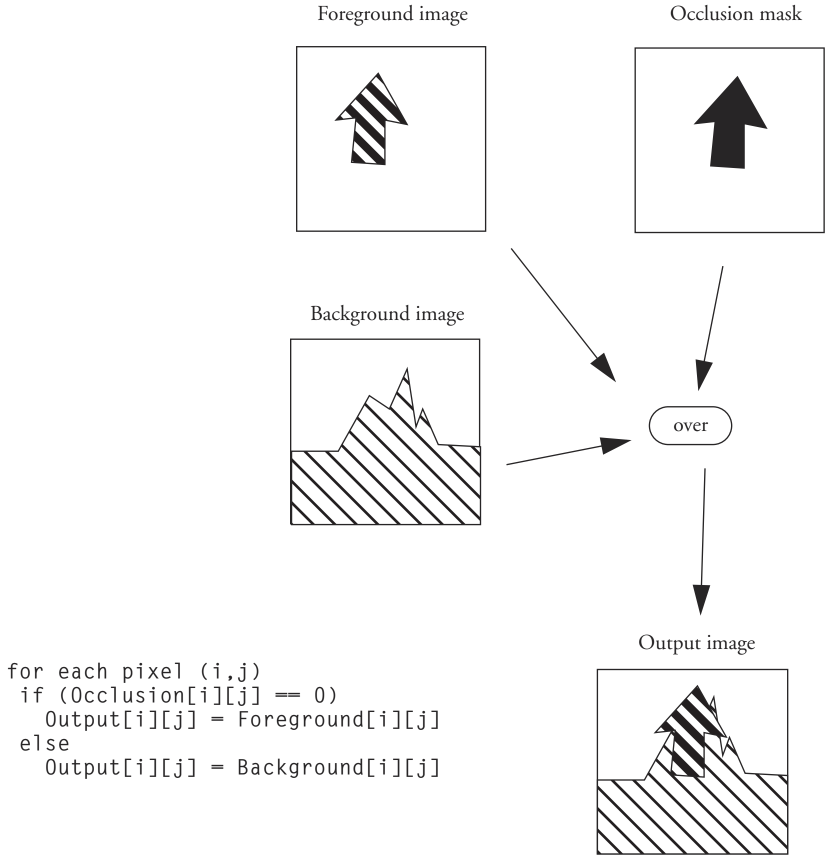


**Figure A.4** 2½D compositing without transparency

Compositing the images using a one-bit occlusion mask, that is, the color of a foreground image pixel in the output image, is an all-or-nothing decision. However, if the foreground image is calculated by considering semitransparent surfaces or partial pixel coverage, then fractional occlusion information is available and anti-aliasing during the pixel-merging process must be taken into account. Instead of using a one-bit mask for an all-or-nothing decision, using more bits allows a partial decision. This gray-scale mask is called *alpha* and is commonly maintained as a fourth *alpha channel* in addition to the three (red, green, and blue) color channels. The alpha channel is used to hold an opacity factor for each pixel. Even this is a shortcut; to be more accurate, an alpha channel for each of the three colors *R, G,* and *B* is required. A typical size for the alpha channel is eight bits.

The fractional occlusion information available in the alpha channel is an approximation used in lieu of detailed knowledge about the three-dimensional geometry of the two scenes to be combined. Ideally, in the process of determining the color of a pixel, polygons[1] from both scenes are made available to the renderer and visibility is resolved at the subpixel level. The combined scene is anti-aliased, and a color for each pixel is generated. However, it is often either necessary or more efficient to composite the images made from different scenes. Each scene is

[1] To simplify the discussion and diagrams, one must assume that the scene geometry is defined by a collection of polygons. However, any geometric element can be accommodated provided that coverage, occlusion, color, and opacity can be determined on a subpixel basis.

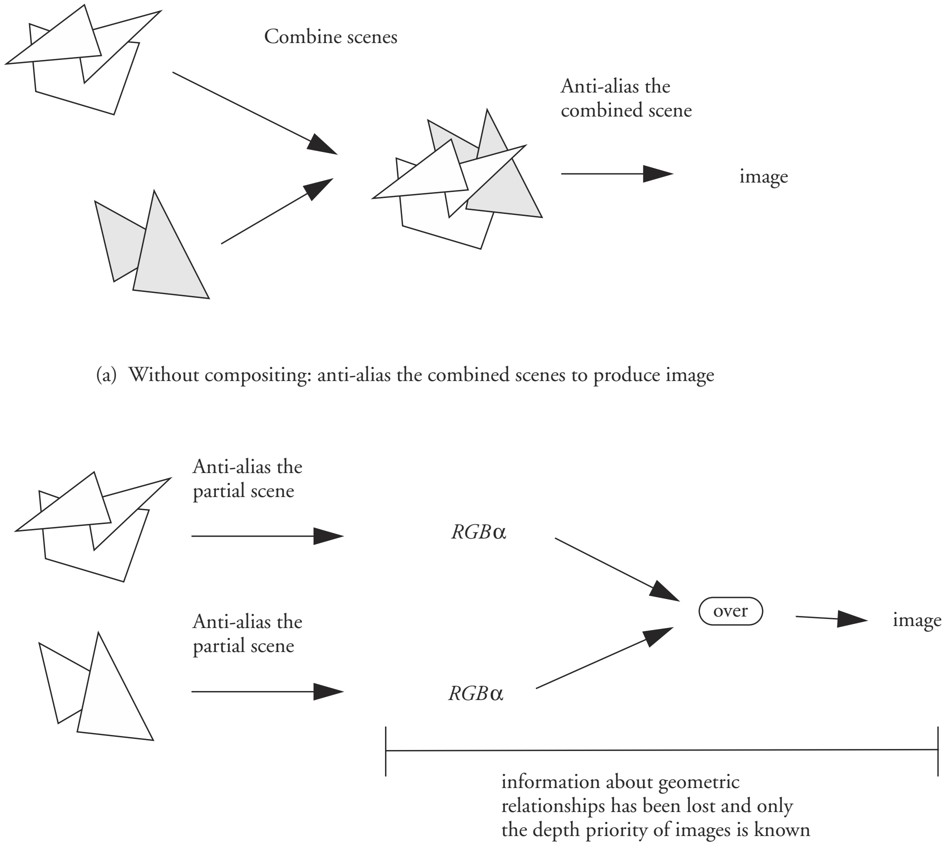


```
for each pixel (i,j)
 if (Occlusion[i][j] == 0)
   Output[i][j] = Foreground[i][j]
 else
   Output[i][j] = Background[i][j]
```

**Figure A.5** Compositing using a one-bit occlusion mask

anti-aliased independently, and, for each pixel, the appropriate color and opacity values are generated in rendering the images. These pixels are then combined using the opacity values (alpha) to form the corresponding pixel in the output image. Note that all geometric relationships are lost between polygons of the two scenes once the image pixels have been generated. Note also that the **over** operator in Figure A.6b could not produce the image of the combined scenes as it appears in Figure A.6a because the operator, as defined above, must give visibility priority to one or the other partial scenes.

The compositing operator, **over**, operates on a color value, *RGB,* and an alpha value between 0 and 1 stored at each pixel. The alpha value can be considered either the opacity of the surface that covers the pixel or the fraction of the pixel covered by an opaque surface, or a combination of the two. The alpha channel can be generated by visible surface algorithms that handle transparent surfaces and/or


Combine scenes
Anti-alias the combined scene
image
Anti-alias the partial scene
Anti-alias the partial scene
$RGB\alpha$
$RGB\alpha$
over
image
information about geometric relationships has been lost and only the depth priority of images is known

(a) Without compositing: anti-alias the combined scenes to produce image

(b) With compositing: anti-alias the partial scenes and then combine to produce image

**Figure A.6** Anti-aliasing combined scenes versus alpha channel compositing

perform some type of anti-aliasing. The alpha value for a given image pixel represents the amount that the pixel's color contributes to the color of the output image when composited with an image behind the given image. To characterize this value as the fraction of the pixel covered by surfaces from the corresponding scene is not entirely accurate. It actually needs to be the coverage of areas contributing to the pixel color weighted by the anti-aliasing filter kernel.[2] In the case of a box filter over nonoverlapping pixel areas, the alpha value equates to the fractional coverage.

[2] The *filter kernel* is the weighting function used to blend color fragments that partially cover a pixel's area.

To composite pixel colors based on the **over** operator, the user computes the new alpha value for the pixel: $\alpha = \alpha_F + (1 - \alpha_F) \cdot \alpha_B$. The composited pixel color is then computed by Equation A.2.

$$(\alpha_F \cdot RGB_F + (1 - \alpha_F) \cdot \alpha_B \cdot RGB_B) / \alpha \tag{Eq. A.2}$$

where $RGB_F$ is the color of the foreground, $RGB_B$ is the color of the background, $\alpha_F$ is the alpha channel of the foreground pixel, and $\alpha_B$ is the alpha channel of the background pixel. The **over** operator is not commutative but associative (see Equation A.3).

$$\begin{aligned} F\,\textbf{over}\,B = &\begin{cases} \alpha_{F\,\textbf{over}\,B} = \alpha_F + (1 - \alpha_F) \cdot \alpha_B & \textbf{over}\text{ operator} \\ RGB_{F\,\textbf{over}\,B} = (\alpha_F \cdot RGB_F + (1 - \alpha_F) \cdot \alpha_B \cdot RGB_B) / \alpha_{F\,\textbf{over}\,B} \end{cases} \\ &A\,\textbf{over}\,B \neq B\,\textbf{over}\,A \qquad \text{noncommutative} \\ &(A\,\textbf{over}\,B)\,\textbf{over}\,C = A\,\textbf{over}\,(B\,\textbf{over}\,C) \qquad \text{associative} \end{aligned} \tag{Eq. A.3}$$

The compositing operator, **over**, assumes that the fragments in the two input images are uncorrelated. The assumption is that the color in the images comes from randomly distributed fragments. For example, if the alpha of the foreground image is 0.5, then the color fragments of what is behind the foreground image will, on average, show through 50 percent of the time. Consider the case of a foreground pixel and middle ground pixel, both with partial coverage in front of a background pixel with full coverage (Figure A.7).

The result of the compositing operation is shown in Equation A.4.

$$\begin{aligned} &RGB_{FMB} \\ &= (\alpha_F \cdot RGB_F + (1 - \alpha_F) \cdot \alpha_M \cdot RGB_M) + (1 - \alpha_{FM}) \cdot \alpha_B \cdot RGB_B \\ &\text{where} \quad \alpha_{FM} = (\alpha_F + (1 - \alpha_F) \cdot \alpha_M) \quad \text{and} \quad \alpha_{FMB} = 1.0 \\ &RGB = 0.5 \cdot RGB_F + 0.25 \cdot RGB_M + 0.25 \cdot RGB_B \end{aligned} \tag{Eq. A.4}$$

If the color fragments are correlated, for example, if they share an edge in the image plane, then the result of the compositing operation is incorrect. The computations are the same, but the result does not accurately represent the configuration of the colors in the combined scene. In the example of Figure A.8, none of the background should show through. A similarly erroneous result occurs if the middle ground image has its color completely on the other side of the edge, in which case none of the middle ground color should appear in the composited pixel. Because geometric information has been discarded, compositing fails to handle these cases correctly.

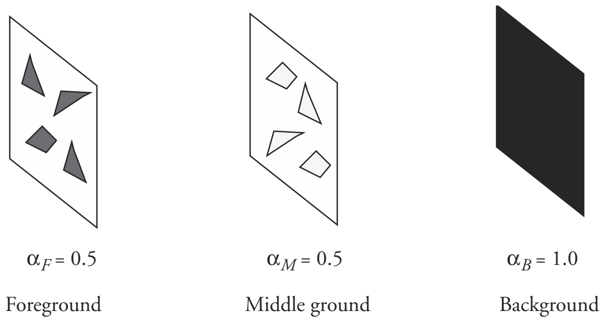


**Figure A.7** Compositing randomly distributed color fragments for a pixel

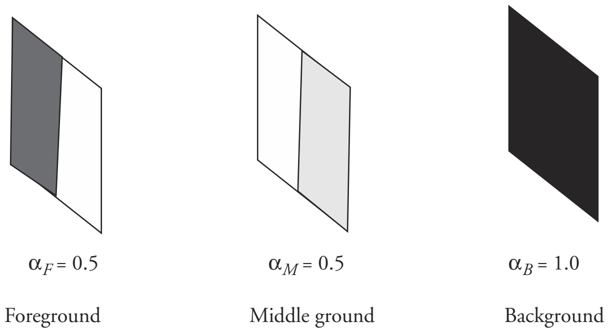


**Figure A.8** Compositing correlated colors for a pixel

When $\alpha_F$ and $\alpha_B$ represent full coverage opacities or uncorrelated partial coverage fractions, the **over** operator computes a valid result. However, if the alphas represent partial coverages that share an edge, then the compositing **over** operator does not have enough information to tell whether, for example, the partial coverages overlap in the pixel area or whether the areas of coverage they represent are partially or completely disjoint. Resolution of this ambiguity requires that additional information be stored at each pixel indicating which part of the pixel is covered by a surface fragment. For example, the A-buffer[3] algorithm [1] provides this information.

*Alpha channel* is a term that represents a combination of the partial coverage and the transparency of the surface or surfaces whose color is represented at the pixel. Notice that when one composits colors, a color always appears in the equation multiplied by its alpha value. It is therefore expedient to store the color value

[3] The *A*-buffer is a *Z*-buffer in which information recorded at each pixel includes the relative depth and coverage of all fragments, in *z*-sorted order, which contribute to the pixel's final color.

already scaled by its alpha value. In the following discussion, lowercase *rgb* refers to a color value that has already been scaled by alpha. Pixels and images whose colors have been scaled by alpha are called *premultiplied.* Uppercase *RGB* refers to a color value that has not been scaled by alpha. In a premultiplied image, the color at a pixel is considered to already be scaled down by its alpha factor, so that if a surface is white with an *RGB* value of (1, 1, 1) and it covers half a pixel as indicated by an alpha value of 0.5, then the *rgb* stored at that pixel will be (0.5, 0.5, 0.5). It is important to recognize that storing premultiplied images is very useful.

## A.2.2 Compositing with Pixel Depth Information

In compositing, independently generated images may sometimes not be disjoint in depth. In such cases, it is necessary to interleave the images in the compositing process. Duff [4] presents a method for compositing 3D rendered images in which depth separation between images is not assumed. An ***rgbαz*** (premultiplied) representation is used for each pixel that is simply a combination of an *rgb* value, the alpha channel, and the *z*-, or depth, value. The *z*-value associated with a pixel is the depth of the surface visible at that pixel; this value is produced by most rendering algorithms.

Binary operators are defined to operate on a pair of images *f* and *b* on a pixel-by-pixel basis to generate a resultant image (Equation A.5). Applying the operators to a sequence of images in an appropriate order will produce a final image.

$$c = f\,\mathbf{op}\,b \tag{Eq. A.5}$$

The first operator to define is the **over** operator. Here, it is defined using colors that have been premultiplied by their corresponding alpha values. The **over** operator blends together the color and alpha values of an ordered pair of images on a pixel-by-pixel basis. The first image is assumed to be "over" or "in front of" the second image. The color of the resultant image is the color of the first image plus the product of the color of the second image and the transparency (one minus opacity) of the first image. The alpha value of the resultant image is computed as the alpha value of the first image plus the product of the transparency of the first and the opacity of the second. Values stored at each pixel of the image, resulting from $c = f$ **over** $b$, are defined as shown in Equation A.6.

$$\begin{aligned} rgb_c &= rgb_f + (1-\alpha_f)\cdot rgb_b \\ \alpha_c &= \alpha_f + (1-\alpha_f)\cdot \alpha_b \end{aligned} \tag{Eq. A.6}$$

For a given foreground image with corresponding alpha values, the foreground *rgb*'s will be unattenuated during compositing with the **over** operator and the background will show through more as $\alpha_f$ decreases. Notice that when $\alpha_f = 1$, then $rgb_c = rgb_f$ and $\alpha_c = \alpha_f = 1$; when $\alpha_f = 0$ (and therefore $rgb_f = 0, 0, 0$), then $rgb_c = rgb_b$ and $\alpha_c = \alpha_b$. Using **over** with more than two layers requires that their ordering in $z$ be taken into account when compositing. The **over** operator can be successfully used when compositing planes adjacent in $z$. If nonadjacent planes are composited, a plane lying between these two cannot be accurately composited; the opacity of the closest surface is not separately represented in the composited image. **Over** is not commutative, although it is associative.

The second operator to define is the $z$-depth operator, **zmin**, which operates on the *rgb,* alpha, and $z$-values stored at each pixel. The **zmin** operator simply selects the ***rgbαz*** values of the closer pixel (the one with the minimum $z$). Values stored at each pixel of the image resulting from $c = f$ **zmin** $b$ are defined by Equation A.7.

$$\begin{aligned} rgb\alpha_c &= \text{if}(z_f < z_b) \qquad \text{then}(rgb\alpha_f) \qquad \text{else}(rgb\alpha_b) \\ z_c &= \min(z_f, z_b) \end{aligned} \tag{Eq. A.7}$$

The order in which the surfaces are processed by **zmin** is irrelevant; it is commutative and associative and can be successfully used on nonadjacent layers.

**Comp** is an operator that combines the action of **zmin** and **over**. As before, each pixel contains an *rgb* value and an $\alpha$ value. However, for an estimate of relative coverage, each pixel has $z$-values at each of its four corners. Because each $z$-value is shared by four pixels, the upper left $z$-value can be stored at each pixel location. This requires that an extra row and extra column of pixels be kept in order to provide the $z$-values for the pixels in the rightmost row and bottommost column of the image; the *rgb* and $\alpha$ values for these pixels in the extra row and column are never used.

To compute $c = f$ **comp** $b$ at a pixel, one must first compute the $z$-values at the corners to see which is larger. There are $2^4 = 16$ possible outcomes of the four corner comparisons. If the comparisons are not the same at all four corners, the pixel is referred to as *confused*. This means that within this single pixel, the layers cross each other in $z$. For any edge of the pixel whose endpoints compare differently, the $z$-values are interpolated to estimate where along the edge the surfaces actually meet in $z$. Figure A.9 illustrates the implied division of a pixel into areas of coverage based on the relative $z$-values at the corners of the foreground and background pixels.

Regarding symmetry, there are really four cases to consider in computing $\beta$, the coverage fraction for the surface $f$: whole, corner, split, and two-opposite corners. *Whole* refers to the simple cases in which the entire pixel is covered by one surface (cases a and p in Figure A.9); in this case, $\beta$ equals either 1 or 0. *Corner* refers to the cases in which one surface covers the entire pixel except for one corner (cases b, c, e, h, i, l, n, and o in Figure A.9). If $f$ is the corner surface, then the coverage is $\beta = 1/2 \cdot s \cdot t$, where $s$ and $t$ represent the fraction of the edge indicated by the $z$-value interpolation as measured from the corner vertex. If $b$ is the corner surface,

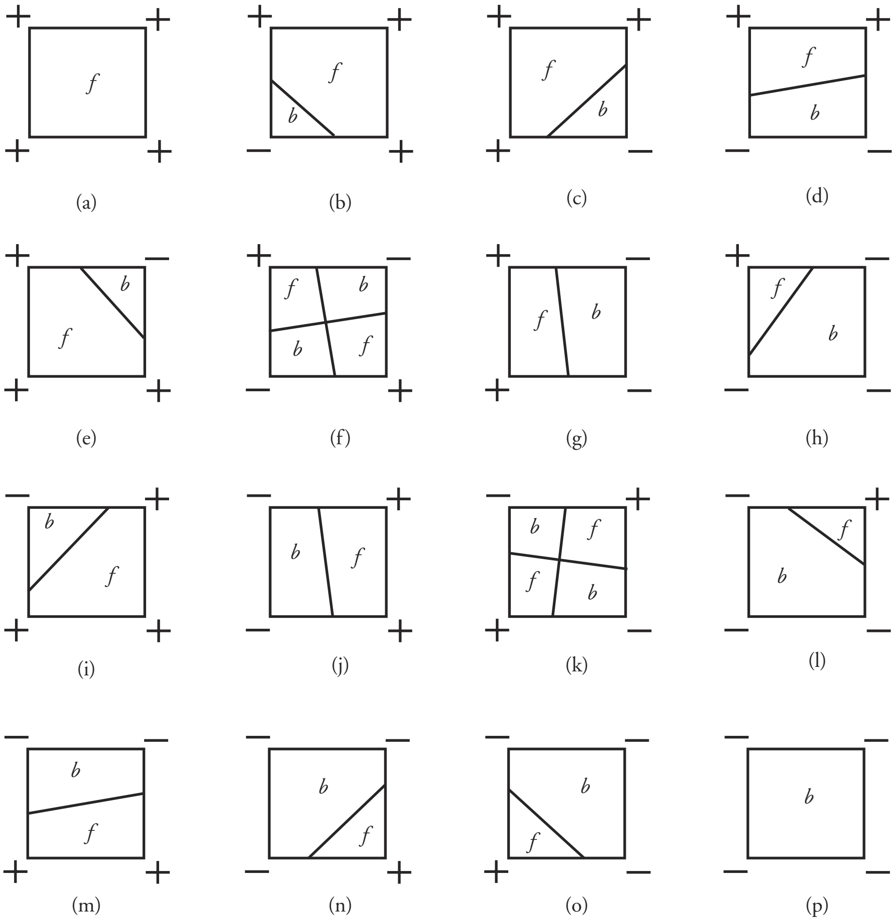


**Figure A.9** Categories of pixels based on $z$ comparisons at the corners; the label at each corner is sign($z_{\mathbf{f}}$ - $z_{\mathbf{b}}$).

then $\beta = 1.0 - (1/2 \cdot s \cdot t)$. *Split* refers to the situation in which the vertices of opposite edges are tagged differently (cases d, g, j, and m in Figure A.9); the coverage of the surface is $\beta = s + t/2$, where $s$ and $t$ represent the fraction of the edge indicated by the $z$-value interpolation as measured from the vertices tagged the same toward the other vertices. The fraction for the other surface would be $\beta = 1.0 - s + t/2$. If diagonally opposite vertices are tagged differently (cases f and k in Figure A.9), the equation for β is the same as for the "split" case, with $s$ and $t$ measured from vertices tagged the same and calculated individually for each of the two vertices.

Once β is computed, then that fraction of the pixel is considered as having the surface *f* as the front surface and the rest of the pixel is considered as having the surface *b* as the front surface. The **comp** operator is defined as the linear blend, according to the interpolant β, of two applications of the **over** operator—one with surface *f* in front and one with surface *b* in front (Equation A.8).

$$\begin{aligned} rgb_c &= \beta \cdot (rgb_f + (1-\alpha_f) \cdot rgb_b) + (1-\beta) \cdot (rgb_b + (1-\alpha_b) \cdot rgb_f) \\ \alpha_c &= \beta \cdot (\alpha_f + (1-\alpha_f) \cdot \alpha_b) + (1-\beta) \cdot (\alpha_b + (1-\alpha_b) \cdot \alpha_f) \\ z_c &= \min(z_f, z_b) \end{aligned} \qquad \text{(Eq. A.8)}$$

**Comp** decides on a pixel-by-pixel basis which image represents a surface in front of the other, including the situation in which the surfaces change relative depth within a single pixel. Thus, images that are not disjoint in depth can be successfully composited using the **comp** operator, which is commutative (with β becoming **1**-β) as well as associative.

## A.3 Displaying Moving Objects: Motion Blur

If it is assumed that objects are moving in time, it is worth addressing the issue of effectively displaying these objects in a frame of animation [5] [6]. In the same way that aliasing is a sampling issue in the spatial domain, it is also a sampling issue in the temporal domain. As frames of animation are calculated, the positions of objects change in the image, and this movement changes the color of a pixel as a function of time. In an animation, the color of a pixel is sampled in the time domain. If the temporal frequency of a pixel's color function is too high, then the temporal sampling can miss important information.

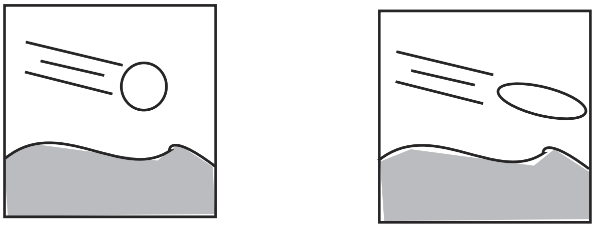
Consider an object moving back and forth in space in front of an observer. Assume the animation is calculated at the rate of thirty frames per second. Now assume that the object starts on the left side of the screen and moves to the right side in one-sixtieth of a second and moves back to the left side in another one-sixtieth of a second. This means that every thirtieth of a second (the rate at which frames are calculated and therefore the rate at which positions of objects are sampled) the object is on the left side of the screen. The entire motion of the object is missed because the sampling rate is too low to capture the high-frequency motion of the object, resulting in temporal aliasing. Even if the motion is not this contrived, displaying a rapidly moving object by a single instantaneous sample can result in motions that appear jerky and unnatural. As mentioned in Chapter 1, images of a fast-moving object can appear to be disjointed, resulting in jerky motion similar to that of live action under a strobe light (and this is often called *strobing*).

Conventional animation has developed its own techniques for representing fast motion. Speed lines can be added to moving objects, objects can be stretched in the direction of travel, or both speed lines and object stretching can be used [7] (see Figure A.10).

There is an upper limit on the amount of detail the human eye can resolve when viewing a moving object. If the object moves too fast, mechanical limitations of muscles and joints that control head and eye movement will fail to maintain an accurate track. This will result in an integration of various samples from the environment as the eye tries to keep up. If the eye is not tracking the object and the object moves across the field of view, receptors will again receive various samples from the environment integrated together, forming a cumulative effect in the eye-brain. Similarly, a movie camera will open its shutter for an interval of time, and an object moving across the field of view will create a blurred image on that frame of the film. This will smooth out the apparent motion of the object. In much the same way, the synthetic camera can (and should) consider a frame to be an interval of time instead of an instance in time. Unfortunately, accurately calculating the effect of moving objects in an image requires a nontrivial amount of computation.

To fully consider the effect that moving objects have on an image pixel, one must take into account the area of the image represented by the pixel, the time interval for which a given object is visible in that pixel, the area of the pixel in which the object is visible, and the color variation of the object over that time interval in that area, as well as such dynamic effects as rotating textured objects, shadows, and specular highlights [6].

There are two analytic approaches to calculating motion blur: continuous and discrete. Continuous approaches attempt to be more accurate but are only tractable in limited situations. Discrete approaches, while less accurate, are more generally applicable and more robust; only discrete approaches are considered here. Ray



Speed lines

Speed lines and object stretching

**Figure A.10** Methods used in conventional animation for displaying speed

tracing is probably the easiest domain in which to understand the discrete process. It is common in ray tracing to generate more than one ray per pixel in order to spatially anti-alias the image. These rays can be distributed in a regular pattern or stochastically [2] [3] [8]. To incorporate temporal anti-aliasing, one need only take the rays for a pixel and distribute them in time as well as in space. In this case, the frame is not considered to be an instant in time but rather an interval. The interval can be broken down into subintervals and the rays distributed into these subintervals. The various samples are then filtered to form the final image. See Figure A.11 (Plate 12) for an example.

One of the extra costs associated with temporal anti-aliasing is that the motion of the objects must be computed at a higher rate than the frame rate. For example, if a 4x4 grid of subsamples are used for anti-aliasing in ray tracing and these are distributed over separate time subintervals, then the motion of the objects must be calculated at sixteen times the frame rate. If the subframe motion is computed using complex motion control algorithms, this may be a significant computational cost. Linear interpolation is often used to estimate the positions of objects at subframe intervals.

Although discussed above in terms of distributed ray tracing, this same strategy can be used with any display algorithm, as Korein and Badler [5] note. Multiple frame buffers can be used to hold rendered images at subframe intervals. These can then be filtered to form the final output image. The rendering algorithm can be a standard *z*-buffer, a ray tracer, a scanline algorithm, or any other technique. Because of the discrete sampling, this is still susceptible to temporal aliasing arti-

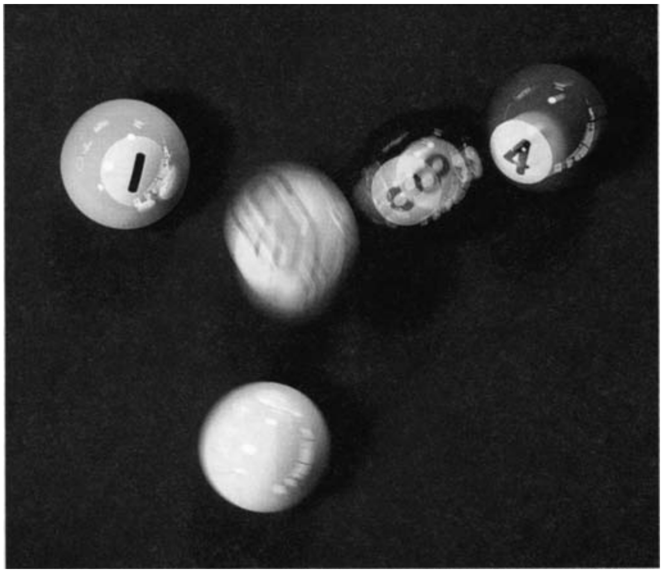


**Figure A.11** An example of synthetic (calculated) motion blur

facts if the object is moving too fast relative to the size of its features in the direction of travel; instead of motion blur, multiple images of the object may result because intermediate pixels are effectively “jumped over” by the object during the sampling process.

In addition to analytic methods, hand manipulation of the object shape by the animator can reduce the amount of strobing. For example, the animator can stretch the object in the direction of travel. This will tend to reduce or eliminate the amount of separation between images of the object in adjacent frames.

## A.4 Drop Shadows

The shadow cast by an object onto a surface is an important visual cue in establishing the distance between the two. Contact shadows, or shadows produced by an object contacting the ground, are especially important. Without them, objects appear to float just above the ground plane. In high-quality animation, shadow decisions are prompted by lighting, cinematography, and visual understanding considerations. However, for most other animations, computational expense is an important concern and computing all of the shadows cast by all objects in the scene onto all other objects in the scene is overkill. Much computation can be saved if the display system supports the user specification of which objects in a scene cast shadows onto which other objects in a scene. For example, the self-shadowing[4] of an object is often not important to understanding the scene visually. Shadows cast by moving objects onto other moving objects are also often not of great importance. These principles can be observed in traditional hand-drawn animation in which only a select set of shadows is drawn. In Figure A.12 (Plate 13), shadows on the ground beneath the characters help to locate them in the environment, but the shadows of one character on another are not included in the rendering.

By far the most useful type of shadow in animated sequences is the drop shadow. The drop shadow is the shadow that an object projects to the ground plane. The drop shadow lets the viewer know how far above the ground plane an object is as well as the object’s relative depth and, therefore, relative size (see Figures A.13 [Plate 14] and A.14 [Plate 15]).

Drop shadows can be produced in several different ways. When as object is perspectively projected from the light source to a flat ground plane, an image of the object can be formed (see Figure A.15). If this image is colored dark and displayed

[4] *Self-shadowing* refers to an object casting shadows onto its own surface.

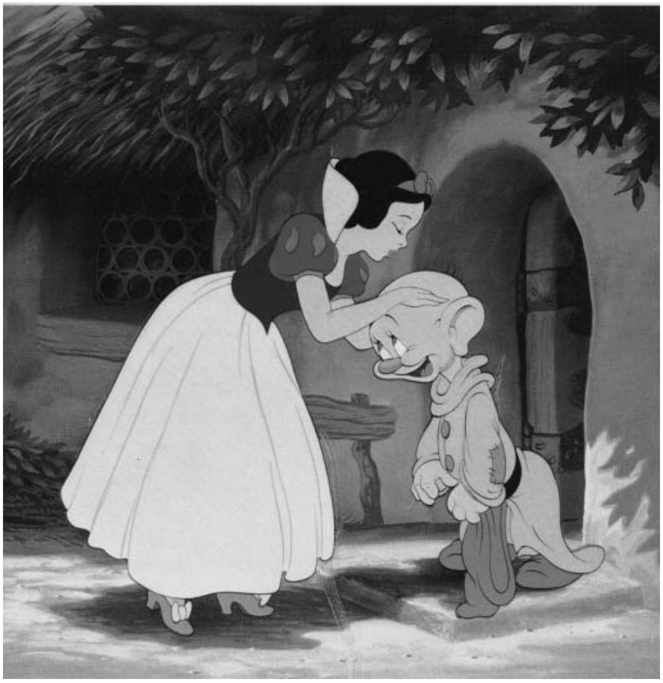


**Figure A.12** Animation frame showing selective shadows

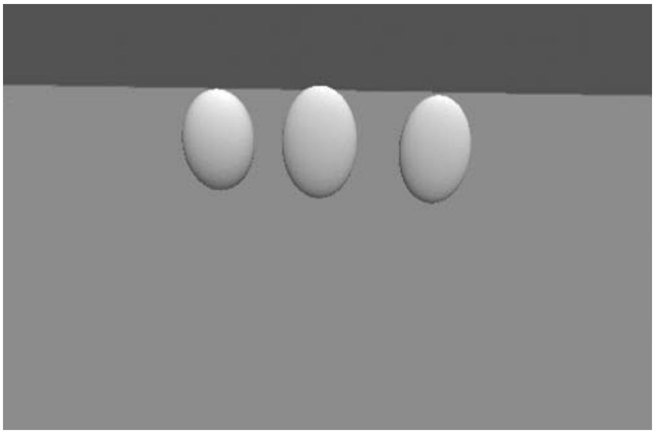


**Figure A.13** Scene without drop shadows; without shadows, it is nearly impossible to estimate relative heights and distances if the sizes of the objects are not known

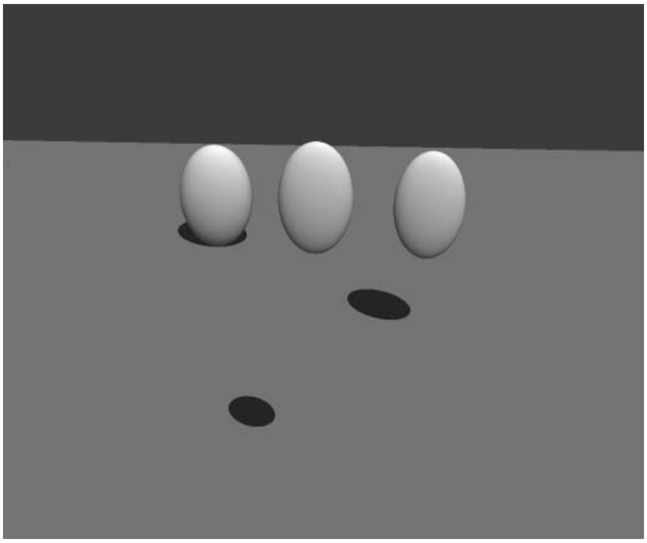


**Figure A.14** Scene with drop shadows indicating relative depth and, therefore, relative height and size

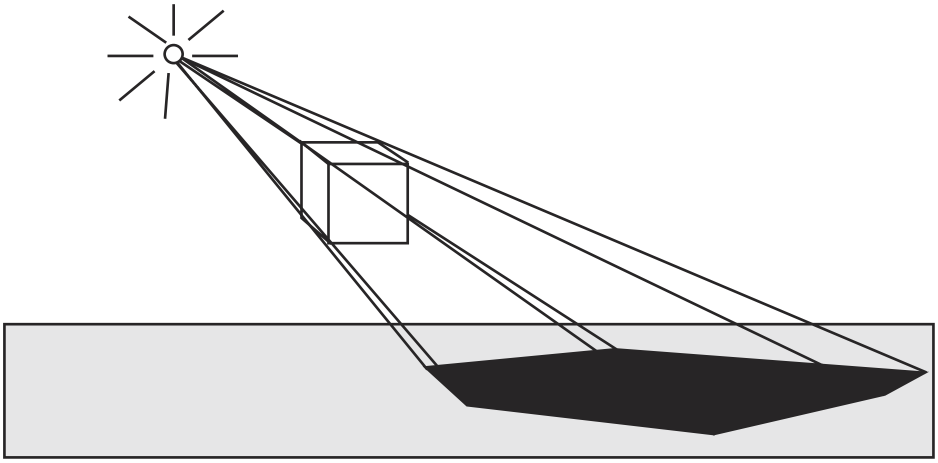


**Figure A.15** Computing the drop shadow by perspective projection

on the ground plane (as a texture map, for example), then it is an effective shortcut.

Another inexpensive method for creating a drop shadow is to make a copy of the object, scale it flat vertically, color it black (or make it dark and transparent), and position it just on top of the ground plane (see Figures A.16 and A.17 [Plate

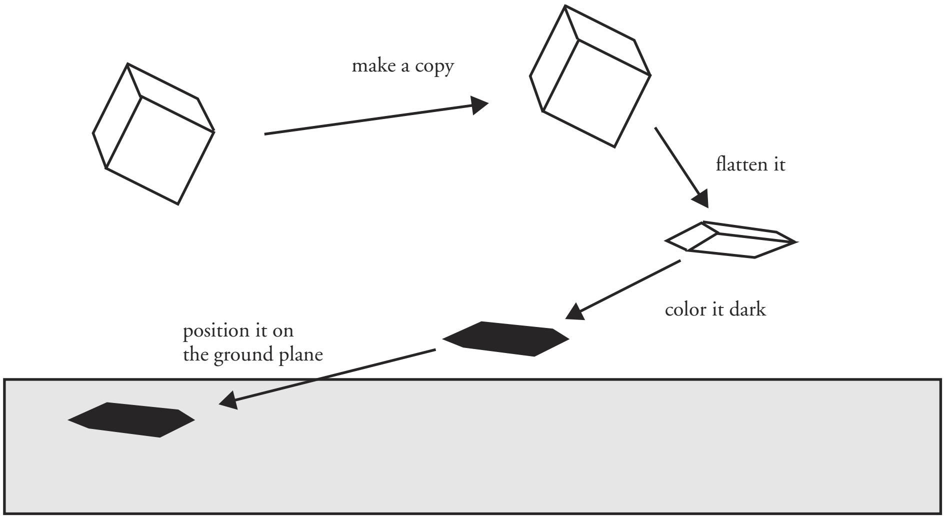


**Figure A.16** Computing the drop shadow by flattening a copy of the object

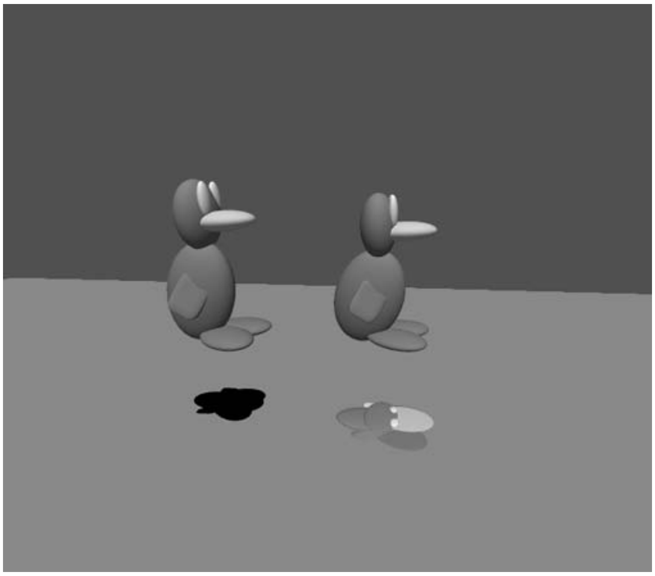


**Figure A.17** Drop shadow using a flattened copy of the object

16]). The drop shadow has the correct silhouette for a light source directly overhead but without the computational expense of the perspective projection method.

The drop shadow can be effective even if it is only an approximation of the real shadow that would be produced by a light source in the environment. The height of the object can be effectively indicated by merely controlling the relative size and softness of a drop shadow, which only suggests the shape of the object that casts it. For simple drop shadows, circles can be used, as in Figure A.18 (Plate 17).

Globular shapes that may more closely indicate the shapes of the original objects can also be used as shadows. The drop shadow, however it is produced, can be represented on the ground plane in several different ways. It can be colored black and placed just above the ground plane. It can be made into a darkly colored transparent object so that any detail of the ground plane shows through. It can be colored darker in the middle with more transparency toward the edges to simulate a shadow's penumbra[5] (Figure A.19).

When placing a shadow over the ground plane, one must take care to keep it close enough to the ground so that it does not appear as a separate object and at the same time does not conflict with the ground geometry. To avoid the problems of using a separate geometric element to represent the drop shadow, the user can incorporate the shadow directly into the texture map of the ground plane.

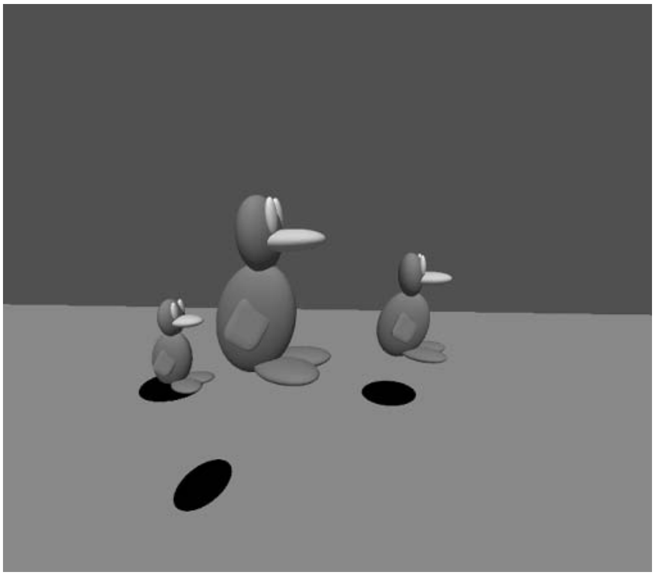


**Figure A.18** Circular drop shadows

[5] The *penumbra* is the area partially shadowed by an opaque body; it receives partial illumination from a light source. The *umbra* is the area completely shadowed by an opaque body; it receives no illumination from the light source.

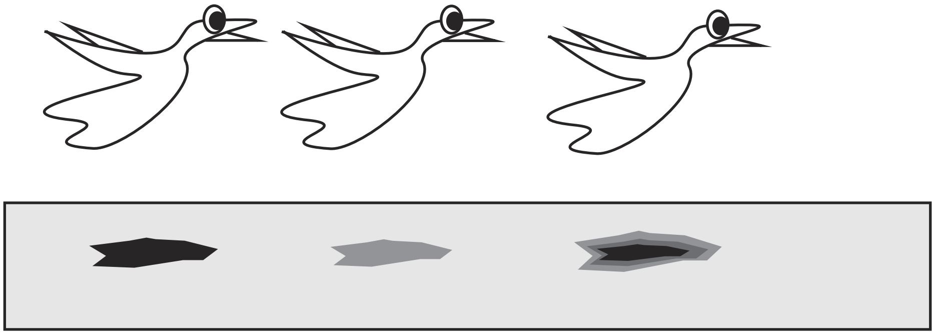


**Figure A.19** Various ways to render a drop shadow: black, transparent, and black with transparent edges

## A.5 Summary

Although these techniques are not concerned with specifying or controlling the motion of graphical objects, they are important to the process of generating images for computer animation. Double buffering helps to smoothly update a display of images. Compositing helps to conserve resources and combine elements from different sources. Motion blur prevents fast-moving objects from appearing jerky and distracting to the viewer. Shadows help to locate objects relative to surfaces, but only those shadows that effectively serve such a purpose need to be generated.

## References

1. L. Carpenter, "The A-Buffer Hidden Surface Method," *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 103–108 (July 1984, Minneapolis, Minnesota).
2. R. Cook, "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics,* 5 (1), pp. 51–71, January 1986.
3. R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 137–146 (July 1984, Minneapolis, Minn.).
4. T. Duff, "Compositing 3-D Rendered Images," *Computer Graphics* (Proceedings of SIGGRAPH 85), 19 (3), pp. 41– 44 (August 1985, San Francisco, Calif.). Edited by B. A. Barsky.
5. J. Korein and N. Badler, "Temporal Anti-Aliasing in Computer Generated Animation," *Computer Graphics* (Proceedings of SIGGRAPH 83), 17 (3), pp. 377–388 (July 1983, Detroit, Mich.).

6. M. Potmesil and I. Chadkravarty, “Modeling Motion Blur in Computer Generated Images,” *Computer Graphics* (Proceedings of SIGGRAPH 83), 17 (3), pp. 389–400 (July 1983, Detroit, Mich.).
7. F. Thomas and O. Johnson, *The Illusion of Life*, Hyperion, NY, 1981.
8. T. Whitted, “An Improved Illumination Model for Shaded Display,” *Communication of the ACM*, 23 (6), pp. 343–349 (June 1980).

APPENDIX B

# Background Information and Techniques

## B.1 Vectors and Matrices

A *vector* is a one-dimensional list of values. This list can be shown as either a row vector or a column vector (e.g., Equation B.1). In general, a matrix is an *n*-dimensional array of values. For purposes of this book, a matrix is two-dimensional (e.g., Equation B.2).

$$\begin{bmatrix} a & b & c \end{bmatrix} \qquad \begin{bmatrix} a \\ b \\ c \end{bmatrix} \tag{Eq. B.1}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{Eq. B.2}$$

Matrices are multiplied together by taking the *i*th row of the first matrix and multiplying each element by the corresponding element of the *j*th column of the second matrix and summing all the products to produce the *i,j*th element. When computing $C = A \cdot B$, where *A* has *v* elements in each row and *B* has *v* elements in each column, an element $C_{ij}$ is computed according to Equation B.3.

$$\begin{aligned} C_{ij} &= A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} + A_{i3} \cdot B_{3j} + \ldots + A_{iv} \cdot B_{vj} \\ &= \sum_{k=1}^{v} A_{ik} \cdot B_{kj} \end{aligned} \tag{Eq. B.3}$$

The "inside" dimension of the matrices must match in order for the matrices to be multiplied together. That is, if *A* and *B* are multiplied and *A* is a matrix with *U* rows and *V* columns (a $U \times V$ matrix), then *B* must be a $V \times W$ matrix; the result will be a $U \times W$ matrix. Said another way, the number of columns (the number of elements in a row) of *A* must be equal to the number of rows (the number of elements in a column) of *B*. As a more concrete example, consider multiplying two 3x3 matrices. Equation B.4 shows the computation for the first element.

$$\begin{aligned} A \cdot B &= \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} \\ &= \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & \ldots & \ldots \\ \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots \end{bmatrix} \end{aligned} \tag{Eq. B.4}$$

The *transpose* of a vector or matrix is the original vector or matrix with its rows and columns exchanged (e.g., Equation B.5). The *identity matrix* is a square matrix with ones along its diagonal and zeros elsewhere (e.g., Equation B.6). The *inverse* of a square matrix when multiplied by the original matrix produces the identity matrix (e.g., Equation B.7). The *determinant* of a 3x3 matrix is formed as shown in Equation B.8. The determinant of matrices greater than 3x3 can be defined recursively. First, define an element's *submatrix* as the matrix formed when removing the element's row and column from the original matrix. The determinant is formed by considering any row, element by element. The determinant is the first element of the row times the determinant of its submatrix, minus the second element of the row times the determinant of its submatrix, plus the third element of the row times the determinant of its submatrix, and so on. The sum is formed for the entire row, alternating additions and subtractions.

$$\begin{bmatrix} a & b & c \end{bmatrix}^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} \qquad \text{(Eq. B.5)}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. B.6)}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} \cdot \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. B.7)}$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \cdot (e \cdot i - f \cdot h) - b \cdot (f \cdot g - d \cdot i) + c \cdot (d \cdot h - e \cdot g) \qquad \text{(Eq. B.8)}$$

## B.1.1 Inverse Matrix and Solving Linear Systems

The inverse of a matrix is useful in computer graphics to represent the inverse of a transformation and in computer animation to solve a set of linear equations. There are various ways to compute the inverse. One common method, which is also useful for solving sets of linear equations, is LU decomposition. The basic idea is that a square matrix, for example, a 4x4, can be decomposed into a lower triangular matrix times an upper triangular matrix. How this is done is discussed later. For now, it is assumed that the LU decomposition is available (Equation B.9).

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

$$= L \cdot U = \begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} \qquad \text{(Eq. B.9)}$$

The decomposition of a matrix *A* into the *L* and *U* matrices can be used to easily solve a system of linear equations. For example, consider the case of 4 unknowns (*x*) and four equations shown in Equation B.10. Use of the decomposition permits the system of equations to be solved by forming two systems of equations using triangular matrices (Equation B.11).

$$\begin{aligned} A_{11} \cdot x_1 + A_{12} \cdot x_2 + A_{13} \cdot x_3 + A_{14} \cdot x_4 &= b_1 \\ A_{21} \cdot x_1 + A_{22} \cdot x_2 + A_{23} \cdot x_3 + A_{24} \cdot x_4 &= b_2 \\ A_{31} \cdot x_1 + A_{32} \cdot x_2 + A_{33} \cdot x_3 + A_{34} \cdot x_4 &= b_3 \\ A_{41} \cdot x_1 + A_{42} \cdot x_2 + A_{43} \cdot x_3 + A_{44} \cdot x_4 &= b_4 \end{aligned}$$

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$$A \cdot x = b \qquad \text{(Eq. B.10)}$$

$$\begin{aligned} A \cdot x &= b \\ (L \cdot U) \cdot x &= b \\ L \cdot (U \cdot x) &= b \\ U \cdot x &= y \\ L \cdot y &= b \end{aligned} \qquad \text{(Eq. B.11)}$$

This solves the original set of equations. The advantage of doing it this way is that both of the last two equations resulting from the decomposition involve triangular matrices and, therefore, can be solved trivially with simple substitution methods. For example, Equation B.12 shows the solution to $L \cdot y = b$. Notice that by solving the equations in a top to bottom fashion, the results from the equations of previous rows are used so that there is only one unknown in any equation being considered. Once the solution for $y$ has been determined, it can be used to solve for $x$ in $U \cdot x = y$ using a similar approach. Once the LU decomposition of $A$ is formed, it can be used repeatedly to solve sets of linear equations that differ only in right-hand sides, such as those for computing the inverse of a matrix. This is one of the advantages of LU decomposition over methods such as Gauss-Jordan elimination.

$$L \cdot y = b$$

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

First row: $L_{11} \cdot y_1 = b_1$

$$y_1 = b_1 / L_{11}$$

Second row: $L_{21} \cdot y_1 + L_{22} \cdot y_2 = b_2$

$$y_2 = (b_2 - (L_{21} \cdot y_1)) / L_{22}$$

Third row: $L_{31} \cdot y_1 + L_{32} \cdot y_2 + L_{33} \cdot y_3 = b_3$

$$y_3 = (b_3 - (L_{31} \cdot y_1) - (L_{32} \cdot y_2)) / L_{33}$$

Fourth row: $L_{41} \cdot y_1 + L_{42} \cdot y_2 + L_{43} \cdot y_3 + L_{44} \cdot y_4 = b_4$

$$y_4 = (b_4 - (L_{41} \cdot y_1) - (L_{42} \cdot y_2) - (L_{43} \cdot y_3)) / L_{44} \quad \text{(Eq. B.12)}$$

The decomposition procedure sets up equations and orders them so that each is solved simply. Given the matrix equation for the decomposition relationship, one can construct equations on a term-by-term basis for the $A$ matrix. This results in $N^2$ equations with $N^2 + N$ unknowns. As there are more unknowns than equations, $N$ elements are set to some arbitrary value. A particularly useful set of values is $L_{ii} = 1.0$. Once this is done, the simplest equations (for $A_{11}$, $A_{12}$, etc.) are used to establish values for some of the $L$ and $U$ elements. These values are then used in the more complicated equations. In this way the equations can be ordered so there is only one unknown in any single equation by the time it is evaluated. Consider the case of a 4x4 matrix. Equation B.13 repeats the original matrix equation for reference and then shows the resulting sequence of equations in which the underlined variable is the only unknown.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 \\ L_{41} & L_{42} & L_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

$$\begin{array}{ll}
\text{For the first column of } A & \underline{U_{11}} = A_{11} \\
 & \underline{L_{21}} \cdot U_{11} = A_{21} \\
 & \underline{L_{31}} \cdot U_{11} = A_{31} \\
 & \underline{L_{41}} \cdot U_{11} = A_{41} \\
\text{For the second column of } A & \underline{U_{12}} = A_{12} \\
 & L_{21} \cdot U_{12} + \underline{U_{22}} = A_{22} \\
 & L_{31} \cdot U_{12} + \underline{L_{32}} \cdot U_{22} = A_{32} \\
 & L_{41} \cdot U_{12} + \underline{L_{42}} \cdot U_{22} = A_{42} \\
\text{For the third column of } A & \underline{U_{13}} = A_{13} \\
 & L_{21} \cdot U_{13} + \underline{U_{23}} = A_{23} \\
 & L_{31} \cdot U_{13} + L_{32} \cdot U_{23} + \underline{U_{33}} = A_{33} \\
 & L_{41} \cdot U_{13} + L_{42} \cdot U_{23} + \underline{L_{43}} \cdot U_{33} = A_{43} \\
\text{For the fourth column of } A & \underline{U_{14}} = A_{14} \\
 & L_{21} \cdot U_{14} + \underline{U_{24}} = A_{24} \\
 & L_{31} \cdot U_{14} + L_{32} \cdot U_{24} + \underline{U_{34}} = A_{34} \\
 & L_{41} \cdot U_{14} + L_{42} \cdot U_{24} + L_{43} \cdot U_{34} + \underline{U_{44}} = A_{44}
\end{array}$$

**(Eq. B.13)**

Notice a two-phase pattern in each column in which terms from the $U$ matrix from the first row to the diagonal are determined first, followed by terms from the $L$ matrix below the diagonal. This pattern can be generalized easily to matrices of arbitrary size [16], as shown in Equation B.14. The computations for each column $j$ must be completed before proceeding to the next column.

$$\begin{aligned} U_{ij} &= A_{ij} - \sum_{k=1}^{i-1} L_{ik} \cdot U_{kj} && \text{for } i = 1, \ldots, j \\ L_{ij} &= \frac{1}{U_{jj}}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} \cdot U_{kj}\right) && \text{for } i = j+1, \ldots, n \end{aligned} \qquad \textbf{(Eq. B.14)}$$

So far this is fairly simple and easy to follow. Now comes the complication—*partial pivoting*. Notice that some of the equations require a division to compute the value for the unknown. For this computation to be numerically stable (i.e., the numerical error less sensitive to particular input values), this division should be by a relatively large number. By reordering the rows, one can exert some control over what divisor is used. Reordering rows does not affect the computation if the matrices are viewed as a system of linear equations; reordering obviously matters if the inverse of a matrix is being computed. However, as will be shown later, as long as the reordering is recorded, it can easily be undone when the inverse matrix is formed from the $L$ and $U$ matrices.

Consider the first column of the 4x4 matrix. The divisor used in the last three equations is $U_{11}$, which is equal to $A_{11}$. However, if the rows are reordered, then a different value might end up as $A_{11}$. So the objective is to swap rows so that the largest value (in the absolute value sense) of $A_{11}$, $A_{21}$, $A_{31}$, $A_{41}$ ends up at $A_{11}$, which makes the computation more stable. Similarly, in the second column, the divisor is $U_{22}$, which is equal to $A_{22} - (L_{21} \cdot U_{12})$. As in the case of the first column, the rows below this are checked to see if a row swap might make this value larger. The row above is not checked because that row was needed to calculate $U_{12}$, which is needed in the computations of the rows below it. For each successive column, there are fewer choices because the only rows that are checked are the ones below the topmost row that requires the divisor.

There is one other modification to partial pivoting. Because any linear equation has the same solution under a scale factor, an arbitrary scale factor applied to a linear equation could bias the comparisons made in the partial pivoting. To factor out the effect of an arbitrary scale factor when comparing values for the partial pivoting, one scales the coefficients for that row, just for the purposes of the comparison, so that the largest coefficient of a particular row is equal to one. This is referred to as *implicit pivoting*.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ L_{21} & U_{22} & U_{23} & U_{24} \\ L_{31} & L_{32} & U_{33} & U_{34} \\ L_{41} & L_{42} & L_{43} & U_{44} \end{bmatrix}$$

**Figure B.1** In-place computation of the *L* and *U* values assuming no row exchanges

In the code that follows, the LU decomposition approach is used to solve a system of linear equations and is broken down into several procedures. These procedures follow those found in *Numerical Recipes* [16]. The first procedure (**LUdecomp**) performs the actual decomposition by replacing the values of the input matrix with the values of *L* and *U* (Figure B.1). Notice that the diagonal elements of the *L* matrix do not have to be stored because they are routinely set to one. If row exchanges take place, then the rows of the matrices in Figure B.1 will be mixed up. For solving the linear system of equations, the row exchanges have no effect on the computed answers. However, the row exchanges are recorded in a separate array so that they can be undone for the formation of the inverse matrix.

After the execution of **LUdecomp**, the A matrix contains the elements of the *L* and *U* matrices. This matrix can then be used either for solving a linear system of equations or for computing the inverse of a matrix. The previously discussed simple substitution methods can be used to solve the equations that involve triangular matrices. In the code below, the subroutine **LUsubstitute** is called with the newly computed *A* matrix, the dimension of *A,* the vector of row swaps, and the right-hand vector (i.e., the *b* in $A \cdot x = b$).

One of the advantages of the LU decomposition is that the decomposition matrices can be reused if the only change in a system of linear equations is the right-hand vector of values. In such a case, the routine **LUdecomp** only needs to be called once to form the LU matrices. The substitution routine, **LUsubstitute**, needs to be called with each new vector of values (remember to reuse the *A* matrix that holds the LU decomposition).

To perform matrix inversion, use the *L* and *U* matrices repeatedly with the *b* matrix holding column-by-column values of the identity matrix. In this way the inverse matrix is built up column by column.

```
/* LU Decomposition
*  with partial implicit pivoting
*  partial means that the pivoting only happens by row
*  implicit means that the pivots are scaled by the maximum value in the
   row
*/
```

```
/* ================================================================= */
/* LUdecomp
*  inputs: A matrix of coefficients
*     n — dimension of A
* outputs: A matrix replaced with L and U diagonal matrices (diagonal
  values of L == 1)
*     Rowswaps — vector to keep track of row swaps
*     Val — indicator of odd/even number of row swaps
*/
int LUdecomp(float **A,int n,int *rowswaps,int *val)
{
   float  epsilon,*rowscale, temp;
   float  sum;
   float  pvt;
   int        ipvt;
   int        i,j,k;
   rowscale = (float *)malloc(sizeof(float)*n);

   epsilon = 0.00000000001;  /* small value to avoid division by zero */
   *val = 1;                       /* even/odd indicator (valence) */

   /* initialize the rowswap vector to indicate no swaps */
   for (i=0; i<n; i++) rowswaps[i] = i;

   /* for each row, find largest (in absolute value sense) element and
      record in rowscale */
   for (i=0; i<n; i++) {
      temp = fabs(A[i][0]);
      for (j=1; j<n; j++)
         if (fabs(A[i][j]) > temp) temp = fabs(A[i][j]);
      if (temp == 0) return(-1); /* got a row of all zeros — can't deal
      with that */
      rowscale[i] = 1/temp; /* later we need to divide by largest
      element */
   }

   /*  loop through the columns  of A (and U) */
   for (j=0; j<n; j++) {
      /* do the rows down to the diagonal — these don't need a division
      so no swap */
      for (i=0; i<j; i++) {
         sum = A[i][j];
         for (k=0; k<i; k++) sum = sum - A[i][k]*A[k][j];
         A[i][j] = sum;
      }
      /* do the rows from the diagonal down */
      pvt = 0.0;
      ipvt = -1;
      for (i=j; i<n; i++) {
         sum = A[i][j];
         for (k=0; k<j; k++) sum = sum - A[i][k]*A[k][j];
         A[i][j] = sum;
         /* calculate the scaled value for pivoting consideration */
            temp = rowscale[i]*fabs(sum);
            if (temp >= pvt) {ipvt = i; pvt = temp;}
      }

      /* if a better pivot value is found, interchange the rows */
         if (j != ipvt) {
            for (k=0; k<n; k++) {
               temp = A[ipvt][k];
               A[ipvt][k] = A[j][k];
               A[j][k] = temp;
            }
            *val = -(*val);  /* keep track of even/odd number
            interchanges  */
            rowscale[ipvt] = rowscale[j];  /* and record which was
            swapped  */
         }
      rowswaps[j] = ipvt;

      if (A[j][j] == 0.0) A[j][j] = epsilon; /* to guard against
      divisions by zero */
      /* now the row is ready for division */
      for (i=j+1; i<n; i++)  A[i][j] = A[i][j]/A[j][j];
   }
   return 1;
}

/* ================================================================== */
/* LUsubstitute
*  inputs: A — matrix holding the L and U matrix values as a result of
   LUdecomp
*          n — dimension of A
*          Rowswaps — vector holding a record of the row swaps performed
           in LUdecomp
*          b — vector of right-hand values as in Ax = b
```

```
*/
void LUsubstitute(float **A,int n,int *rowswaps,float *b)
{
   int    i,j,ib;
   float  sum;
   int    m;

      /* row swap version */
      ib = -1;
      for (i=0; i<n; i++) {
         m = rowswaps[i];
         sum = b[m];
         b[m] = b[i];
         if (ib != -1) {
            for (j=ib; j<i; j++) sum = sum-A[i][j]*b[j];
         }
         else {
            if (sum != 0.0) ib = i;
         }
         b[i] = sum;
      }

      for (i=n-1; i>=0; i--) {
         sum = b[i];
         for (j=i+1; j<n; j++) sum = sum - A[i][j]*b[j];
         b[i] = sum/A[i][i];
      }
   return;
}
```

## B.1.2 Singular Value Decomposition

Singular value decomposition (SVD) is a popular method used for solving linear least-squares problems ($A \cdot x = b$, where the number of rows of $A$ is greater than the number of columns). It gives the user information about how ill conditioned the set of equations is and allows the user to step in and remove sources of numerical inaccuracy.

As with LU decomposition, the first step decomposes the $A$ matrix into more than one matrix (Equation B.15). In this case, an $M \times N$ $A$ matrix is decomposed into a column-orthogonal $M \times N$ $U$ matrix, a diagonal $N \times M$ $W$ matrix, and an $N \times N$ orthogonal $V$ matrix.

$$A = U \cdot W \cdot V^T \quad \text{(Eq. B.15)}$$

The magnitude of the elements in the *W* matrix indicates the potential for numerical problems. Zeros on the diagonal indicate singularities. Small values (where *small* is user defined) indicate the potential of numerical instability. It is the user's responsibility to inspect the *W* matrix and zero out values that are small enough to present numerical problems. Once this is done, the matrices can be used to solve the least-squares problem using a back substitution method similar to that used in LU decomposition. The code for SVD is available in various software packages and can be found in *Numerical Recipes* [16].

## B.2 Geometric Computations

A vector (a one-dimensional list of numbers) is often used to represent a point in space or a direction and magnitude in space (e.g., Figure B.2). A slight complication in terminology results because a direction and magnitude in space is also referred to as a *vector*. As a practical matter, this distinction is usually not important. A vector in space has no position, only magnitude and direction. For geometric computations, a matrix usually represents a transformation (e.g., Figure B.3).

### B.2.1 Components of a Vector

A vector, A, with coordinates ($A_x$, $A_y$, $A_z$) can be written as a sum of vectors, as shown in Equation B.16, in which *i, j, k* are unit vectors along the principal axes, *x, y,* and *z,* respectively.

$$A = A_x \cdot i + A_y \cdot j + A_z \cdot k \qquad \textbf{(Eq. B.16)}$$

### B.2.2 Length of a Vector

The *length* of a vector is computed as in Equation B.17. If a vector is of unit length, then $|A| = 1.0$, and it is said to be *normalized*. Dividing a vector by its length, forming a unit-length vector, is said to be *normalizing* the vector.

$$|A| = \sqrt{A_x^2 + A_y^2 + A_z^2} \qquad \textbf{(Eq. B.17)}$$

### B.2.3 Dot Product of Two Vectors

The *dot product*, or *inner product*, of two vectors is computed as in Equation B.18. The computation is commutative (Equation B.19) and associative (Equation B.20). The dot product of a vector with itself results in the square of its length

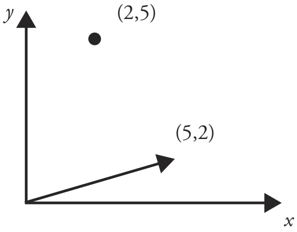


**Figure B.2** A point and vector in two-space

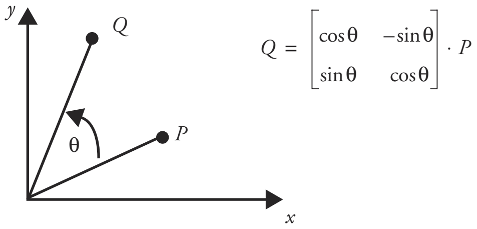

y
Q
θ
P
x
$Q = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot P$

**Figure B.3** A matrix representing a rotation

(Equation B.21). The dot product of two vectors, *A* and *B,* is equal to the lengths of the vectors times the cosine of the angle between them (Equation B.22, Figure B.4). As a result, the angle between two vectors can be determined by taking the arccosine of the dot product of the two normalized vectors (or, if they are not normalized, by taking the arccosine of the dot product divided by the lengths of the two vectors). The dot product of two vectors is equal to zero if the vectors are perpendicular to each other, as in Figure B.5 (or if one or both of the vectors are zero vectors). The dot product can also be used to compute the projection of one vector onto another vector (Figure B.6). This is useful in cases in which the coordinates of a vector are needed in an auxiliary coordinate system (Figure B.7).

$$A \bullet B = A_x \cdot B_x + A_y \cdot B_y + A_z \cdot B_z \qquad \text{(Eq. B.18)}$$

$$A \bullet B = B \bullet A \qquad \text{(Eq. B.19)}$$

$$(A \bullet B) \bullet C = A \bullet (B \bullet C) \qquad \text{(Eq. B.20)}$$

$$A \bullet A = |A|^2 \qquad \text{(Eq. B.21)}$$

$$A \bullet B = |A| \cdot |B| \cdot \cos\theta \qquad \text{(Eq. B.22)}$$

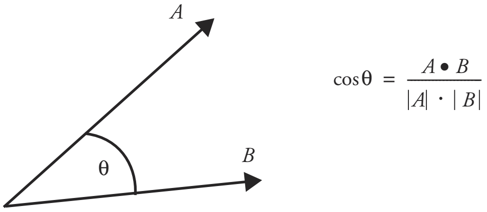
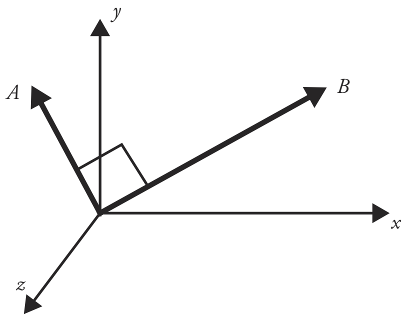
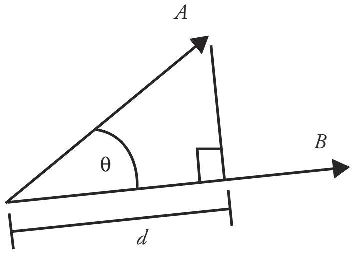


**Figure B.4** Using the dot product to compute the cosine of the angle between two vectors



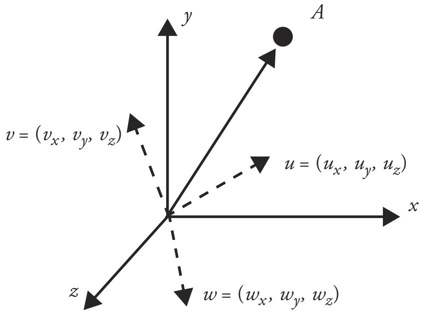
$$A \bullet B = 0$$

**Figure B.5** The dot product of perpendicular vectors



$$d = |A| \cdot \cos\theta = \frac{A \bullet B}{|B|}$$

**Figure B.6** Computing the length of the projection of vector *A* onto vector *B*



$$A_u = A \bullet u$$

$$A_v = A \bullet v$$

$$A_w = A \bullet w$$

$$A = (A_x, A_y, A_z) = A_u \cdot u + A_v \cdot v + A_w \cdot w$$

**Figure B.7** Computing the coordinates of a vector in an auxiliary coordinate system

### B.2.4 Cross Product of Two Vectors

The *cross product,* or *outer product,* of two vectors can be defined using the determinant of a 3x3 matrix as shown in Equation B.23, where *i, j,* and *k* are unit vectors in the directions of the principal axes. Equation B.24 shows the definition as an explicit equation. The cross product is not commutative (Equation B.25), but it is associative (Equation B.26).

$$A \times B = \begin{vmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} \quad \text{(Eq. B.23)}$$

$$A \times B = (A_y \cdot B_z - A_z \cdot B_y) \cdot i + (A_z \cdot B_x - A_x \cdot B_z) \cdot j + (A_x \cdot B_y - A_y \cdot B_x) \cdot k \quad \text{(Eq. B.24)}$$

$$A \times B = -(B \times A) = (-A) \times (-B) \quad \text{(Eq. B.25)}$$

$$A \times (B \times C) = (A \times B) \times C \quad \text{(Eq. B.26)}$$

The direction of $A \times B$ is perpendicular to both $A$ and $B$ (Figure B.8), and the direction is determined by the right-hand rule (if $A$ and $B$ are in right-hand space). If the thumb of the right hand is put in the direction of the first vector ($A$) and the index finger is put in the direction of the second vector ($B$), the cross product of the two vectors will be in the direction of the middle finger when it is held perpendicular to the first two fingers.

The magnitude of the cross product is the length of one vector times the length of the other vector times the sine of the angle between them (Equation ). A zero vector will result if the two vectors are colinear or if either vector is a zero vector (Equation B.28). This relationship is useful for determining the sine of the angle between two vectors (Figure B.9) and for computing the perpendicular distance from a point to a line (Figure B.10).

$$|A \times B| = |A| \cdot |B| \cdot \sin\theta$$
where $\theta$ is the angle from $A$ to $B$, $0 < \theta < 180$ (Eq. B.27)

$$|A \times B| = 0 \text{ if and only if } A \text{ and } B \text{ are colinear,}$$
i.e., $\sin\theta = \sin 0 = 0$, or $A = 0$ or $B = 0$ (Eq. B.28)

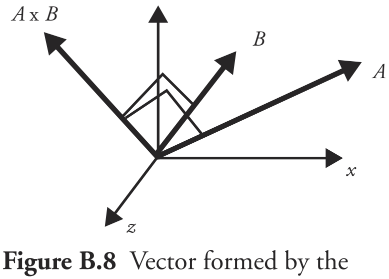


**Figure B.8** Vector formed by the cross product of two vectors

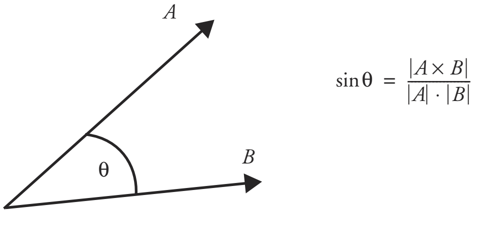
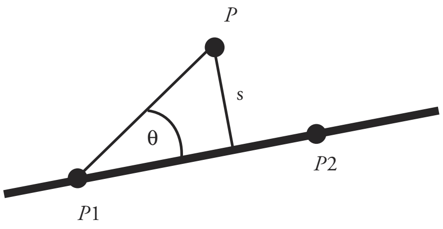


**Figure B.9** Using the cross product to compute the sine of the angle between two vectors



$$\begin{aligned} s &= |P - P1| \cdot \sin\theta \\ &= \frac{|(P - P1) \times (P2 - P1)|}{|P2 - P1|} \end{aligned}$$

**Figure B.10** Computing the perpendicular distance from a point to a line

## B.2.5 Vector and Matrix Routines

Simple vector and matrix routines are given here. These are used in some of the routines in this appendix.

### Vector Routines

Some vector routines can be implemented just as easily as in-line code using *#define.*

```
/* Vector.c */

typedef   struct xyz_struct {
float     x,y,z;
xyz_td;

}

/* ============================================================ */
/* compute the cross product of two vectors */
xyz_td crossProduct(xyz_td v1,xyz_td v2)
{
   xyz_tdp;

   p.x = v1.y*v2.z - v1.z*v2.y;
   p.y = v1.z*v2.x - v1.x*v2.z;
   p.z = v1.x*v2.y - v1.y*v2.x;
   return p;
}

/* ============================================================ */
/* compute the dot product of two vectors */
float dotProduct(xyz_td v1,xyz_td v2)
{
   return v1.x*v2.x+v1.y*v2.y+v1.z*v2.z;
}
```

```
/* ================================================================== */
/* normalize a vector */
void normalizeVector(xyz_td *v)
{
   float  len;

   len = sqrt(v->x*v->x + v->y*v->y + v->z*v->z);
   v->x /= len;
   v->y /= len;
   v->z /= len;
}

/* ================================================================== */
/* form the vector from the first point to the second */
xyz_td formVector(xyz_td p1, xyz_td p2)
{
   xyz_tdp;


   p.x = p2.x-p1.x;
   p.y = p2.y-p1.y;
   p.z = p2.z-p1.z;
   return p;
}

/* ================================================================== */
/* compute the length of a vector */
float length(xyz_td v)
{
   return sqrt(v.x*v.x+v.y*v.y+v.z*v.z);
}
```

### Matrix Routines

```
/* Matrix.c */

/* ================================================================== */
/* Matrix multiplication */
/* C x B = A */
void Matrix4x4MatrixMult (float **C,float **B,float **A)
{
   int  i,j;

   for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {
         A[i][j] = C[i][0]*B[0][j]+ C[i][1]*B[1][j]+
                   C[i][2]*B[2][j]+ C[i][3]*B[3][j];
      }
   }
}

/* =================================================================== */
/* matrix-vector multiplication */
/* N = M x V */
void Matrix4x4Vector4Mult (float **M,float *V,float *N)
{
   N[0] = M[0][0]*V[0]+M[0][1]*V[1]+M[0][2]*V[2]+M[0][3]*V[3];
   N[1] = M[1][0]*V[0]+M[1][1]*V[1]+M[1][2]*V[2]+M[1][3]*V[3];
   N[2] = M[2][0]*V[0]+M[2][1]*V[1]+M[2][2]*V[2]+M[2][3]*V[3];
   N[3] = M[3][0]*V[0]+M[3][1]*V[1]+M[3][2]*V[2]+M[3][3]*V[3];
}

/* =================================================================== */
/* vector-matrix multiplication */
/* N = V x M */
void Vector4Matrix4x4Mult (float *V,float **M,float *N)
{
   N[0] = M[0][0]*V[0]+M[1][0]*V[1]+M[2][0]*V[2]+M[3][0]*V[3];
   N[1] = M[0][1]*V[0]+M[1][1]*V[1]+M[2][1]*V[2]+M[3][1]*V[3];
   N[2] = M[0][2]*V[0]+M[1][2]*V[1]+M[2][2]*V[2]+M[3][2]*V[3];
   N[3] = M[0][3]*V[0]+M[1][3]*V[1]+M[2][3]*V[2]+M[3][3]*V[3];
}

/* =================================================================== */
/* compute the inverse of a matrix */
void ComputeInverse4x4(float **M,float **Minv)
{
   int      rowswaps[4];
   int      val;
   float    b[4];
   int      i,j;
   float**A;

   A = (float **)malloc(sizeof(float *)*4);
   for (i=0; i<4; i++) {
      A[i] = (float *)malloc(sizeof(float)*4);
   }
   for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {
         A[i][j] = M[i][j];
      }
   }

   LUdecomp(A,4,rowswaps,&val);

   for (i=0; i<4; i++) {
      for (j=0; j<4; j++) b[j] = (i==j) ? 1:0;
      LUsubstitute(A,4,rowswaps,b);
      for (j=0; j<4; j++) Minv[j][i] = b[j];
   }
}
```

## B.2.6 Closest Point between Two Lines in Three-Space

The intersection of two lines in three-space often needs to be calculated. Because of numerical imprecision, the lines rarely, if ever, actually intersect in three-space. As a result, the computation that needs to be performed is to find the two points, one from each line, at which the lines are closest to each other. The points *P*1 and *P*2 at which the lines are closest form a line segment perpendicular to both lines (Figure B.11). They can be represented parametrically as points along the lines, and then the parametric interpolants can be solved for by satisfying the equations that state the requirement for perpendicularity (Equation B.29).

$$
\begin{aligned}
&(P2 - P1) \bullet V = 0 \\
&(P2 - P1) \bullet W = 0 \\
&(B + t \cdot W - (A + s \cdot V)) \bullet V = 0 \\
&(B + t \cdot W - (A + s \cdot V)) \bullet W = 0 \\
&t = \frac{-B \bullet V + A \bullet V + s \cdot V \bullet V}{W \bullet V} \\
&t = \frac{-B \bullet W + A \bullet W + s \cdot V \bullet W}{W \bullet W} \\
&\frac{-B \bullet V + A \bullet V + s \cdot V \bullet V}{W \bullet V} = \frac{-B \bullet W + A \bullet W + s \cdot V \bullet W}{W \bullet W} \\
&(-B \bullet V + A \bullet V + s \cdot V \bullet V) \cdot (W \bullet W) = (-B \bullet W + A \bullet W + s \cdot V \bullet W) \cdot (W \bullet V) \\
&s = \frac{(A \bullet W - B \bullet W) \cdot (W \bullet V) + (B \bullet V - A \bullet V) \cdot (W \bullet W)}{(V \bullet V) \cdot (W \bullet W) - (V \bullet W)^2} \\
&t = \frac{(B \bullet V - A \bullet V) \cdot (W \bullet V) + (A \bullet W - B \bullet W) \cdot (V \bullet V)}{(V \bullet V) \cdot (W \bullet W) - (V \bullet W)^2}
\end{aligned}
\tag*{(Eq. B.29)}
$$

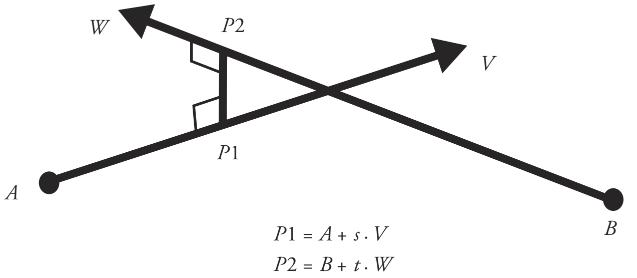


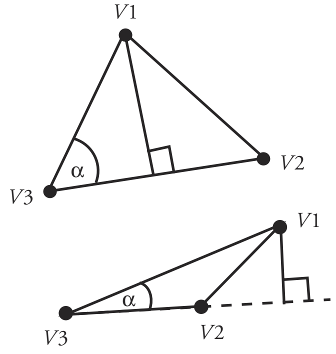
**Figure B.11** Two lines are closest to each other at points *P*1 and *P*2

## B.2.7 Area Calculations

### Area of a Triangle

The area of a triangle consisting of vertices *V*1, *V*2, *V*3 is one-half times the length of one edge times the perpendicular distance from that edge to the other vertex. The perpendicular distance from the edge to a vertex can be computed using the cross product. See Figure B.12. For triangles in 2D, the *z*-coordinates are essentially considered zero and the cross product computation is simplified accordingly (only the *z*-coordinate of the cross product is nonzero).

The signed area of the triangle is required for computing the area of a polygon (see below). In the 2D case, this is done simply by not taking the absolute value of the *z*-coordinate of the cross product. In the 3D case, a vector normal to the polygon can be used to indicate the positive direction. The direction of the vector produced by the cross product can be compared to the normal (using the dot product) to determine whether it is in the positive or negative direction. The length of the cross product vector can then be computed and the appropriate sign applied to it.



$$\begin{aligned}\text{Area}(V1,\ V2,\ V3) &= \frac{1}{2}\cdot|V2-V3|\cdot|V1-V3|\cdot\sin\alpha\\ &= \frac{1}{2}\cdot|V2-V3|\cdot\frac{|(V1-V3)\times(V2-V3)|}{|V2-V3|}\\ &= \frac{1}{2}\cdot|(V1-V3)\times(V2-V3)|\end{aligned}$$

**Figure B.12** Area of a triangle

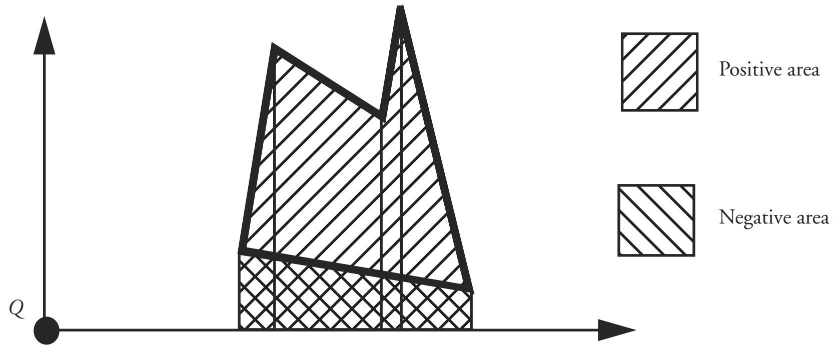


**Figure B.13** Computing the area of a polygon

### Area of a Polygon

The area of a polygon can be computed as a sum of the signed areas of simple elements. In the 2D case, the signed area under each edge of the polygon can be summed to form the area (Figure B.13). The area under an edge is the average height of the edge times its width (Equation B.30, where subscripts are computed modulo $n + 1$).

$$\begin{aligned}\text{Area} &= \sum_{i=1}^{n} \frac{(y_i + y_{i+1})}{2} \cdot (x_{i+1} - x_i) \\ &= \frac{1}{2} \cdot \sum_{i=1}^{n} (y_i \cdot x_{i+1} - y_{i+1} \cdot x_i) \qquad \textbf{(Eq. B.30)}\end{aligned}$$

The area of a polygon can also be computed by using each edge of the polygon to construct a triangle with the origin (Figure B.14). The signed area of the triangle must be used so that edges directed clockwise with respect to the origin cancel out edges directed counterclockwise with respect to the origin. Although this is more computationally expensive than summing the areas under the edges, it suggests a way to compute the area of a polygon in three-space. In the 3D case, one of the vertices of the polygon can be used to construct a triangle with each edge, and the 3D version of the vector equations of Figure B.12 can be used.

## B.2.8 The Cosine Rule

The cosine rule states the relationship between the lengths of the edges of a triangle and the cosine of an interior angle (Figure B.15). It is useful for determining the interior angle of a triangle when the locations of the vertices are known.

$$\begin{aligned}\text{Area of Polygon}(A, B, C, D, E) = {} & \text{Area of Triangle}(Q, A) \\ & + \text{Area of Triangle}(Q, B) \\ & + \text{Area of Triangle}(Q, C) \\ & + \text{Area of Triangle}(Q, D) \\ & + \text{Area of Triangle}(Q, E)\end{aligned}$$

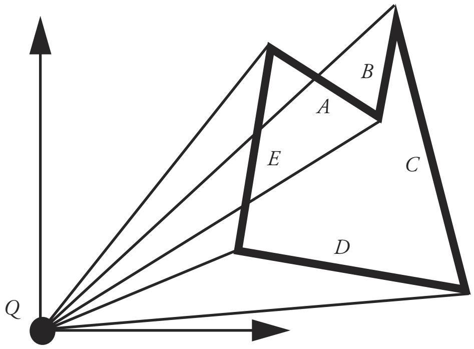
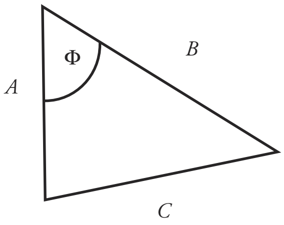


**Figure B.14** The area of a two-dimensional polygon; the edges are labeled with letters, triangles are constructed from each edge to the origin, and the areas of the triangles are signed according to the direction of the edge with respect to the origin



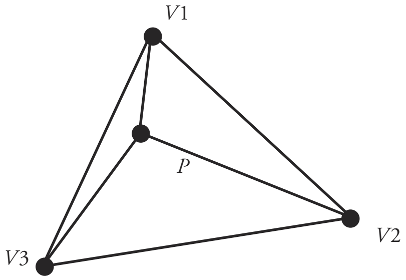
$$|C|^2 = |A|^2 + |B|^2 - 2 \cdot |A| \cdot |B| \cdot \cos\Phi$$

$$\cos\Phi = \frac{|A|^2 + |B|^2 - |C|^2}{2 \cdot |A| \cdot |B|}$$

**Figure B.15** The cosine rule

## B.2.9 Barycentric Coordinates

Barycentric coordinates are the coordinates of a point in terms of weights associated with other points. Most commonly used are the barycentric coordinates of a point with respect to vertices of a triangle. The barycentric coordinates ($u1$, $u2$, $u3$) of a point, *P*, with respect to a triangle with vertices $V1$, $V2$, $V3$ are shown in Figure B.16. Notice that for a point inside the triangle, the coordinates always sum to one. This can be extended easily to any convex polygon by a direct generalization of the equations. However, it cannot be extended to concave polygons.



$$u1 = \frac{\text{Area}(P, V2, V3)}{\text{Area}(V1, V2, V3)}$$

$$u2 = \frac{\text{Area}(P, V3, V1)}{\text{Area}(V1, V2, V3)}$$

$$u3 = \frac{\text{Area}(P, V1, V2)}{\text{Area}(V1, V2, V3)}$$

$$P = u1 \cdot V1 + u2 \cdot V2 + u3 \cdot V3$$

**Figure B.16** The barycentric coordinates of a point with respect to vertices of a triangle

## B.2.10 Computing Bounding Shapes

Bounding volumes are useful as approximate extents of more complex objects. Often, simpler tests can be used to determine the general position of an object by using bounding volumes, thus saving computation. In computer animation, the most obvious example occurs in testing for object collisions. If the bounding volumes of objects are not overlapping, then the objects themselves must not be penetrating each other. Planar polyhedra are considered here because the bounding volumes can be determined by inspecting the vertices of the polyhedra. Nonplanar objects require more sophisticated techniques. *Axis-aligned bounding boxes* (AABBs) and bounding spheres are relatively easy to calculate but can be poor approximations of the object's shape. Slabs and *oriented bounding boxes* (OBBs) can provide a much better fit. OBBs are rectangular bounding boxes at an arbitrary orientation [7]. For collision detection, bounding shapes are often hierarchically organized into tighter and tighter approximations of the object's space. A *convex hull*, the smallest convex shape bounding the object, provides an even tighter approximation but requires more computation.

### Bounding Boxes

*Bounding box* typically refers to a boundary cuboid (or rectangular solid) whose sides are aligned with the principal axes. A bounding box for a collection of points is easily computed by searching for minimum and maximum values for the *x*-, *y*-, and *z*-coordinates. A point is inside the bounding box if its coordinates are between min/max values for each of the three coordinate pairs. While the bounding box may be a good fit for some objects, it may not be a good fit for others (Figure B.17). How well the bounding box approximates the shape of the object is rotationally variant, as shown in Figures B.17b and B.17c.

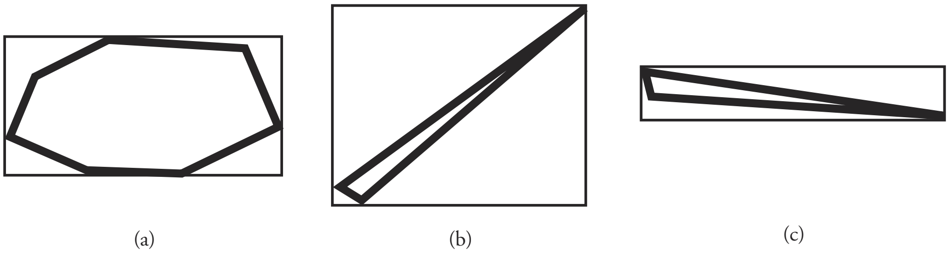


**Figure B.17** Sample objects and their bounding boxes in 2D

### Bounding Slabs

Bounding slabs are a generalization of bounding boxes. A pair of arbitrarily oriented planes are used to bound the object. The orientation of the pair of planes is specified by a user-supplied normal vector.

The normal defines a family of planes that vary according to perpendicular distance to the origin. In the planar equation $a \cdot x + b \cdot y + c \cdot y = d$, $(a, b, c)$ represents a vector normal to the plane. If this vector has unit length, then $d$ is the perpendicular distance to the plane. If the length of $(a, b, c)$ is not one, then $d$ is the perpendicular distance scaled by the vector's length. Notice that $d$ is equal to the dot product of the vector $(a, b, c)$ and a point on the plane.

Given a user-supplied normal vector, the user computes the dot product of that vector and each vertex of the object and records the minimum and maximum values (Equation B.31). The normal vector and these min/max values define the bounding slab. See Figure B.18 for a 2D diagram illustrating the idea. Multiple slabs can be used to form an arbitrarily tight bounding volume of the convex hull of the polyhedron. A point is inside this bounding volume if the result of the dot

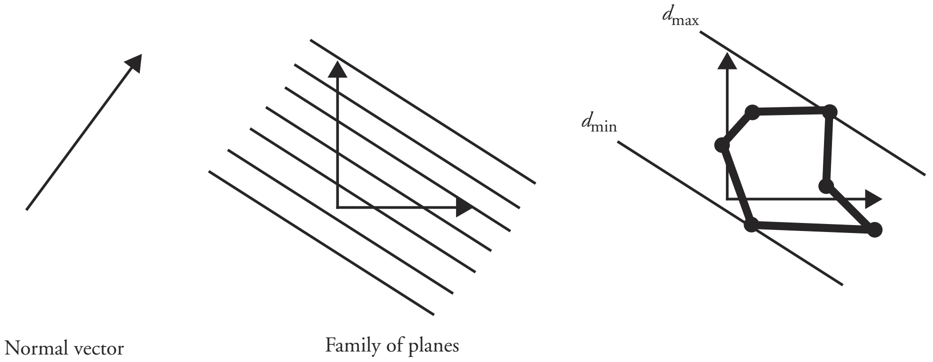
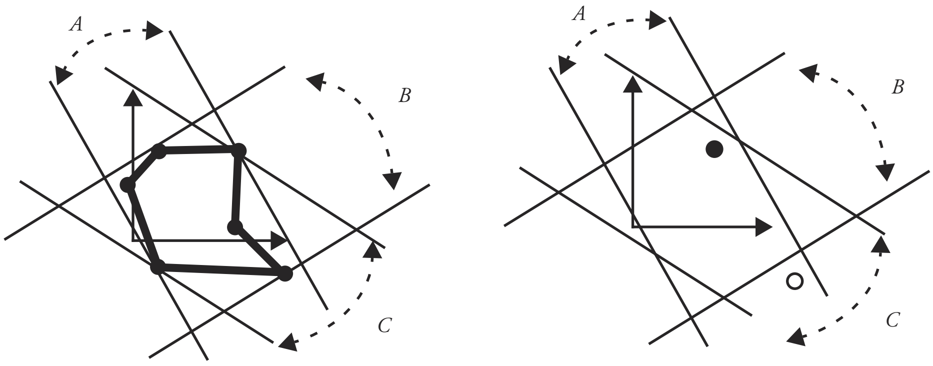


**Figure B.18** Computing a boundary slab for a polyhedron



Three bounding slabs: *A, B, C*

● Inside of all slabs

○ Outside of slab *B*

**Figure B.19** Multiple bounding slabs

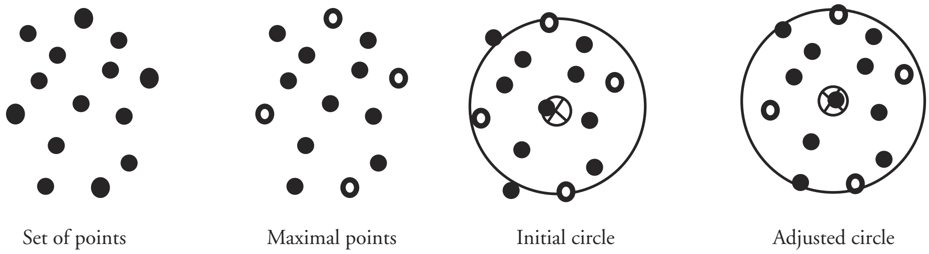
product of it and the normal vector is between the corresponding min/max values for each slab (see Figure B.19 for an example in 2D).

$$P = (x, y, z) \quad \text{vertex}$$
$$N = (a, b, c) \quad \text{normal}$$
$$P \cdot N = d \quad \text{computing the planar equation constant} \qquad \textbf{(Eq. B.31)}$$

### Bounding Sphere

Computing the optimal bounding sphere for a set of points can be expensive. However, more tractable approximate methods exist. A quick and fairly accurate method of computing an approximate bounding sphere for a collection of points is to make an initial guess at the bounding sphere and then incrementally enlarge the sphere as necessary by inspecting each point in the set. The description here follows Ritter [17].

The first step is to loop through the points and record the minimum and maximum points in each of the three principal directions. The second step is to use the maximally separated pair of points from the three pairs of recorded points and create an initial approximation of the bounding sphere. The third step is, for each point in the original set, to adjust the bounding sphere as necessary to include the point. Once all the points have been processed, a near-optimal bounding sphere has been computed (Figure B.20). This method is fast and it is easy to implement.


Set of points

Maximal points

Initial circle

Adjusted circle

**Figure B.20** Computing a bounding circle for a set of points

```
/*
** Bounding Sphere Computation
*/
void boundingSphere(xyz_td *pnts,int n, xyz_td *cntr, float *radius)
{
   int     i,minxi,maxxi,minyi,maxyi,minzi,maxzi,p1i,p2i;
   float   minx,maxx,miny,maxy,minz,maxz;
   float   diam2,diam2x,diam2y,diam2z,rad,rad2;
   float   dx,dy,dz;
   float   cntrx,cntry,cntrz;
   float   delta;
   float   dist,dist2;
   float   newrad,newrad2;
   float   newcntrx,newcntry,newcntrz;

   /* step one: find minimal and maximal  points in each of 3 principal
   directions */
   minxi = 0; minx = pnts[0].x;  maxxi = 0; maxx = pnts[0].x;
   minyi = 0; miny = pnts[0].y;  maxyi = 0; maxy = pnts[0].y;
   minzi = 0; minz = pnts[0].z;  maxzi = 0; maxz = pnts[0].z;
   for (i=1; i<n; i++) {
      if (pnts[i].x < minx) { minx = pnts[i].x; minxi = i; }
      if (pnts[i].x > maxx) { maxx = pnts[i].x; maxxi = i; }
      if (pnts[i].y < miny) { miny = pnts[i].y; minyi = i; }
      if (pnts[i].y > maxy) { maxy = pnts[i].y; maxyi = i; }
      if (pnts[i].z < minz) { minz = pnts[i].z; minzi = i; }
      if (pnts[i].z > maxz) { maxz = pnts[i].z; maxzi = i; }
   }

   /* step two: find maximally separated points from the 3 pairs;  use
   to initialize sphere */
```

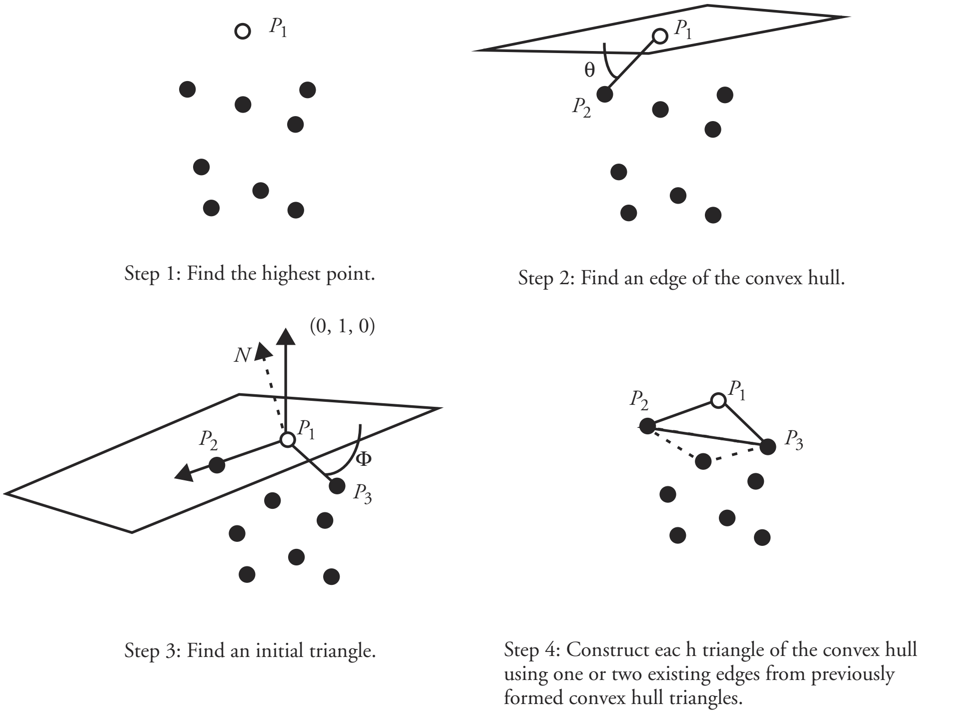
```
/* find maximally separated points  by comparing the distance squared
between points */
dx = pnts[minxi].x - pnts[maxxi].x;
dy = pnts[minxi].y - pnts[maxxi].y;
dz = pnts[minxi].z - pnts[maxxi].z;
diam2x = dx*dx + dy*dy + dz*dz;
dx = pnts[minyi].x - pnts[maxyi].x;
dy = pnts[minyi].y - pnts[maxyi].y;
dz = pnts[minyi].z - pnts[maxyi].z;
diam2y = dx*dx + dy*dy + dz*dz;
dx = pnts[minzi].x - pnts[maxzi].x;
dy = pnts[minzi].y - pnts[maxzi].y;
dz = pnts[minzi].z - pnts[maxzi].z;
diam2z = dx*dx + dy*dy + dz*dz;
diam2 = diam2x; p1i = minxi; p2i = maxxi;
if (diam2y>diam2) { diam2 = diam2y; p1i=minyi; p2i=maxyi; }
if (diam2z>diam2) { diam2 = diam2z; p1i=minzi; p2i=maxzi;}
/* center  of initial sphere is average of two points */
cntrx = (pnts[p1i].x+pnts[p2i].x)/2;
cntry = (pnts[p1i].y+pnts[p2i].y)/2;
cntrz = (pnts[p1i].z+pnts[p2i].z)/2;
/* calculate radius and radius squared of initial sphere - from
diameter squared*/
rad2 = diam2/4;
rad = sqrt(rad2);
printf("maximally separated pair: (%f,%f,%f):(%f,%f,%f),%f\n",
   pnts[p1i].x,pnts[p1i].y,pnts[p1i].z,
   pnts[p2i].x,pnts[p2i].y,pnts[p2i].z,diam2);
printf("initial center: (%f,%f,%f)\n",cntrx,cntry,cntrz);
printf("initial diam2: %f\n",diam2);
printf("initial radius, radius2 = %f,%f\n",rad,rad2);

/* third step: now step through the set of points and adjust bounding
sphere as necessary */
for (i=0; i<n; i++) {
   dx = pnts[i].x - cntrx;
   dy = pnts[i].y - cntry;
   dz = pnts[i].z - cntrz;
   dist2 = dx*dx + dy*dy + dz*dz;  /* distance squared of old
   center to pnt */
   if (dist2 > rad2) {             /* need to update sphere if this
   point is outside  old radius*/
      dist = sqrt(dist2);
```

```
         /* new radius is average of current radius and distance from
         center to pnt */
         newrad = (rad + dist)/2;
         newrad2 = newrad*newrad;
         printf("new radius = %f\n",newrad);
         delta = dist - newrad;/* distance from old center to new
         center */
         /* delta/dist and rad/dist are weights of pnt and old center to
         compute new center */
         newcntrx = (newrad*cntrx+delta*pnts[i].x)/dist;
         newcntry = (newrad*cntry+delta*pnts[i].y)/dist;
         newcntrz = (newrad*cntrz+delta*pnts[i].z)/dist;

         /* test to see if new radius and center contain the point */
         /* this test should only fail by an epsilon due to numeric
         imprecision */
         dx = pnts[i].x - newcntrx;
         dy = pnts[i].y - newcntry;
         dz = pnts[i].z - newcntrz;
         dist2 = dx*dx + dy*dy + dz*dz;
         if (dist2 > newrad2) {
            printf("ERROR by %lf\n",((double)(dist2))-newrad2);
            printf("   center - radius: (%f,%f,%f) -
            %f\n",cntrx,cntry,cntrz,rad);
            printf("  New center - radius: (%f,%f,%f) - %f\n",
               newcntrx,newcntry,newcntrz,newrad);

         }
         cntrx = newcntrx;
         cntry = newcntry;
         cntrz = newcntrz;
         rad = newrad;
         rad2 = rad*rad;

      }
   }

   *radius = rad;
   cntr->x = cntrx;
   cntr->y = cntry;
   cntr->z = cntrz;
   return;
}
```

## Convex Hull

The convex hull of a set of points is the smallest convex polyhedron that contains all the points. A simple algorithm for computing the complex hull is given here, although more efficient techniques exist. Since this is usually a onetime code for an object (unless the object is deforming), this is a case where efficiency can be traded for ease of implementation. Refer to Figure B.21.

1. Find a point on the convex hull by finding the point with the largest $y$-coordinate. Refer to the point found in this step as $P_1$.
2. Construct an edge on the convex hull by using the following method. Find the point that, when connected with $P_1$, makes the smallest angle with the horizontal plane passing through $P_1$. Use $L$ to refer to the line from $P_1$ to the point. Finding the smallest sine of the angle is equivalent to finding the smallest angle. The sine of the angle between $L$ and the horizontal plane passing through $P_1$ is equal to the cosine of the angle between $L$ and the vector (0, –1, 0). The dot product of these two vectors is used to compute the cosine of the angle between them. Refer to the point found in this step as $P_2$, and refer to the line from $P_1$ to $P_2$, as $L$.



Step 1: Find the highest point.

Step 2: Find an edge of the convex hull.

Step 3: Find an initial triangle.

Step 4: Construct eac h triangle of the convex hull using one or two existing edges from previously formed convex hull triangles.

**Figure B.21** Computing the convex hull

3. Construct a triangle on the convex hull by the following method. First, construct the plane defined by $L$ and a horizontal line perpendicular to $L$ at $P_1$. The horizontal line is constructed according to ($L \times (0, -1, 0)$). All of the points are below this plane. Find the point that, when connected with $P_1$, makes the smallest angle with this plane. Use $K$ to refer to the line from $P_1$ to the point. The sine of the angle between $K$ and the plane is equal to the cosine of the angle between $K$ and the downward-pointing normal vector of the plane. This normal can be computed as $N = (L \times (0, -1, 0)) \times L$ (in right-hand space). Refer to the point found in this step as $P_3$. The triangle on the convex hull is defined by these three points. A consistent ordering of the points should be used so that they, for example, compute an outward-pointing normal vector ($N = (P_3 - P_1) \times (P_2 - P_1)$, $N_y > 0.0$) in right-hand space. Initialize the list of convex hull triangles with this triangle and its outward-pointing normal vector. Mark each of its three edges as *unmatched* to indicate that the triangle that shares the edge has not been found yet.
4. Search the current list of convex hull triangles and find an *unmatched* edge. Construct the triangle of the convex hull that shares this edge by the following method. Find the point that, when connected by a line from a point on the edge, makes the smallest angle with the plane of the triangle while creating a dihedral angle (interior angle between two faces measured at a shared edge) greater than 90 degrees. The dihedral angle can be computed using the angle between the normals of the two triangles. When the point has been found, add the triangle defined by this point and the marked edge to the list of convex hull triangles and the unmarked edge. The rest of the unmarked edges in the list must be searched to see if the two other edges of the new triangle already occur in the list. If either of the new edges does not have a match in the list, then it should be marked as unmatched. Otherwise, mark the edge as *matched*. Now go back through the list of convex hull triangles and look for another *unmatched* edge and repeat the procedure to construct a new triangle that shares that edge. When there are no more *unmatched* edges in the list of convex hull triangles, the hull has been constructed.

Step 4 in the algorithm above does not handle the case in which there are more than three coplanar vertices. To handle these cases, instead of forming a triangle, form a convex hull polygon for the coplanar points. This is done similarly in two and three dimensions. First, collect all of the coplanar points (all of the points within some epsilon of being coplanar) into a set. Second, initialize the current edge to be the unmatched edge of step 4 and add the first point of the current edge to the set of coplanar points. Third, iteratively find the point in the set of coplanar points that makes the smallest angle with the current edge as measured from the first point of the edge to the second point of the edge to the candidate point. When the point is found, remove it from the set of coplanar points and make the new current edge the edge from the second point of the old current edge to the newly found point. Continue iterating until the first point of the original unmatched edge is found. This will complete the convex hull polygon, which is then added to the list of convex hull polygons; its newly formed edges are processed as in step 4.

```
/* ConvexHull.c
 * This code uses a brute force algorithm to construct the convex hull
 * and does not handle more than three coplanar points
 */

#include "Vector.h"

typedef struct conHullTri_struct {
   int      pi[3];
   int      matched[3];
   xyz_td   normal;
   struct conHullTri_struct *next;
} conHullTri_td;

conHullTri_td *chtList;

/* ================================================================ */
/* CONVEX HULL */
int ConvexHull(xyz_td *pntList,int num,int **triangleList,int
*numTriangles)
{
   int      i;
   int      p1i,p2i,p3i,pi;
   xyz_td   yaxis;
   xyz_td   v,n,nn,nnn,v1,v2;
   float    t,t1,t2;
   int      count;
   conHullTri_td *chtPtr,*chtPtrTail,*chtPtrNew,*chtPtrA;
   int      done;
   int      *triList;
   int      dummy,notError;

   yaxis.x = 0; yaxis.y = 1; yaxis.z = 0;

   /* find the highest point */
   p1i = 0; t = pntList[0].y;
   for (i=1; i<num; i++) {
      if (pntList[i].y > t) { p1i=i; t=pntList[i].y;}
   }
```

```
/* find point that makes minimum angle with horizontal plane */
p2i = (p1i==0) ? 1:0;
v = formVector(pntList[p1i],pntList[p2i]);
normalizeVector(&v);
t = v.y;
if (t>0.0) {
   printf(" ERROR - found higher point\n");
   scanf("%d",&dummy);
   return 1;
}
for (i=p2i+1; i<num; i++) {
   if (i!=p1i) {
      v = formVector(pntList[p1i],pntList[i]);
      normalizeVector(&v);
      t1 = v.y;
      if (t1 > t) {p2i = i; t = t1;}
   }
}

/* find point that makes triangle with minimum angle with horizontal
plane through edge */
v1 = formVector(pntList[p1i],pntList[p2i]);

if ((p1i!=0) && (p2i!=0)) p3i=0;
else if ((p1i!=1) && (p2i!=1)) p3i=1;
else p3i=2;
v2 = formVector(pntList[p2i],pntList[i]);
n = crossProduct(v2,v1);
normalizeVector(&n);
if (n.y < 0) {
   n.x = -n.x; n.y = -n.y; n.z = -n.z;
}
for (i=p3i+1; i<num; i++) {
   if ((i!=p1i)&&(i!=p2i)) {
      v = formVector(pntList[p2i],pntList[i]);
      nn = crossProduct(v1,v);
      normalizeVector(&nn);
      if (nn.y < 0) { nn.x = -nn.x; nn.y = -nn.y; nn.z = -nn.z; }
      if (nn.y>n.y) {
         p3i=i;
         n.x = nn.x; n.y = nn.y; n.z = n.z;
      }
   }
}

/* compute outward-pointing normal vector in right-hand space for
clockwise triangle */
/* recalculate the normal vector */
v1 = formVector(pntList[p1i],pntList[p2i]);
v2 = formVector(pntList[p2i],pntList[p3i]);
n = crossProduct(v2,v1);
normalizeVector(&n);
if (n.y<0) {
   n.x = - n.x; n.y = -n.y; n.z = -n.z;
   pi = p1i; p1i = p2i; p2i = pi;
}

/* make a convex hull entry */
count = 1;
chtPtr = (conHullTri_td *)malloc(sizeof(conHullTri_td));
if (chtPtr == NULL) {
   printf(" unsuccessful memory allocation 1\n");
   scanf("%d",&dummy);
   return 1;
}
chtPtr->pi[0] = p1i;
chtPtr->pi[1] = p2i;
chtPtr->pi[2] = p3i;
chtPtr->matched[0] = FALSE;
chtPtr->matched[1] = FALSE;
chtPtr->matched[2] = FALSE;
chtPtr->normal = n;

/* initialize the convex hull triangle list with the triangle */
chtList = chtPtr;
chtPtr->next = NULL;
chtPtrTail = chtPtr;

/* check and make sure all vertices are 'underneath' the initial
triangle */
for (i=0; i<num; i++) {
   if ((i!=chtPtr->pi[0]) &&
      (i!=chtPtr->pi[1]) &&
      (i!=chtPtr->pi[2])    ) {
      v = formVector(pntList[chtPtr->pi[0]],pntList[i]);
      t = dotProduct(v,n);
      if (t>0.0) {
            /* ERROR - point above initial triangle */
            printf(" ERROR - found a point above initial triangle
            (%d)\n",i);
            return 1;
      }
  }
}
/* now loop through the convex hull triangle list and process
unmatched edges */
done = FALSE;
chtPtr = chtList;
while (chtPtr!=NULL) {
      /* look for first unmatched edge */
      if ( (!(chtPtr->matched[0])) ||
         (!(chtPtr->matched[1])) ||
         (!(chtPtr->matched[2]))   ) {

         /* set it now as matched, and record 3 points with unmatched as
         first two */
         if (!(chtPtr->matched[0])) {
            p1i=chtPtr->pi[0]; p2i=chtPtr->pi[1]; p3i=chtPtr->pi[2];
            chtPtr->matched[0]  = TRUE;
         }
         else if (!(chtPtr->matched[1])) {
            p1i=chtPtr->pi[1]; p2i=chtPtr->pi[2]; p3i=chtPtr->pi[0];
            chtPtr->matched[1]  = TRUE;
         }
         else if (!(chtPtr->matched[2])) {
            p1i=chtPtr->pi[2]; p2i=chtPtr->pi[0]; p3i=chtPtr->pi[1];
            chtPtr->matched[2]  = TRUE;
         }
         /* get info of triangle of unmatched edge */
         n.x = chtPtr->normal.x;
         n.y = chtPtr->normal.y;
         n.z = chtPtr->normal.z;
         v1=formVector(pntList[p2i],pntList[p1i]);
         /* find new vertex which, with unmatched edge, makes
         triangle */
         /* whose normal is closest to normal of triangle of unmatched
         edge */
```

```
pi = -1;
for (i=0; i<num; i++) {
   if ((i!=p1i)&&(i!=p2i)&&(i!=p3i)) {
      v=formVector(pntList[p1i],pntList[i]);
      /* test to see if point is above triangle */
      t1 = dotProduct(v,n);
      if (t1>0) {
         /* ERROR - point above initial triangle */
         printf(" ERROR - found a point above initial triangle
         (%d)\n",i);
         return 1;
      }
      /* compute normal of proposed new triangle */
      nn = crossProduct(v,v1);
      normalizeVector(&nn);
      /* test for concave corner */
      nnn = crossProduct(n,nn);
      t2 = dotProduct(nnn,v1);
      if (t2<0.0) {
         printf(" ERROR - concave corner found\n");
         return 1;
      }
      /* compute angle made by faces (=angle made by normals)
      */
      t1 = dotProduct(n,nn);
      /* printf(" %d: dot product of normals: %f\n",i,t1); */
      /* printf(" normal for comparison: %f %f
      %f\n",n.x,n.y,n.z); */
      /* printf("     normal: %f %f %f\n",nn.x,nn.y,nn.z); */
      /* save smallest angle (largest cosine) */
      if (pi==-1) {pi=i; t=t1;}
      else if (t1>t) {pi=i; t=t1;}
   }
}

/* check and make sure all vertices are 'underneath' this
triangle */
v=formVector(pntList[p1i],pntList[pi]);
nn = crossProduct(v,v1);
normalizeVector(&nn);
for (i=0; i<num; i++) {
   if ((i!=p2i) &&
      (i!=p1i) &&
      (i!=pi)    ) {
      v = formVector(pntList[p1i],pntList[i]);
      t = dotProduct(v,nn);
      if (t>0.0) {
         /* ERROR - point above new triangle */
         printf(" ERROR - found a point above new triangle
         (%d)\n",i);
         return 1;
       }
   }
}

/* search for p2i-pi or pi-pi1 already in database - error
condition */
chtPtrA = chtList; notError = TRUE;
while ((chtPtrA!=NULL)&&notError) {
   if ((chtPtrA->pi[0]==p1i)&&(chtPtrA->pi[1]==pi)) notError =
   FALSE;
   else if ((chtPtrA->pi[1]==p1i)&&(chtPtrA->pi[2]==pi))
   notError = FALSE;
   else if ((chtPtrA->pi[2]==p1i)&&(chtPtrA->pi[0]==pi))
   notError = FALSE;
   else if ((chtPtrA->pi[0]==pi)&&(chtPtrA->pi[1]==p2i))
   notError = FALSE;
   else if ((chtPtrA->pi[1]==pi)&&(chtPtrA->pi[2]==p2i))
   notError = FALSE;
   else if ((chtPtrA->pi[2]==pi)&&(chtPtrA->pi[0]==p2i))
   notError = FALSE;
   else chtPtrA = chtPtrA->next;
   /* end while */
if (!notError) {
   printf(" ERROR - duplicating edge
   (%d,%d,%d)\n",p1i,p2i,pi);
   return 1;
}

/* add p1i, p2i, pi */
count++;
chtPtrNew = (conHullTri_td *)malloc(sizeof(conHullTri_td));
if (chtPtrNew == NULL) {
   printf(" unsuccessful memory allocation 2\n");
   return 1;
}

chtPtrTail->next = chtPtrNew;
chtPtrNew->pi[0] = p2i;
chtPtrNew->pi[1] = p1i;
chtPtrNew->pi[2] = pi;
chtPtrNew->matched[0] = TRUE;
chtPtrNew->matched[1] = FALSE;
chtPtrNew->matched[2] = FALSE;
chtPtrNew->normal.x = nn.x;
chtPtrNew->normal.y = nn.y;
chtPtrNew->normal.z = nn.z;
chtPtrNew->next = NULL;
chtPtrTail = chtPtrNew;

/* search for p2i-pi or pi-p1i already in database in reverse
order */
chtPtrA = chtList;
while (chtPtrA!=NULL) {
   if (!chtPtrA->matched[0]&&(chtPtrA->pi[0]==pi)&&(chtPtrA-
   >pi[1]==p1i)) {
      chtPtrA->matched[0] = TRUE;
      chtPtrNew->matched[1] = TRUE;
   }
   else if (!chtPtrA->matched[1]&&(chtPtrA-
   >pi[1]==pi)&&(chtPtrA->pi[2]==p1i)) {
      chtPtrA->matched[1] = TRUE;
      chtPtrNew->matched[1] = TRUE;
   }
   else if (!chtPtrA->matched[2]&&(chtPtrA-
   >pi[2]==pi)&&(chtPtrA->pi[0]==p1i)) {
      chtPtrA->matched[2] = TRUE;
      chtPtrNew->matched[1] = TRUE;
   }
   else if (!chtPtrA->matched[0]&&(chtPtrA-
   >pi[0]==p2i)&&(chtPtrA->pi[1]==pi)) {
      chtPtrA->matched[0] = TRUE;
      chtPtrNew->matched[2] = TRUE;
   }
   else if (!chtPtrA->matched[1]&&(chtPtrA-
   >pi[1]==p2i)&&(chtPtrA->pi[2]==pi)) {
```

```
               chtPtrA->matched[1] = TRUE;
               chtPtrNew->matched[2] = TRUE;
            }
            else if (!chtPtrA->matched[2]&&(chtPtrA-
            >pi[2]==p2i)&&(chtPtrA->pi[0]==pi)) {
               chtPtrA->matched[2] = TRUE;
               vchtPtrNew->matched[2] = TRUE;
            }
            else {
               chtPtrA = chtPtrA->next;
            }
         } /* end while */

      } /* end endif */
      else {
         chtPtr = chtPtr->next;
      }
   }

   triList = (int *)malloc(sizeof(int)*count*3);
   chtPtr = chtList;
   for (i=0; i<count; i++) {
      if (chtPtr==NULL) {
         printf(" ERROR: count %d doesn't match data
         structure\n",count);
         return 1;
      }
      triList[3*i] = chtPtr->pi[0];
      triList[3*i+1] = chtPtr->pi[1];
      triList[3*i+2] = chtPtr->pi[2];
      chtPtr=chtPtr->next;
   }
   *numTriangles = count;
   *triangleList = triList;
   return 0;
}

void printCHTlist(conHullTri_td *chtPtr)
{
   printf(" CHT list\n");
   while (chtPtr!=NULL) {
      printf(" %d:%d:%d ; %d:%d:%d ; %f,%f,%f\n",
         chtPtr->pi[0],chtPtr->pi[1],chtPtr->pi[2],
         chtPtr->matched[0],chtPtr->matched[1],chtPtr->matched[2],
         chtPtr->normal.x,chtPtr->normal.y,chtPtr->normal.z);
      chtPtr = chtPtr->next;
   }
}
```

## B.3 Transformations

### B.3.1 Transforming a Point Using Vector-Matrix Multiplication

Vector-matrix multiplication is usually how the transformation of a point is represented. Because a vector is just an *N*x1 matrix, vector-matrix multiplication is actually a special case of matrix-matrix multiplication. Vector-matrix multiplication is usually performed by premultiplying a column vector by a matrix. This is equivalent to postmultiplying a row vector by the transpose of that same matrix. Both notations are encountered in the graphics literature, but use of the column vector is more common. The example in Equation  uses a 4x4 matrix and a point in three-space using homogeneous coordinates, consistent with what is typically encountered in graphics applications.

$$\begin{bmatrix} Q_x \\ Q_y \\ Q_z \\ Q_w \end{bmatrix} = Q = M \cdot P = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} Q_x & Q_y & Q_z & Q_w \end{bmatrix} = Q^T = P^T \cdot M^T$$

$$= \begin{bmatrix} P_x & P_y & P_z & 1 \end{bmatrix} \cdot \begin{bmatrix} M_{11} & M_{21} & M_{31} & M_{41} \\ M_{12} & M_{22} & M_{32} & M_{42} \\ M_{13} & M_{23} & M_{33} & M_{43} \\ M_{14} & M_{24} & M_{34} & M_{44} \end{bmatrix}$$

**(Eq. B.32)**

## B.3.2 Transforming a Vector Using Vector-Matrix Multiplication

In addition to transforming points, it is also often useful to transform vectors, such as normal vectors, from one space to another. However, the computations used to transform a vector are different from those used to transform a point. Vectors have direction and magnitude but do not have a position in space. Thus, for example, a pure translation has no effect on a vector. If the transformation of one space to another is a pure rotation and uniform scale, then those transformations can be applied directly to the vector. However, it is not so obvious how to apply transformations that incorporate nonuniform scale.

The transformation of a vector can be demonstrated by considering a point, *P*, which satisfies a planar equation (Equation B.33). Note that (*a, b, c*) represents a vector normal to the plane. Showing how to transform a planar equation will, in effect, show how to transform a vector. The point is transformed by a matrix, *M* (Equation B.34). Because the transformations of rotation, translation, and scale preserve planarity, the transformed point, $P'$, will satisfy some new planar equation, $N'$, in the transformed space (Equation B.35). Substituting the definition of the transformed point, Equation B.34, into Equation B.35 produces Equation B.36. If the transformed planar equation is equal to the original normal postmultiplied by the inverse of the transformation matrix (Equation B.37), then Equation B.35 is satisfied, as shown by Equation B.38. The transformed normal vector is, therefore, $(a', b', c')$.

$$a \cdot x + b \cdot y + c \cdot z + d = 0$$

$$\begin{bmatrix} a & b & c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

$$N^T \cdot P = 0 \tag{Eq. B.33}$$

$$P' = M \cdot P \tag{Eq. B.34}$$

$$N'^T \cdot P' = 0 \tag{Eq. B.35}$$

$$N'^T \cdot M \cdot P = 0 \tag{Eq. B.36}$$

$$N'^T = N^T \cdot M^{-1} \tag{Eq. B.37}$$

$$N^T \cdot M^{-1} \cdot M \cdot P = N^T \cdot P = 0 \tag{Eq. B.38}$$

If one has a vector (*a, b, c*) to transform, one should just assume that it is a normal vector for a plane passing through the origin [*a, b, c,* 0] and postmultiply it by the inverse of the transformation matrix (Equation B.37). If it is desirable to keep all vectors as column vectors, then Equation B.39 can be used.

$$N' = (N'^T)^T = (N^T \cdot M^{-1})^T = (M^{-1})^T \cdot N \qquad \text{(Eq. B.39)}$$

## B.3.3 Axis-Angle Rotations

Given an axis of rotation $A = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}$ of unit length and an angle θ to rotate by (Figure B.22), the rotation matrix *M* can be formed by Equation B.40. This is a more direct way to rotate a point around an axis, as opposed to implementing the rotation as a series of rotations about the global axes.

$$\hat{A} = \begin{bmatrix} a_x \cdot a_x & a_x \cdot a_y & a_x \cdot a_z \\ a_y \cdot a_x & a_y \cdot a_y & a_y \cdot a_z \\ a_z \cdot a_x & a_z \cdot a_y & a_z \cdot a_z \end{bmatrix}$$

$$A^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

$$\begin{aligned} M &= \hat{A} + \cos\theta \cdot (I - \hat{A}) + \sin\theta \cdot A^* \\ P' &= M \cdot P \end{aligned} \qquad \text{(Eq. B.40)}$$

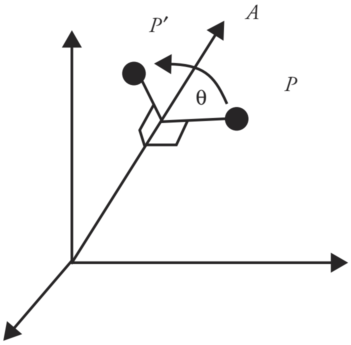


**Figure B.22** Axis-angle rotation

## B.3.4 Quaternions

Quaternions are discussed in Chapters 2 and 3. The equations from those chapters, along with additional equations, are collected here to facilitate the discussion.

### Quaternion Arithmetic

Quaternions are four-tuples and can be considered as a scalar combined with a vector (Equation B.41). Addition and multiplication are defined for quaternions by Equation B.42 and Equation B.43, respectively. Quaternion multiplication is associative (Equation B.44), but it is not commutative (Equation B.45). The magnitude of a quaternion is computed as the square root of the sum of the squares of its four components (Equation B.46). Quaternion multiplication has an identity (Equation B.47) and an inverse (Equation B.48). The inverse distributes over quaternion multiplication similarly to how the inverse distributes over matrix multiplication (Equation B.49). A quaternion is normalized by dividing it by its magnitude (Equation B.50).

$$q = [s, x, y, z] = [s, v] \tag*{(Eq. B.41)}$$

$$[s_1, v_1] + [s_2, v_2] = [s_1 + s_2, v_1 + v_2] \tag*{(Eq. B.42)}$$

$$[s_1, v_1] \cdot [s_2, v_2] = [s_1 \cdot s_2 - v_1 \bullet v_2, s_1 \cdot v_2 + s_2 \cdot v_1 + v_1 \times v_2] \tag*{(Eq. B.43)}$$

$$(q_1 \cdot q_2) \cdot q_3 = q_1 \cdot (q_2 \cdot q_3) \tag*{(Eq. B.44)}$$

$$q_1 \cdot q_2 \neq q_2 \cdot q_1 \tag*{(Eq. B.45)}$$

$$\|q\| = \sqrt{s^2 + x^2 + y^2 + z^2} \tag*{(Eq. B.46)}$$

$$\left[s, v\right] \cdot \left[1, (0, 0, 0)\right] = \left[s, v\right] \tag*{(Eq. B.47)}$$

$$\begin{aligned} q^{-1} &= (1/\|q\|)^2 \cdot [s, -v] \\ q^{-1} \cdot q = q \cdot q^{-1} &= \left[1, (0, 0, 0)\right] \end{aligned} \tag*{(Eq. B.48)}$$

$$(p \cdot q)^{-1} = q^{-1} \cdot p^{-1} \tag*{(Eq. B.49)}$$

$$q/(\|q\|) \tag*{(Eq. B.50)}$$

### Rotations by Quaternions

A point in space is represented by a vector quantity in quaternion form by using a zero scalar value (Equation B.51). A quaternion can be used to rotate a vector using quaternion multiplication (Equation B.52). Compound rotations can be implemented by premultiplying the corresponding quaternions (Equation B.53), similar to what's routinely done when rotation matrices are used. As should be expected, compounding a rotation with its inverse produces the identity transformation for vectors (Equation B.54). An axis-angle rotation is represented by a unit quaternion, as shown in Equation B.55. Any scalar multiple of a quaternion represents the same rotation. In particular, the negation of a quaternion (negating each of its four components, $-q = [-s, -x, y, -z]$) represents the same rotation that the original quaternion represents (Equation B.56).

$$v = [0, x, y, z] \qquad \text{(Eq. B.51)}$$

$$v' = \mathrm{Rot}(v) = q \cdot v \cdot q^{-1} \qquad \text{(Eq. B.52)}$$

$$\begin{aligned} \mathrm{Rot}_q(\mathrm{Rot}_p(v)) &= q \cdot (p \cdot v \cdot p^{-1}) \cdot q^{-1} \\ &= ((q \cdot p) \cdot v \cdot (p^{-1} \cdot q^{-1})) \\ &= ((q \cdot p) \cdot v \cdot (q \cdot p)^{-1}) \\ &= \mathrm{Rot}_{qp}(v) \end{aligned} \qquad \text{(Eq. B.53)}$$

$$\begin{aligned} \mathrm{ot}^{-1}(\mathrm{Rot}(v)) &= q^{-1} \cdot (q \cdot v \cdot q^{-1}) \cdot q \\ &= (q^{-1} \cdot q) \cdot v \cdot (q^{-1} \cdot q) = v \end{aligned} \qquad \text{(Eq. B.54)}$$

$$\mathrm{Rot}_{[\theta, x, y, z]} = [\cos(\theta/2), \sin(\theta/2) \cdot (x, y, z)] \qquad \text{(Eq. B.55)}$$

$$\begin{aligned} -q &= \mathrm{Rot}_{[-\theta, -(x, y, z)]} \\ &= [\cos(-\theta/2), \sin((-\theta)/2) \cdot (-(x, y, z))] \\ &= [\cos(\theta/2), -\sin(\theta/2) \cdot (-(x, y, z))] \\ &= [\cos(\theta/2), \sin(\theta/2) \cdot x, y, z] \\ &= \mathrm{Rot}_{[\theta, (x, y, z)]} \\ &= q \end{aligned} \qquad \text{(Eq. B.56)}$$

### Conversions

It is often useful to convert back and forth between rotation matrices and quaternions. Often, quaternions are used to interpolate between orientations, and the result is converted to a rotation matrix so as to combine it with other matrices in the display pipeline.

Given a unit quaternion ($q = [s, x, y, z]$, $s^2 + x^2 + y^2 + z^2 = 1$), one can easily determine the corresponding rotation matrix by rotating the three unit vectors that correspond to the principal axes. The rotated vectors are the columns of the equivalent rotation matrix (Equation B.57).

$$\begin{bmatrix} 1-2\cdot y^2-2\cdot z^2 & 2\cdot x\cdot y-2\cdot s\cdot z & 2\cdot x\cdot z+2\cdot s\cdot y \\ 2\cdot x\cdot y+2\cdot s\cdot z & 1-2\cdot x^2-2\cdot z^2 & 2\cdot y\cdot z-2\cdot s\cdot x \\ 2\cdot x\cdot z-2\cdot s\cdot y & 2\cdot y\cdot z+2\cdot s\cdot x & 1-2\cdot x^2-2\cdot y^2 \end{bmatrix} \qquad \text{(Eq. B.57)}$$

Given a rotation matrix, one can use the definitions for the terms of the matrix in Equation B.57 to solve for the elements of the equivalent unit quaternion. The fact that the unit quaternion has a magnitude of one ($s^2 + x^2 + y^2 + z^2 = 1$), makes it easy to see that the diagonal elements sum to $4 \cdot s^2 - 1$. Summing the diagonal elements of the matrix in Equation B.58 results in Equation B.59. The diagonal elements can also be used to solve for the remaining terms (Equation B.60). The square roots of these last equations can be avoided if the off-diagonal elements are used to solve for *x, y,* and *z* at the expense of testing for a divide by an *s* that is equal to zero (in which case Equation B.60 can be used).

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} \qquad \text{(Eq. B.58)}$$

$$s = \frac{\sqrt{m_{0,0} + m_{1,1} + m_{2,2} + 1}}{2} \qquad \text{(Eq. B.59)}$$

$$m_{0,0} = 1-2\cdot y^2-2\cdot z^2 = 1-2\cdot(y^2+z^2)$$

$$= 1-2\cdot(1-x^2-s^2) = -1+2\cdot x^2+2\cdot s$$

$$x = \sqrt{\frac{m_{0,0}+1-2\cdot s^2}{2}}$$

$$y = \sqrt{\frac{m_{1,1}+1-2\cdot s^2}{2}}$$

$$z = \sqrt{\frac{m_{2,2}+1-2\cdot s^2}{2}} \qquad \text{(Eq. B.60)}$$

## B.4 Interpolating and Approximating Curves

This section covers many of the basic terms and concepts needed to interpolate values in computer animation. It is not a complete treatise of curves but an overview of the important ones. While many of the terms and concepts discussed are applicable to functions in general, they are presented as they relate to functions having to do with the practical interpolation of points in Euclidean space as typically used in computer animation applications. For more complete discussions of the topics contained here, see, for example, Mortenson [14], Rogers and Adams [18], Farin [4], and Bartels, Beatty, and Barsky [1].

### B.4.1 Equations: Some Basic Terms

For present purposes, there are three types of equations: *explicit*, *implicit*, and *parametric*. *Explicit equations* are of the form $y = f(x)$. The explicit form is good for generating points because it generates a value of $y$ for any value of $x$ put into the function. The drawback of the explicit form is that it is dependent on the choice of coordinate axes and it is ambiguous if there is more than one $y$ for a given $x$ (such as $y = \sqrt{x}$, in which an input value of 4 would generate values of either 2 or –2). *Implicit equations* are of the form $f(x, y) = 0$. The implicit form is good for testing to see if a point is on a curve because the coordinates of the point can easily be put into the equation for the curve and checked to see if the equation is satisfied. The drawback of the implicit form is that generating a series of points along a curve is often desired, and implicit forms are not generative. *Parametric equations* are of the form $x = f(t)$, $y = g(t)$. For any given value of $t$, a point $(x, y)$ is generated. This form is good for generating a sequence of points as ordered values of $t$ are given. The parametric form is also useful because it can be used for multivalued functions of $x$, which are problematic for explicit equations.

Equations can be classified according to the terms contained in them. Equations that contain only variables raised to a power are *polynomial* equations. If the highest power is one, then the equation is *linear*. If the highest power is two, then the equation is *quadratic*. If the highest power is three, then it is *cubic*. If the equation is not a simple polynomial but rather contains sines, cosines, log, or a variety of other functions, then it is called *transcendental*. In computer graphics, the most commonly encountered type of function is the cubic polynomial.

*Continuity* refers to how well behaved the curve is in a mathematical sense. For a value arbitrarily close to a $x_0$ if the function is arbitrarily close to $f(x_0)$, then it has *positional*, or *zeroth-order*, continuity ($C^0$) at that point. If the slope of the curve (or the first derivative of the function) is continuous, then the function has *tangential*, or *first-order*, continuity ($C^1$). This is extended to all of the function's derivatives, although for purposes of computer animation the concern is with first-order continuity or, possibly, *second-order,* or *curvature,* continuity ($C^2$). Polynomials are infinitely continuous.

If a curve is pieced together from individual curve segments, one can speak of *piecewise properties*—the properties of the individual pieces. For example, a sequence of straight line segments, sometimes called a *polyline* or a *wire*, is piecewise linear. A major concern regarding piecewise curves is the continuity conditions at the junctions of the curve segments. If one curve segment begins where the previous segment ends, then there is *zeroth-order,* or *positional,* continuity at the junction. If the beginning tangent of one curve segment is the same as the ending tangent of the previous curve segment, then there is *first-order,* or *tangential,* continuity at the junction. If the beginning curvature of one curve segment is the same as the ending curvature of the previous curve segment, then there is *second-order,* or *curvature,* continuity at the junction. Typically, computer animation is not concerned with continuity beyond second order.

Sometimes in discussions of the continuity at segment junctions, a distinction is made between *parametric continuity* and *geometric continuity* (e.g., [14]). So far the discussion has concerned parametric continuity. Geometric continuity is less restrictive. First-order parametric continuity, for example, requires that the ending tangent vector of the first segment be the same as the beginning tangent vector of the second. First-order geometric continuity, on the other hand, requires that only the direction of the tangents be the same, and it allows the magnitudes of the tangents to be different. Similar definitions exist for higher-order geometric continuity. One distinction worth mentioning is that parametric continuity is sensitive to the rate at which the parameter varies relative to the length of the curve traced out. Geometric continuity is not sensitive to this rate.

When a curve is constructed from a set of points and the curve passes through the points, it is said to *interpolate* the points. However, if the points are used to control the general shape of the curve, with the curve not necessarily passing through them, then the curve is said to *approximate the points. Interpolation* is also used generally to refer to all approaches for constructing a curve from a set of points. For a given interpolation technique, if the resulting curve is guaranteed to lie within the convex hull of the set of points, then it is said to have the *convex hull property.*

## B.4.2 Simple Linear Interpolation: Geometric and Algebraic Forms

Simple linear interpolation is given by Equation B.61 and shown in Figure B.23. Notice that the interpolants, $1 - u$ and $u,$ sum to one. This property ensures that the interpolating curve (in this case a straight line) falls within the convex hull of

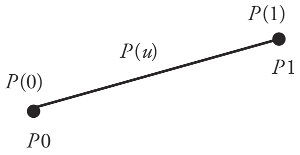


**Figure B.23** Linear interpolation

the geometric entities being interpolated (in this simple case the convex hull is the straight line itself).

$$P(u) = (1-u) \cdot P0 + u \cdot P1 \qquad \text{(Eq. B.61)}$$

Using more general notation, one can rewrite the equation above as in Equation B.62. Here $F0$ and $F1$ are called *blending* functions. This is referred to as the *geometric* form because the geometric information, in this case $P0$ and $P1$, is explicit in the equation.

$$P(u) = F_0(u) \cdot P0 + F_1(u) \cdot P1 \qquad \text{(Eq. B.62)}$$

The linear interpolation equation can also be rewritten as in Equation B.63. This form is typical of polynomial equations in which the terms are collected according to coefficients of the variable raised to a power. It is more generally written as Equation B.64. In this case there are only linear terms. This way of expressing the equation is referred to as the *algebraic* form.

$$P(u) = (P1 - P0) \cdot u + P0 \qquad \text{(Eq. B.63)}$$

$$P(u) = a1 \cdot u + a0 \qquad \text{(Eq. B.64)}$$

Alternatively, both of these forms can be put in a *matrix representation*. The geometric form becomes Equation B.65 and the algebraic form becomes Equation B.66. The geometric form is useful in situations in which the geometric information (the points defining the curve) needs to be frequently updated or replaced. The algebraic form is useful for repeated evaluation of a single curve for different values of the parameter. The fully expanded form is shown in Equation B.67. The curves discussed below can all be written in this form. Of course, depending on the actual curve type, the $U$ (variable), $M$ (coefficient), and $B$ (geometric information) matrices will contain different values.

$$P(u) = \begin{bmatrix} F_0(u) \\ F_1(u) \end{bmatrix} \begin{bmatrix} P0 & P1 \end{bmatrix} = FB^T \qquad \text{(Eq. B.65)}$$

$$P(u) = \begin{bmatrix} u & 1 \end{bmatrix} \begin{bmatrix} a1 \\ a0 \end{bmatrix} = U^T A \qquad \text{(Eq. B.66)}$$

$$P(u) = \begin{bmatrix} u & 1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} P0 \\ P1 \end{bmatrix} = U^T MB = FB = U^T A \qquad \text{(Eq. B.67)}$$

### B.4.3 Parameterization by Arc Length

It should be noted that in general there is not a linear relationship between changes in the parameter *u* and the distance traveled along a curve (its *arc length*). It happens to be true in the example above concerning a straight line and the parameter *u*. However, as Mortenson [14] points out, there are other equations that trace out a straight line in space that are fairly convoluted in their relationship between changes in the parameter and distance traveled. For example, consider Equation B.68, which is linear in $P0$ and $P1$. That is, it traces out a straight line in space between $P0$ and $P1$. However, it is nonlinear in *u*. As a result, the curve is not traced out in a nice monotonic, constant-velocity manner. The nonlinear relationship is evident in most parameterized curves unless special care is taken to ensure constant velocity. (See Chapter 3, "Controlling the Motion Along a Curve.")

$$P(u) = P0 + ((1-u) \cdot u + u) \cdot (P1 - P0) \qquad \text{(Eq. B.68)}$$

### B.4.4 Computing Derivatives

One of the matrix forms for parametric curves, as shown in Equation B.67 for linear interpolation, is $U^T MB$. Parametric curves of any polynomial order can be put into this matrix form. Often, it is useful to compute the derivatives of a parametric curve. This can be done easily by taking the derivative of the $U$ vector. For example, the first two derivatives of a cubic curve, shown in Equation B.69, are easily evaluated for any value of *u*.

$$\begin{aligned} P(u) &= U^T MB = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} MB \\ P'(u) &= U'^T MB = \begin{bmatrix} 3 \cdot u^2 & 2 \cdot u & 1 & 0 \end{bmatrix} MB \\ P''(u) &= U''^T MB = \begin{bmatrix} 6 \cdot u & 2 & 0 & 0 \end{bmatrix} MB \end{aligned} \qquad \text{(Eq. B.69)}$$

## B.4.5 Hermite Interpolation

Hermite interpolation generates a cubic polynomial from one point to another. In addition to specifying the beginning and ending points ($P_i$, $P_{i+1}$), the user needs to supply beginning and ending tangent vectors ($P'_i$, $P'_{i+1}$) as well (Figure B.24). The general matrix form for a curve is repeated in Equation B.70, and the Hermite matrices are given in Equation B.71.

$$P(u) = U^T MB \qquad \text{(Eq. B.70)}$$

$$U^T = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} P_i \\ P_{i+1} \\ P'_i \\ P'_{i+1} \end{bmatrix} \qquad \text{(Eq. B.71)}$$

Continuity between beginning and ending tangent vectors of connected segments is ensured by merely using the ending tangent vector of one segment as the beginning tangent vector of the next. A composite Hermite curve (piecewise cubic with first-order continuity at the junctions) is shown in Figure B.25.

Trying to put a Hermite curve through a large number of points, which requires the user to specify all of the needed tangent vectors, can be a burden. There are several techniques to get around this. One is to enforce second-degree continuity.

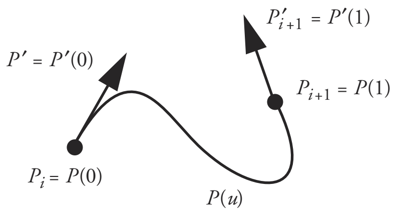


**Figure B.24** Hermite interpolation

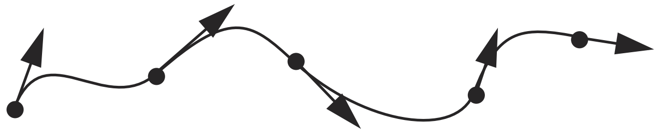


**Figure B.25** Composite Hermite curve

This requirement provides enough constraints so that the user does not have to provide interior tangent vectors; they can be calculated automatically. See Rogers and Adams [18] or Mortenson [14] for alternative formulations. A more common technique is the Catmull-Rom spline.

## B.4.6 Catmull-Rom Spline

The Catmull-Rom curve can be viewed as a Hermite curve in which the tangents at the interior control points are automatically generated according to a relatively simple geometric procedure (as opposed to the more involved numerical techniques referred to above). For each interior point, $P_i$, the tangent at that point, $P_i'$, is computed as one-half the vector from the previous control point, $P_{i-1}$, to the following control point, $P_{i+1}$ (Equation B.72), as shown in Figure B.26.[1] The matrices for the Catmull-Rom curve in general matrix form are given in Equation B.73. A Catmull-Rom spline is a specific type of cardinal spline.

$$P_i' = (1/2) \cdot (P_{i+1} - P_{i-1}) \qquad \text{(Eq. B.72)}$$

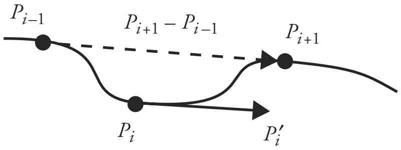


**Figure B.26** Catmull-Rom spline

[1] Farin [4] describes the Catmull-Rom spline curve in terms of a cubic Bezier curve by defining interior control points. Placement of the interior control points is determined by use of an auxiliary knot vector. With a uniform distance between knot values, the control points are displaced from the point to be interpolated by one-sixth of the vector from the previous interpolated point to the following interpolated point. Tangent vectors are three times the vector from an interior control point to the interpolated point. This results in the Catmull-Rom tangent vector described here.

$$U^T = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$M = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix} \qquad \text{(Eq. B.73)}$$

For the end conditions, the user can provide tangent vectors at the very beginning and at the very end of the cubic curve. Alternatively, various automatic techniques can be used. For example, the beginning tangent vector can be defined as follows. The vector from the second point ($P_1$) to the third point ($P_2$) is subtracted from the second point and used as a virtual point to which the initial tangent is directed. This tangent is computed by Equation B.74. Figure B.27 shows the formation of the initial tangent curve according to the equation, and Figure B.28 shows a curve that uses this technique.

$$P'(0.0) = \frac{1}{2}\cdot(P_1 - (P_2 - P_1) - P_0) = \frac{1}{2}\cdot(2\cdot P_1 - P_2 - P_0) \qquad \text{(Eq. B.74)}$$

A drawback of the Catmull-Rom formulation is that an internal tangent vector is not dependent on the position of the internal point relative to its two neighbors. In Figure B.29, all three positions ($Q_i$, $P_i$, $R_i$) for the $i$th point would have the same tangent vector.

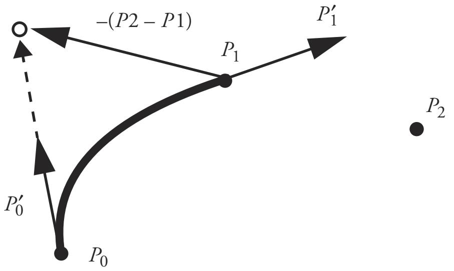


**Figure B.27** Automatically forming the initial tangent of a Catmull-Rom spline

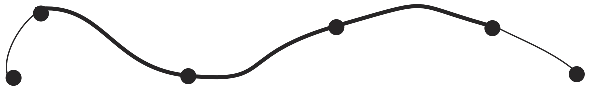


**Figure B.28** Catmull-Rom spline with end conditions using Equation B.74

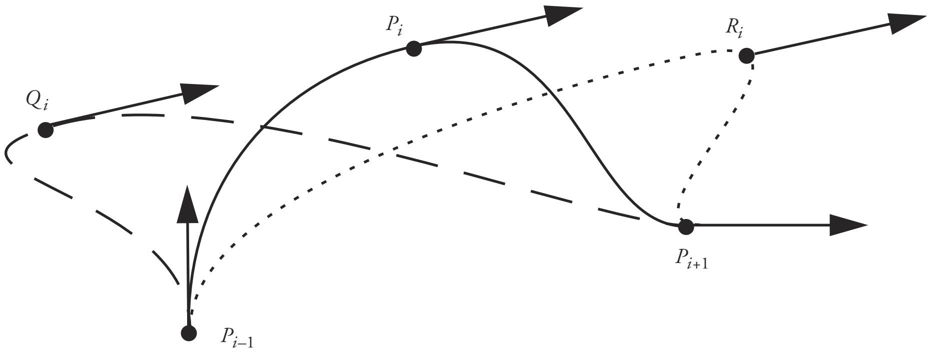


**Figure B.29** Three curve segments, $(P_{i-1}, P_i, P_{i+1})$, $(P_{i-1}, Q_i, P_{i+1})$, $(P_{i-1}, R_i, P_{i+1})$, using the standard Catmull-Rom form for computing the internal tangent

An advantage of Catmull-Rom is that the calculation to compute the internal tangent vectors is extremely simple and fast. However, for each segment the tangent computation is a one-time-only cost. It is then used repeatedly in the computation for each new point in that segment. Therefore, it often makes sense to spend a little more time computing more appropriate internal tangent vectors to obtain a better set of points along the segment. One alternative is to use a vector perpendicular to the plane that bisects the angle made by $P_{i-1} - P_i$ and $P_{i+1} - P_i$ (Figure B.30). This can be computed easily by adding the normalized vector from $P_{i-1}$ to $P_i$ with the normalized vector from $P_i$ to $P_{i+1}$.

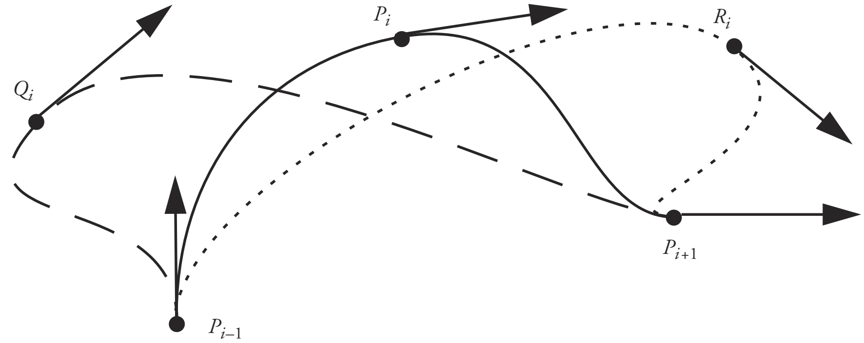


**Figure B.30** Three curve segments, $(P_{i-1}, P_i, P_{i+1})$, $(P_{i-1}, Q_i, P_{i+1})$, $(P_{i-1}, R_i, P_{i+1})$ using the perpendicular to the angle bisector for computing the internal tangent

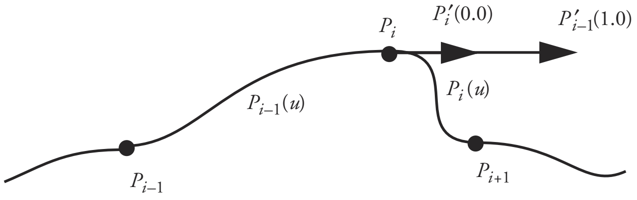


**Figure B.31** Interior tangents based on relative segment lengths

Another modification, which can be used with the original Catmull-Rom tangent computation or with the bisector technique above, is to use the relative position of the internal point ($P_i$) to independently determine the length of the tangent vector for each segment it is associated with. Thus, a point $P_i$ has an ending tangent vector associated with it for the segment from $P_{i-1}$ to $P_i$ as well as a beginning tangent vector associated with it for the segment $P_i$ to $P_{i+1}$. These tangents have the same direction but different lengths. This relaxes the $C^1$ continuity of the Catmull-Rom spline and uses $G^1$ continuity instead. For example, an initial tangent vector at an interior point is determined as the vector from $P_{i-1}$ to $P_{i+1}$. The ending tangent vector for the segment $P_{i-1}$ to $P_i$ is computed by scaling this initial tangent vector by the ratio of the distance between the points $P_i$ and $P_{i-1}$ to the distance between points $P_{i-1}$ and $P_{i+1}$. Referring to the segment between $P_{i-1}$ and $P_i$ as $P_{i-1}(u)$ results in Equation B.75. A similar calculation for the beginning tangent vector of the segment between $P_i$ and $P_{i+1}$ results in Equation B.76. These tangents can be seen in Figure B.31.The computational cost of this approach is only a little more than the standard Catmull-Rom spline and seems to give more intuitive results.

$$P'_{i-1}(1.0) = \frac{|P_i - P_{i-1}|}{|P_{i+1} - P_{i-1}|} \cdot (P_{i+1} - P_{i-1}) \qquad \text{(Eq. B.75)}$$

$$P'_i(0.0) = \frac{|P_{i+1} - P_i|}{|P_{i+1} - P_{i-1}|} \cdot (P_{i+1} - P_{i-1}) \qquad \text{(Eq. B.76)}$$

## B.4.7 Four-Point Form

Fitting a cubic segment to four points ($P_0$, $P_1$, $P_2$, $P_3$), assigned to user-specified parametric values ($u_0$, $u_1$, $u_2$, $u_3$), can be accomplished by setting up the linear system of equations for the points (Equation B.77) and solving for the unknown coefficient matrix. In the case of parametric values of 0, 1/3, 2/3, and 1, the matrix is given by Equation B.78. However, with this form it is difficult to join segments with $C^1$ continuity.

$$P(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} u_0^3 & u_0^2 & u_0 & 1 \\ u_1^3 & u_1^2 & u_1 & 1 \\ u_2^3 & u_2^2 & u_2 & 1 \\ u_3^3 & u_3^2 & u_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{Eq. B.77}$$

$$M = \frac{1}{2}\begin{bmatrix} -9 & 27 & -27 & 9 \\ 18 & -45 & 36 & -9 \\ -11 & 18 & -9 & 2 \\ 2 & 0 & 0 & 0 \end{bmatrix} \tag{Eq. B.78}$$

### B.4.8 Blended Parabolas

Blending overlapping parabolas to define a cubic segment is another approach to interpolating a curve through a set of points. In addition, the end conditions are handled by parabolic segments, which is consistent with how the interior segments are defined. Blending parabolas results in a formulation that is very similar to Catmull-Rom in that each segment is defined by four points, it is an interpolating curve, and local control is provided. Under the assumptions used here for Catmull-Rom and the blended parabolas, the interpolating matrices are identical.

For each overlapping triple of points, a parabolic curve is defined by the three points. A cubic curve segment is created by linearly interpolating between the two overlapping parabolic segments. More specifically, take the first three points, $P_0$, $P_1$, $P_2$, and fit a parabola, $P(u)$, through them using the following constraints: $P(0.0) = P_0$, $P(0.5) = P_1$, $P(1.0) = P_2$. Take the next group of three points, $P_1$, $P_2$, $P_3$, which partially overlap the first set of three points, and fit a parabola, $R(u)$, through them using similar constraints: $R(0.0) = P_1$, $R(0.5) = P_2$, $R(1.0) = P_3$. Between points $P_1$ and $P_2$ the two parabolas overlap. Reparameterize this region into the range [0.0, 1.0] and linearly blend the two parabolic segments (Figure

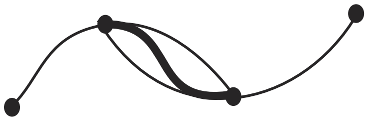


**Figure B.32** Parabolic blend segment

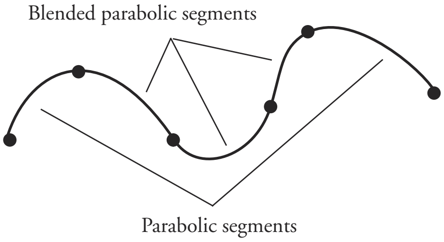


**Figure B.33** Multiple parabolic blend segments

B.32). The result can be put in matrix form for a cubic curve using the four points as the geometric information together with the coefficient matrix shown in Equation B.79. To interpolate a list of points, calculate interior segments using this equation. End conditions can be handled by constructing parabolic arcs at the very beginning and very end (Figure B.33).

$$M = \frac{1}{2}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \qquad \text{(Eq. B.79)}$$

This form assumes that all points are equally spaced in parametric space. Often it is the case that even spacing is not present. In such cases, relative cord length can be used to estimate parametric values. The derivation is a bit more involved [18], but the final result can still be formed into a 4x4 matrix and used to produce a cubic polynomial in the interior segments.

## B.4.9 Bezier Interpolation/Approximation

A cubic Bezier curve is defined by the beginning point and the ending point, which are interpolated, and two interior points, which control the shape of the curve. The cubic Bezier curve is similar to the Hermite form. The Hermite form uses beginning and ending tangent vectors to control the shape of the curve; the Bezier form uses auxiliary control points to define tangent vectors. A cubic curve is defined by four points: $P_0$, $P_1$, $P_2$, $P_3$. The beginning and ending points of the curve are $P_0$ and $P_3$, respectively. The interior control points used to control the shape of the curve and define the beginning and ending tangent vectors are $P_1$ and $P_2$. See Figure B.34. The coefficient matrix for a single cubic Bezier curve is shown in Equation B.80. In the cubic case, $P'(0) = 3 \cdot (P_1 - P_0)$ and $P'(1) = 3 \cdot (P_3 - P_2)$

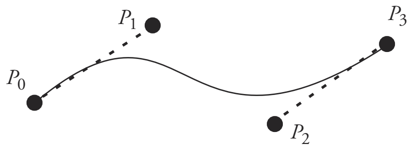


**Figure B.34** Cubic Bezier curve segment

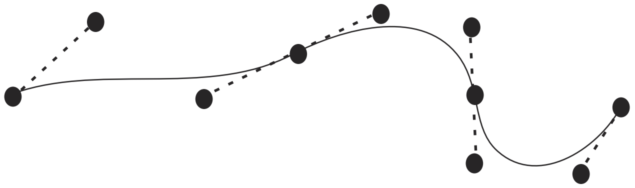


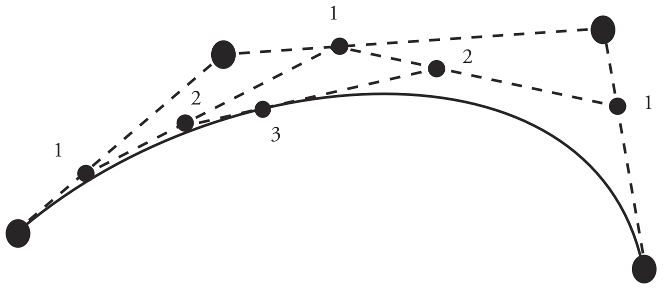
**Figure B.35** Composite cubic Bezier curve showing tangents and colinear control points

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \qquad \text{(Eq. B.80)}$$

Continuity between adjacent Bezier segments can be controlled by colinearity of the control points on either side of the shared beginning/ending point of the two curve segments where they join (Figure B.35). In addition, the Bezier curve form allows one to define a curve of arbitrary order. If three interior control points are used, then the resulting curve will be quartic; if four interior control points are used, then the resulting curve will be quintic. See Mortenson [14] for a more complete discussion.

## B.4.10 De Casteljau Construction of Bezier Curves

The de Casteljau method is a way to geometrically construct a Bezier curve. Figure B.36 shows the construction of a point at $u = 1/3$. This method constructs a point $u$ along the way between paired control points (identified by a “1” in Figure B.36). Then points are constructed $u$ along the way between points just previously constructed. These new points are marked “2” in Figure B.36. In the cubic case, in which there were four initial points, there are two newly constructed points. The point on the curve is constructed by going $u$ along the way between these two points. This can be done for any values of $u$ and for any order of curve. Higher-order Bezier curves require more iterations to produce the final point on the curve.



Interpolation steps

1. 1/3 of the way between paired points
2. 1/3 of the way between points of step 1
3. 1/3 of the way between points of step 2

**Figure B.36** De Casteljau construction of a point on a cubic Bezier curve

## B.4.11 Tension, Continuity, and Bias Control

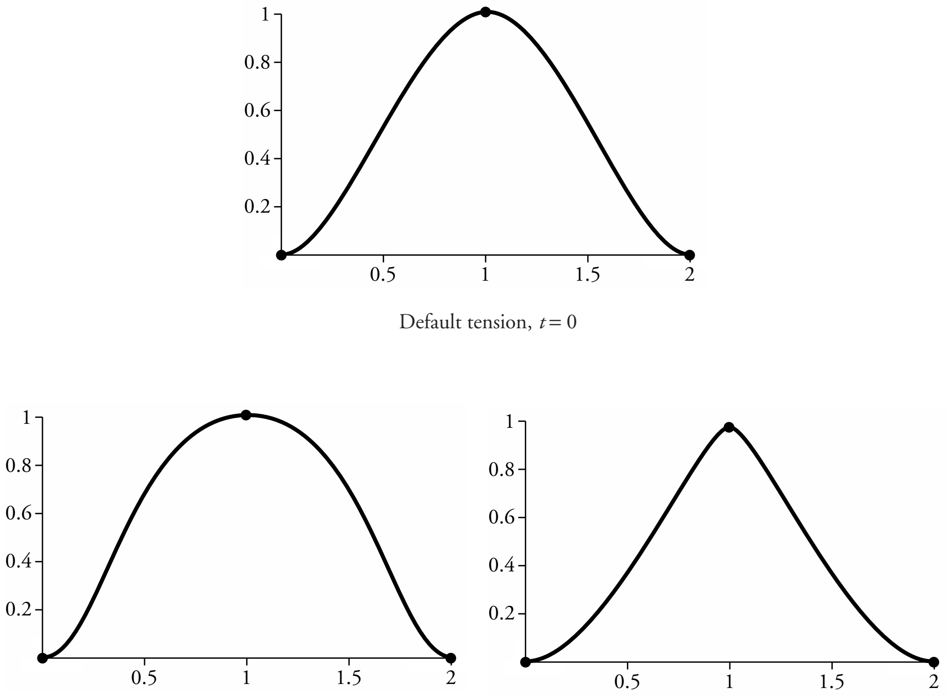
Often an animator wants better control over the interpolation of key frames than the standard interpolating splines provide. For better control of the shape of an interpolating curve, Kochanek [11] suggests a parameterization of the internal tangent vectors based on the three values tension, continuity, and bias. The three parameters are explained by decomposing each internal tangent vector into an incoming part and an outgoing part. These tangents are referred to as the left and right parts, respectively, and are notated by $T_i^L$ and $T_i^R$ for the tangents at $P_i$.

Tension controls the sharpness of the bend of the curve at $P_i$. It does this by means of a scale factor that changes the length of both the incoming and outgoing tangents at the control point (Equation B.81). In the default case, $t = 0$ and the tangent vector is the average of the two adjacent chords or, equivalently, half of the cord between the two adjacent points, as in the Catmull-Rom spline. As the tension parameter, $t$, goes to one, the tangents become shorter until they reach zero. Shorter tangents at the control point mean that the curve is pulled closer to a straight line in the neighborhood of the control point. See Figure B.37.

$$T_i^L = T_i^R = (1-t)\cdot\frac{1}{2}\cdot((P_{i+1}-P_i)+(P_i-P_{i-1})) \tag{Eq. B.81}$$

The continuity parameter, $c$, gives the user control over the continuity of the curve at the control point where the two curve segments join. The incoming (left) and outgoing (right) tangents at a control point are defined symmetrically with respect to the chords on either side of the control point. Assuming default tension, $c$ blends the adjacent chords to form the two tangents, as shown in Equation B.82.

$$\begin{aligned} T_i^L &= \frac{1-c}{2}(P_i-P_{i-1})+\frac{1+c}{2}(P_{i+1}-P_i) \\ T_i^R &= \frac{1+c}{2}(P_i-P_{i-1})+\frac{1-c}{2}(P_{i+1}-P_i) \end{aligned} \tag{Eq. B.82}$$



Default tension, $t = 0$

Low tension, $t = -3/4$

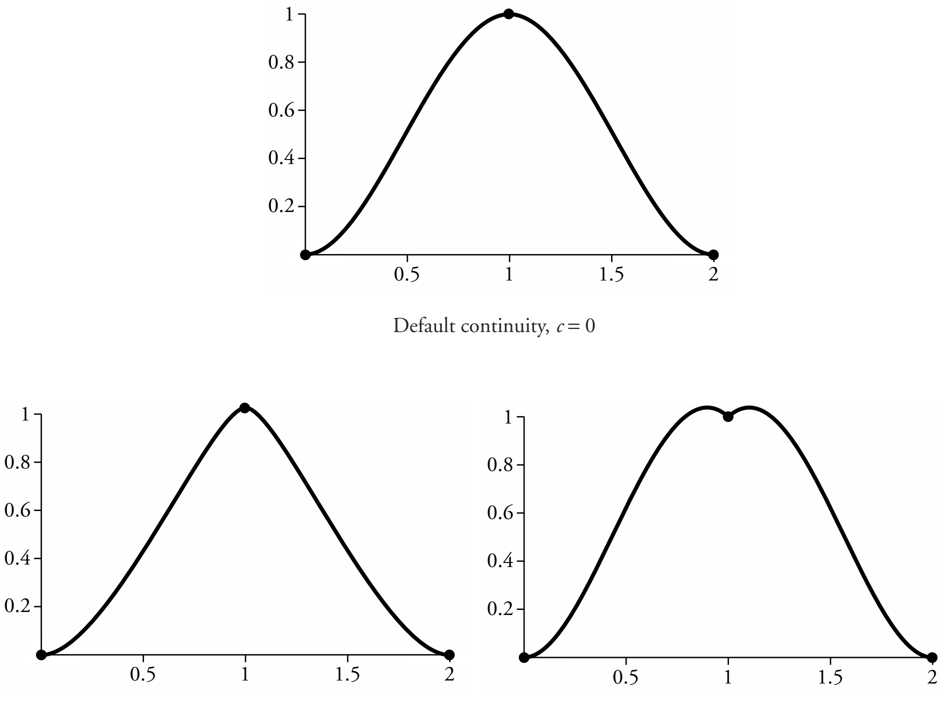
High tension, $t = 3/4$

**Figure B.37** The effect of varying the tension parameter

The default value for continuity is $c = 0$, which produces equal left and right tangent vectors, resulting in continuity at the joint. As $c$ approaches –1, the left tangent approaches equality with the chord to the left of the control point and the right tangent approaches equality with the chord to the right of the control point. As $c$ approaches +1, the definitions of the tangents reverse themselves, and the left tangent approaches the right chord and the right tangent approaches the left chord. See Figure B.38.

Bias, $b$, defines a common tangent vector, which is a blend between the chord left of the control point and the chord right of the control point (Equation B.83). At the default value ($b = 0$), the tangent is an even blend of these two, resulting in a Catmull-Rom type of internal tangent vector. Values of $b$ approaching –1 bias the tangent toward the chord to the left of the control point, while values of $b$ approaching +1 bias the tangent toward the chord to the right. See Figure B.39.

$$T_i^R = T_i^L = \frac{1+b}{2}(P_i - P_{i-1}) + \frac{1-b}{2}(P_{i+1} - P_i) \qquad \text{(Eq. B.83)}$$



Default continuity, $c = 0$

Low continuity, $c = -3/4$

High continuity, $c = +3/4$

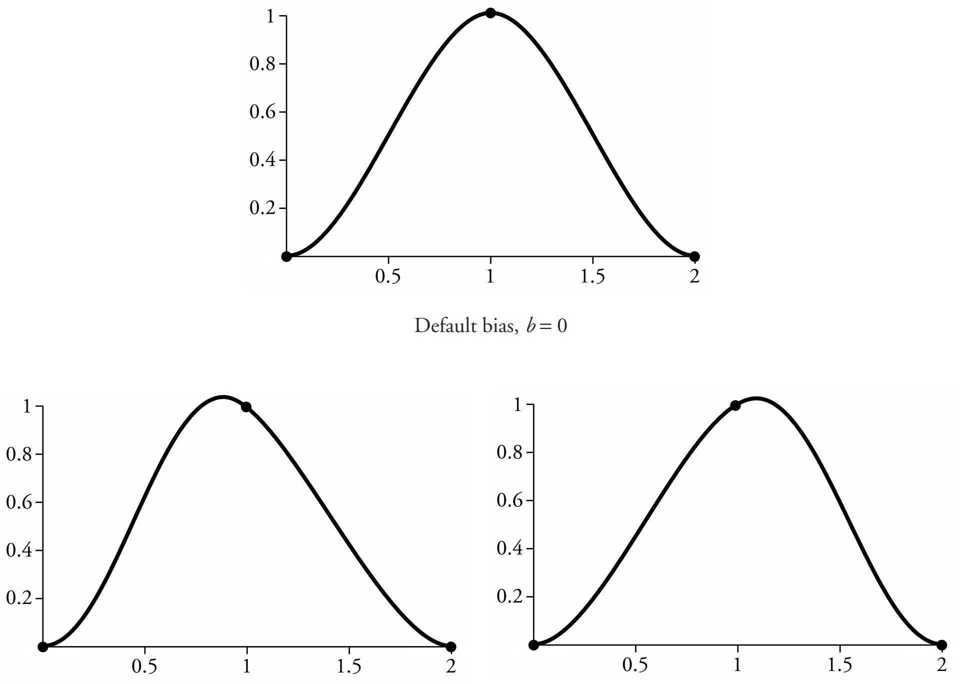
**Figure B.38** The effect of varying the continuity parameter (with default tension)

The three parameters tension, continuity, and bias are combined in Equation B.84.

$$\begin{aligned} T_i^L &= \frac{((1-t)(1-c)(1+b))}{2}(P_i - P_{i-1}) \\ &+ \frac{((1-t)(1+c)(1-b))}{2}(P_{i+1} - P_i) \\ T_i^R &= \frac{((1-t)(1+c)(1+b))}{2}(P_i - P_{i-1}) \\ &+ \frac{((1-t)(1-c)(1-b))}{2}(P_{i+1} - P_i) \end{aligned} \qquad \text{(Eq. B.84)}$$

## B.4.12 B-Splines

B-splines are the most flexible and useful type of curve, but they are also more difficult to grasp intuitively. The formulation includes Bezier curves as a special case. The formulation for B-spline curves decouples the number of control points from



Default bias, $b = 0$

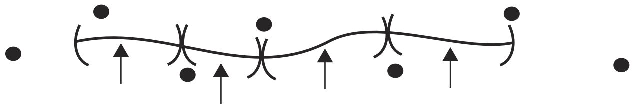
Low bias, $b = -3/4$

High bias, $b = +3/4$

**Figure B.39** The effect of varying the bias parameter (with default tension and continuity)

the degree of the resulting polynomial. It accomplishes this with additional information contained in the *knot vector*. An example of a *uniform knot vector* is [0, 1, 2, 3, 4, 5, 6, . . . , $n + k - 1$], in which the knot values are uniformly spaced apart. In this knot vector, $n$ is the number of control points and $k$ is the degree of the B-spline curve. The parametric value varies between the first and last values of the knot vector. The knot vector establishes a relationship between the parametric value and the control points. With replication of values in the knot vector, the curve can be drawn closer to a particular control point up to the point where the curve actually passes through the control point.

A particularly simple, yet useful, type of B-spline curve is a uniform cubic B-spline curve. It is defined using four control points over the interval zero to one (Equation B.85). A compound curve is generated from an arbitrary number of control points by constructing a curve segment from each four-tuple of adjacent control points: $(P_i, P_{i+1}, P_{i+2}, P_{i+3})$ for $i = 1, 2, \ldots, n - 3$, where $n$ is the total number of control points (Figure B.40). Each section of the curve is generated by



Segments of the curve defined by different sets of four points

**Figure B.40** Compound cubic B-spline curve

multiplying the same 4x4 matrix by four adjacent control points with an interpolating parameter between zero and one. In this case, none of the control points is interpolated.

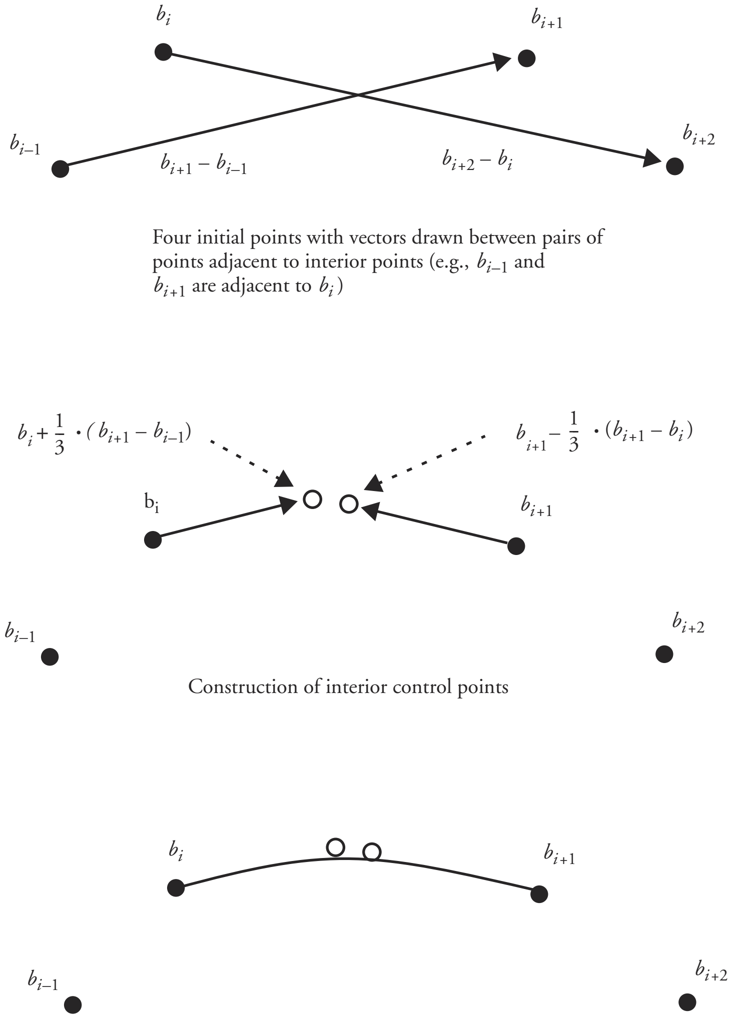
$$P(u) = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix} \quad \text{(Eq. B.85)}$$

NURBS, *Nonuniform rational B-splines*, are even more flexible than basic B-splines. NURBS allow for exact representation of circular arcs, whereas Bezier and nonrational B-splines do not. This is often important in modeling, but for purposes of animation, the basic periodic, uniform cubic B-spline is usually sufficient.

## B.4.13 Fitting Curves to a Given Set of Points

Sometimes it is desirable to interpolate a set of points using a Bezier formulation. The points to be interpolated can be designated as the endpoints of the Bezier curve segments, and the interior control points can be constructed by forming tangent vectors at the vertices, as with the Catmull-Rom formulation. The interior control points can be constructed by displacing the control points along the tangent lines just formed. For example, for the segment between given points $b_i$ and $b_{i+1}$, the first control point for the segment, $c_i^1$, can be positioned at $b_i + 1/3 \cdot (b_{i+1} - b_{i-1})$. The second control point for the segment, $c_i^2$, can be positioned at $b_{i+1} - 1/3 \cdot (b_{i+2} - b_i)$. See Figure B.41.

Other methods exist. Farin [4] presents a more general method of constructing the Bezier curve and, from that, constructing the B-spline control points. Both Farin [4] and Rogers and Adams [18] present a method of constructing a composite Hermite curve through a set of points that automatically calculates internal tangent vectors by assuming second-order continuity at the segment joints.



Four initial points with vectors drawn between pairs of points adjacent to interior points (e.g., $b_{i-1}$ and $b_{i+1}$ are adjacent to $b_i$)

$b_i + \frac{1}{3} \cdot (b_{i+1} - b_{i-1})$
$b_{i+1} - \frac{1}{3} \cdot (b_{i+1} - b_i)$
b$_\text{i}$
$b_{i+1}$
$b_{i-1}$
$b_{i+2}$

Construction of interior control points

Bezier curve segment constructed

**Figure B.41** Constructing a Bezier segment that interpolates points

## B.5 Randomness

Introducing controlled randomness in both modeling and animation can often produce more interesting, realistic, natural-looking imagery. The use of noise and turbulence functions are often used in textures but also can be used in modeling natural phenomena such as smoke and clouds. The code for noise and turbulence that follows is from Peachey's chapter in Ebert [3]. Random perturbations are also useful in human-figure animation to make the motion less “robotic” looking. There are various algorithms proposed in the literature for generating random numbers; Gasch’s [6] is presented at the end of this section.

### B.5.1 Noise

The *noise* function uses a table of pseudorandom numbers between –1 and +1 to represent the integer lattice values. The table is created by *valueTableInit* the first time that *noise* is called. Lattice coordinates are used to index into a table of pseudorandom numbers. A simple function of the coordinates, such as their sum, is used to compute the index. However, this can result in unwanted patterns. To help avoid these artifacts, a table of random permutation values is used to modify the index before it is used. A four-point spline is used to interpolate among the lattice pseudorandom numbers (*FPspline*).

```
#define TABSIZE            256
#define TABMASK            (TABSIZE-1)
#define PERM(x)            perm[(x)&TABMASK]
#define INDEX(ix,iy,iz)    PERM((ix)+PERM((iy)+PERM(iz)))
#define FLOOR(x)           (int)(x)

/* PERMUTATION TABLE */
static unsigned char perm[TABSIZE] = {
225, 155, 210, 108, 175, 199, 221, 144, 203, 116, 70, 213, 69, 158, 33,
252, 5, 82, 173, 133, 222, 139, 174, 27, 9, 71, 90, 246, 75, 130, 91,
191, 169, 138, 2, 151, 194, 235, 81, 7, 25, 113, 228, 159, 205, 253,
134, 142, 248, 65, 224, 217, 22, 121, 229, 63, 89, 103, 96, 104, 156,
17, 201, 129, 36, 8, 165, 110, 237, 117, 231, 56, 132, 211, 152, 20,
181, 111, 239, 218, 170, 163, 51, 172, 157, 47, 80, 212, 176, 250, 87,
49, 99, 242, 136, 189, 162, 115, 44, 43, 124, 94, 150, 16, 141, 247, 32,
10, 198, 223, 255, 72, 53, 131, 84, 57, 220, 197, 58, 50, 208, 11, 241,
28, 3, 192, 62, 202, 18, 215, 153, 24, 76, 41, 15, 179, 39, 46, 55, 6,
128, 167, 23, 188, 106, 34, 187, 140, 164, 73, 112, 182, 244, 195, 227,
13, 35, 77, 196, 185, 26, 200, 226, 119, 31, 123, 168, 125, 249, 68,
183, 230, 177, 135, 160, 180, 12, 1, 243, 148, 102, 166, 38, 238, 251,
37, 240, 126, 64, 74, 161, 40, 184, 149, 171, 178, 101, 66, 29, 59, 146,
61, 254, 107, 42, 86, 154, 4, 236, 232, 120, 21, 233, 209, 45, 98, 193,
114, 78, 19, 206, 14, 118, 127, 48, 79, 147, 85, 30, 207, 219, 54, 88,
234, 190, 122, 95, 67, 143, 109, 137, 214, 145, 93, 92, 100, 245, 0,
216, 186, 60, 83, 105, 97, 204, 52
};
```

```
#define RANDNBR        (((float)rand())/RAND_MASK)

float  valueTab[TABSIZE];

/* ========================================================== */
/* VALUE TABLE INIT */
/* initialize the table of pseudorandom numbers */
void valueTableInit(int seed)
{
   float   *table = valueTab;
   int     i;

   srand(seed);
   for (i=0; i<TABSIZE; i++)
   *(table++) = 1.0 -2.0*RANDNBR;
}

/* ========================================================== */
/* LATTICE function */
/* returns a value corresponding to the lattice point */
float lattice(int ix, int iy, int iz)
{
   return valueTab[INDEX(ix,iy,iz)];
}

/* ========================================================== */
/* NOISE function */
float noise(float x, float y, float z)
{
   int   ix,iy,iz;
   int   i,j,k;
   float fx,fy,fz;
   float xknots[4],yknots[4],zknots[4];
   static int initialized = 0;

   if (!initialized) {
      valueTableInit(665);
      initialized = 1;
   }

   ix = FLOOR(x);
   fx = x - ix;
   iy = FLOOR(y);
   fy = y - iy;
   iz = FLOOR(z);
   fz = z-iz;

   for (k=-1; k<=2; k++) {
      for (j=-1; j<=2; j++) {
         for (i=-1; i<=2; i++)
            xknots[i+1] = lattice(ix+i,iy+j,iz+k);
         yknots[j+1] = spline(fx,xknots);
      }
      zknots[k+1] = spline(fy,yknots);
   }
   return spline(fz,zknots);
}

#define   FP00  -0.5
#define   FP01   1.5
#define   FP02  -1.5
#define   FP03   0.5
#define   FP10   1.0
#define   FP11  -2.5
#define   FP12   2.0
#define   FP13  -0.5
#define   FP20  -0.5
#define   FP21   0.0
#define   FP22   0.5
#define   FP23   0.0
#define   FP30   0.0
#define   FP31   1.0
#define   FP32   0.0
#define   FP33   0.0

/* ========================================================== */
float spline(float u,float *knots)
{
   float   c3,c2,c1,c0;

   c3 = FP00*knots[0] + FP01*knots[1] + FP02*knots[2] + FP03*knots[3];
   c2 = FP10*knots[0] + FP11*knots[1] + FP12*knots[2] + FP13*knots[3];
   c1 = FP20*knots[0] + FP21*knots[1] + FP22*knots[2] + FP23*knots[3];
   c0 = FP30*knots[0] + FP31*knots[1] + FP32*knots[2] + FP33*knots[3];

   return ((c3*u + c2)*u + c1)*u + c0;
}
```

## B.5.2 Turbulence

Turbulence is a stochastic function with a “fractal” power spectrum [3]. The function is a sum of amplitude-varying frequencies. As frequency increases, the amplitude decreases.

```
/* TURBULENCE */
float   turbulence (float x, float y, float z)
{
   float   f;
   float   value = 0;
   for (f = MINFREQ; f < MAXFREQ;  f *= 2)
      value += fabs(noise(x*f, y*f, z*f))/f;
   return value;
}
```

## B.5.3 Random Number Generator

This random number generator returns a random number in the range 0 to 999999999. An auxiliary routine maps this number into an arbitrary range of integers.

```
int r[100];          /* “global” pseudo-random table -- */
                     /* must be visible to rand and init_rand */

/* ========================================================== */
/* RAND */
/* return a random number in the range 0 to 999999999 */
 int rand (void)
 {
   int i = r[98];
   int j = r[99];
   int k;
   int t;

   if ((t = r[i] - r[j]) < 0) t += 1000000000L;

   r[i] = t;

   r[98]--; r[99]--;
      if (r[98] == 0) r[98] = 55;
      if (r[99] == 0) r[99] = 55;

   k = r[100] % 42 + 56;
   r[100] = r[k];
   r[k] = t;

   return(r[100]);
 }

/* ========================================================== */
/* INIT RAND */
/* seed the random number table */
int init_rand (char *seed)
{
   char buf[101];
   int i, j, k;

   if (strlen(seed) > 85) return(0);
   sprintf(buf, “aEbFcGdHeI%s”, seed);
   while (strlen(buf) < 98) strcat(buf, “Q”);

   for (i = 1; i < 98; i++)
      r[i] = buf[i] * 8171717 + i * 997;

   i = 97; j = 12;
   for (k = 1; k < 998; k++) {
      r[i] -= r[j];
      if (r[i] < 0) r[i] += 1000000000;

      i--; j--;
      if (i == 0) i=97;
      if (j == 0) j=97;
   }

   r[98] = 55;
   r[99] = 24;
   r[100] = 77;
}

/* ========================================================== */
/* RAND INT */
/* return a random int between a and b */
/* assumes init_rand already called.  */
int rand_int(int a, int b)
{
   return (a + rand() % (b - a + 1));
}
```

## B.6 Physics Primer

Physically based motion is a limited simulation of physical reality. This can be as simple or as complex as the implementation requires. Below are some of the equations of importance in simple physics simulation that can be found in any one of several standard texts. *The Mechanical Universe* [5] is used as the source for the brief discussion that follows.

### B.6.1 Position, Velocity, and Acceleration

The fundamental equation relating position, distance, and speed is shown in Equation B.86. This can be used to control the positioning of an object for any particular frame of the animation because the frame number is tied directly to time (Equation B.87). The average velocity of a body is the distance moved divided by the time it took to move, as stated by Equation B.88. Notice that the unit of velocity is distance per time, for example, feet/second.

$$\text{distance} = \text{speed} \cdot \text{time} \qquad \textbf{(Eq. B.86)}$$

$$\text{time} = \text{frameNumber} \cdot timePerFrame \qquad \textbf{(Eq. B.87)}$$

$$\text{averageVelocity} = \text{distanceTraveled/time} \qquad \textbf{(Eq. B.88)}$$

For this discussion, distance as a function of time is $s(t)$; the average velocity from time $t1$ to $t2$ is $(s(t2) - s(t1))/(t2 - t1)$. The instantaneous velocity is determined by moving $t2$ closer and closer to $t1$. In the limiting case this becomes the derivative of the distance function with respect to time. Similarly, the average acceleration of an object is the change in velocity divided by the time it took to effect the change. This is presented in Equation B.89, where $v(t)$ is a function that gives the velocity of the object at time $t$. Notice that the unit of acceleration is velocity per time or distance per time per time, for example, feet/second$^2$. In the same way, instantaneous acceleration is the derivative of $v(t)$ with respect to time (Equation B.90). In the case of motion due to gravity, $g$ is the acceleration due to gravity—a constant that has been measured to be 32 feet/second$^2$ or 9.8 meters/second$^2$ (Equation B.91)

$$\text{average\_acceleration} = (v(t2) - v(t1))/(t2 - t1) \qquad \textbf{(Eq. B.89)}$$

$$a(t) = v'(t) = s''(t) \qquad \textbf{(Eq. B.90)}$$

$$
\begin{aligned}
a(t) &= g \\
v(t) &= g \cdot t \\
s(t) &= (1/2) \cdot g \cdot t^2
\end{aligned}
\qquad \text{(Eq. B.91)}
$$

## B.6.2 Circular Motion

Circular motion is important in physics and arises for a variety of phenomena, including the movement of planets and robotic armatures. Circular motion is easily specified by using polar coordinates. The position of a particle orbiting the origin at a distance $r$ can be described using Equation B.92. Here, $i$ and $j$ are orthonormal unit vectors (at right angles to each other and unit length) and $p(t)$ is the positional vector of the particle. In a constant radius circular orbit, $\theta(t)$ varies as a function of time, and the distance $r$ is constant. During uniform circular motion, $\theta(t)$ changes at a constant rate, and *angular velocity* is said to be constant. *Angular velocity* is referred to here as $\Omega(t)$ (Equation B.93). As for constant-velocity linear motion, in which the distance equals speed multiplied by time, for constant angular velocity the angle equals angular velocity multiplied by time (Equation B.94). If $\theta(t)$ is measured in radians and time in seconds, then $\omega(t)$ is measured in radians per second. To simplify the following equations, the functional dependence on time will often be omitted when the dependence is obvious from the context.

$$p(t) = (r \cdot \cos(\theta(t)))i + (r \cdot \sin(\theta(t)))j \qquad \text{(Eq. B.92)}$$

$$\frac{d}{dt}\theta(t) = \omega(t) \qquad \text{(Eq. B.93)}$$

$$\theta(t) = \omega \cdot t \qquad \text{(Eq. B.94)}$$

Taking the derivative of Equation B.92 with respect to time gives the instantaneous velocity (Equation B.95). Notice that the velocity vector, $v(t)$, is perpendicular to the position vector $p(t)$. This can be demonstrated by taking the dot product of the two vectors $v(t)$ and $p(t)$ and showing that it is identically zero.

$$v(t) = \frac{dp}{dt} = (-r \cdot \omega \cdot \sin(\omega \cdot t))i + (r \cdot \omega \cdot \cos(\omega \cdot t))j \qquad \text{(Eq. B.95)}$$

Computing the length of $v(t)$ shows that $|v|\ \ r \cdot \omega$ and, therefore, that the velocity is independent of $t$ (i.e., constant). Notice, however, that a constant circular motion still gives rise to an acceleration. Taking the derivative of Equation B.95 produces Equation B.96, which is called the centripetal acceleration. The centripetal acceleration resulting from uniform circular motion is directed radially inward

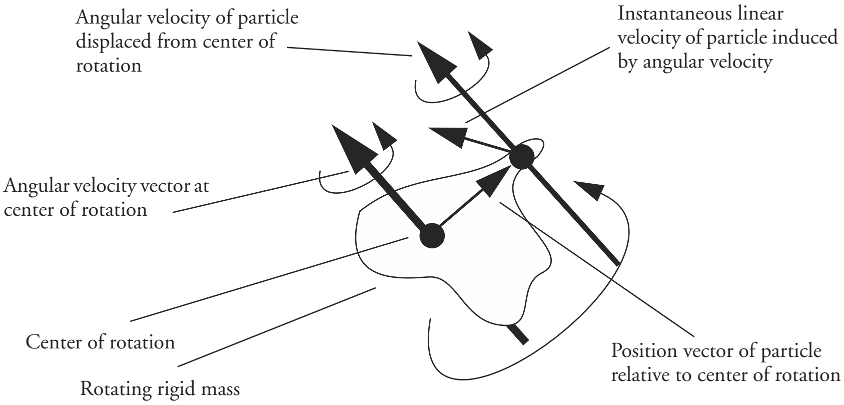

Angular velocity of particle displaced from center of rotation
Instantaneous linear velocity of particle induced by angular velocity
Angular velocity vector at center of rotation
Center of rotation
Rotating rigid mass
Position vector of particle relative to center of rotation

**Figure B.42** Motion of particle in a rotating rigid mass

and has constant magnitude. With the equation for the length of $v(t)$ from above, the magnitude of the acceleration can be written using Equation B.97.

$$a(t) = \frac{dv}{dt} = -\omega^2 \cdot ((r \cdot \cos(\omega \cdot t)) \cdot i + (r \cdot \sin(\omega \cdot t)) \cdot j)$$

$$= (-\omega)^2 \cdot r \qquad \text{(Eq. B.96)}$$

$$a = v^2 / r \qquad \text{(Eq. B.97)}$$

For any particle in a rigid mass undergoing a rotation, that particle is undergoing the same rotation about its own center. In addition, if the particle is displaced from the center of rotation, then it is also undergoing an instantaneous positional translation as a result of its circular motion (Figure B.42).

## B.6.3 Newton's Laws of Motion

It is useful to review Newton's laws of motion. The first law is the principle of inertia. The second law relates force to the acceleration of a mass (Equation B.98). In another form, this law relates force to change in momentum (Equation B.99), where momentum is mass times velocity ($m \cdot v$). The third law states that when an object pushes with a force on another object, the second object pushes back with an equal but opposite force. It is important to note that the force $F$ used here is considered to be the sum of all external forces acting on an object. Force is a vector quantity, and these equations really represent three sets of equations, one for each coordinate (Equation B.100). Newton's laws of motion are stated as follows:

*First Law: If no force is acting on an object, then it will not accelerate. It will maintain a constant velocity.*

*Second Law: The change of motion of an object is proportional to the forces applied to it.*

*Third Law: To every action there is always opposed an equal and opposite reaction.*

$$F = m \cdot a \tag{Eq. B.98}$$

$$F = \frac{d}{dt}((m \cdot v)) \tag{Eq. B.99}$$

$$\begin{aligned} F_x &= m \cdot a_x \\ F_y &= m \cdot a_y \\ F_z &= m \cdot a_z \end{aligned} \tag{Eq. B.100}$$

## B.6.4 Inertia and Inertial Reference Frames

An *inertial frame* is a frame of reference (e.g., a local coordinate system of an object) in which the principle of inertia holds. Any frame that is not accelerating is an inertial frame. In an inertial frame one observes the laws of motion and has no way of determining whether one is at rest or moving in an "absolute" sense (but then, what is "absolute"?).

## B.6.5 Center of Mass

The center of mass of an object is that point at which the object is balanced in all directions. If an external force is applied in line with the center of mass of an object, then the object moves as if all the mass were concentrated at the center ("*c*" in Figure B.43).

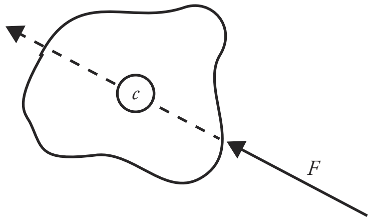
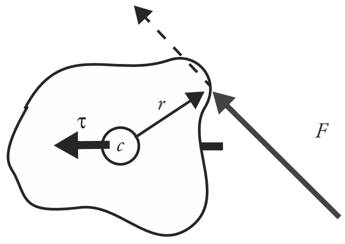


**Figure B.43** Center of mass



$r$ —vector from center of mass to point where force is applied
$F$—force vector
τ—torque vector

**Figure B.44** Applying a force off-center induces a torque

## B.6.6 Torque

The tendency of a force to produce circular motion is called *torque*. Torque is produced by a force applied off-center from the center of mass of an object (Figure B.44) and is computed by Equation B.101.

$$\tau = r \times F \tag{Eq. B.101}$$

It is important to note that *torque* refers to a specific axis of rotation and is perpendicular to both $r$ and $F$ in Figure B.44. The same force can exert different torques about different axes of rotation if it is applied at different locations on an object. A given torque vector for a rigid body is position independent. That is, for any particle in a rigid mass, the particle is undergoing that same torque.

## B.6.7 Equilibrium: Balancing Forces

In the absence of acceleration, there is no change in the motion of an object (Newton's first law). In such cases, the net force and the torque acting on a body vanish (Equation B.102, Equation B.103). There may be forces present, but the vector sum is equal to zero.

$$\sum F = 0 \tag{Eq. B.102}$$

$$\sum \tau = 0 \tag{Eq. B.103}$$

## B.6.8 Gravity

Equation B.104 is Newton's law of universal gravitation. It calculates the force of gravity between two masses ($m_1$ and $m_2$) whose centers of mass are a distance $r$ apart. $G$ is the universal gravitational constant equal to $6.67 \times 10^{-11}$ Newton meter$^2$/kilogram$^2$. When two objects are not touching, each acts on the other as if all its mass were concentrated at its center of mass. When an object is on (or near) the earth's surface, the distance of the object to the center of mass of the earth, the mass of the earth, and the gravitational constant of Equation B.104 can all be combined to produce the object's acceleration (Equation B.105). This acceleration is usually denoted as *g*.

$$F = G \cdot \frac{m_1 \cdot m_2}{r^2} \tag{Eq. B.104}$$

$$a = \frac{F}{m_{\text{object}}} = G \cdot \frac{m_{\text{earth}}}{(\text{radius}_{\text{earth}})^2} = 9.8\,\text{m/s}^2 = 32\,\text{f/s}^2 = g \tag{Eq. B.105}$$

## B.6.9 Centripetal force

Centripetal force is any force that is directed inward, toward the center of an object's motion (centripetal means *center seeking*). In the case of a body in orbit, gravity is the centripetal force that holds the body in the orbit. Consider a body, such as the moon, with mass $M_m$ and a distance $r$ away from the earth. The unit vector from the earth to the moon is $R$. The earth has a mass $M_e$. Equation B.106 calculates the force vector that results from gravity. The acceleration induced by a circular orbit always points toward the center (centripetal), as in Equation B.107. As previously shown, the acceleration due to circular motion has magnitude $v^2/r$ (Equation B.96). This can be used to solve for velocity (Equation B.108).

$$F = \left(-G \cdot \frac{M_m \cdot M_e}{r^2}\right) \cdot R \tag{Eq. B.106}$$

$$\begin{aligned} a &= F/M_m \\ a &= ((-G \cdot M_e)/r^2) \cdot R \end{aligned} \tag{Eq. B.107}$$

$$\begin{aligned} a &= (-v^2/r) \cdot R = ((-G \cdot M_e)/r^2) \cdot R \\ v &= \sqrt{(G \cdot M_e)/r} \end{aligned} \tag{Eq. B.108}$$

## B.6.10 Contact Forces

Gravity is one of the four fundamental forces (gravity, electromagnetic, strong, and weak) that act at a distance. Another important category is *contact forces*. The tension in a wire or rope is an example of a contact force, as is the compression in a rigid rod. These forces arise from the complex interactions of electric forces that tend to keep atoms a certain distance apart. The empirical law that governs such forces, and that is familiar when dealing with springs, is *Hooke's law* (Equation

B.109). Variable *x* is the change from the equilibrium length of the spring, *k* is the spring constant, and *F* is the restoring force of the spring to return to rest length. The constant *k* is a measure of the stiffness of the spring; the larger *k* is, the more sensitive the spring is to motion away from the rest position.

$$F = -k \cdot x \tag{Eq. B.109}$$

Another example of contact force, known as the *normal force*, is the result of repulsion between any two objects pressed against each other. It arises from the repulsion of the atoms of the two objects. It is always perpendicular to the surfaces, and its magnitude is proportional to how hard the two objects are pressed against each other. When objects in motion come in contact, an *impulse force* due to collision is produced. The impulse force is a short-duration force that is applied normal to the surface of contact on each of the two objects. Calculation of the impulse force due to collision is discussed in Chapter 4. Other important examples of contact forces are friction and viscosity.

### Friction

Friction arises from the interaction of surfaces in contact. It can be thought of as resulting from the multitude of collisions of molecules caused by the unevenness of the surfaces at the microscopic level.

The frictional force works against the relative motion of the two objects. Frictional forces from a surface do not exceed an amount proportional to the normal force exerted by the surface on the object. This is stated in Equation B.110, where *s* is the coefficient of static friction and $f_N$ is the normal force (component of the force that is perpendicular to the surface). Variable *s* varies according to the two materials that are in contact.

$$f_s = s \cdot f_N \tag{Eq. B.110}$$

The frictional forces acting between surfaces at rest with respect to each other are called *forces of static friction.* For a force applied to an object sitting on another object that is parallel to the surfaces in contact, there will be a specific force at which the block starts to slip. As the force increases from zero up to that threshold force, the lateral force is counteracted by an equal force of friction in the opposite direction. Once the object begins to move, *kinetic friction* acts on the object and approximately obeys the empirical law of Equation B.111, where *k* is the coefficient of kinetic friction and $f_N$ is the force normal to the surface. Kinetic friction is typically less than static friction. The force of kinetic friction is always opposite to the velocity of the object.

$$f_k = k \cdot f_N \tag{Eq. B.111}$$

### Viscosity

The resistive force of an object moving in a medium is *viscosity.* It is another contact force and is extremely difficult to model accurately. When an object moves at low velocity through a fluid, the viscosity is approximately proportional to the velocity (Equation B.112); $K$ is the constant of proportionality, which depends on the size and the shape of the object, and $n$ is the coefficient of viscosity, which depends on the properties of the fluid. The coefficient of viscosity, $n$, decreases with increasing temperature for liquids and increases with temperature for gases. Stokes's law for a sphere of radius $R$ is given in Equation B.113.

$$F_{\text{vis}} = -K \cdot n \cdot v \qquad \textbf{(Eq. B.112)}$$

$$K = 6 \cdot \pi \cdot R \qquad \textbf{(Eq. B.113)}$$

An object dropping through a liquid attains a constant speed, called the limiting or terminal velocity, at which gravity, acting downward, and the viscous force, acting upward, balance each other and there is no acceleration (e.g., Equation B.114 for a sphere). Terminal velocity is given by Equation B.115. In a viscous medium, heavier bodies fall faster than lighter bodies. For spherical objects falling at a low velocity in a viscous medium, not necessarily at terminal velocity, momentum is given by Equation B.116.

$$m \cdot g = 6 \cdot \pi \cdot R \cdot n \cdot v \qquad \textbf{(Eq. B.114)}$$

$$v = (m \cdot g)/(6 \cdot \pi \cdot R \cdot n) \qquad \textbf{(Eq. B.115)}$$

$$m \cdot \frac{dv}{dt} = m \cdot g - 6 \cdot \pi \cdot R \cdot n \cdot v \qquad \textbf{(Eq. B.116)}$$

## B.6.11 Centrifugal Force

Consider an object (a frame of reference) that rotates in uniform circular motion with respect to a post (an inertial frame) because it is held at a constant distance by a rope. Relative to the inertial frame, each point in the uniformly rotating frame has centripetal acceleration expressed by Equation B.117; $r$ is the distance of the point from the axis of rotation, $R$ is the unit vector from the inertial frame to the rotating frame, and $v$ is the speed of the point. The tension in the rope supplies the force necessary to produce the centripetal acceleration. Relative to the rotating frame (not an inertial frame), the frame itself does not move and therefore the *centrifugal* force necessary to counteract the force supplied by the rope is calculated by using Equation B.118.

$$a = -\frac{v^2}{r} \cdot R \qquad \textbf{(Eq. B.117)}$$

$$F_c = -(m \cdot a) = -\frac{m \cdot v^2 \cdot R}{r} \qquad \text{(Eq. B.118)}$$

## B.6.12 Work and Potential Energy

For a constant force of magnitude *F* moving an object a distance *h* parallel to the force, the work *W* performed by the force is shown in Equation B.119. If a mass *m* is lifted up so that it does not accelerate, then the lifting force is equal to the weight (mass times gravitational acceleration) of the object. Since the weight is constant, the work done to raise the object up to a height *h* is presented in Equation B.120. Energy that a body has by virtue of its location is called *potential energy*. The work in this case is converted into potential energy.

$$W = F \cdot h \qquad \text{(Eq. B.119)}$$

$$W = m \cdot g \cdot h \qquad \text{(Eq. B.120)}$$

## B.6.13 Kinetic Energy

Energy of motion is called *kinetic energy* and is shown in Equation B.121. The velocity of a falling body that started at a height *h* is calculated by Equation B.122. Its kinetic energy is therefore calculated by Equation B.123.

$$K = \frac{1}{2} \cdot m \cdot v^2 \qquad \text{(Eq. B.121)}$$

$$v^2 = 2 \cdot g \cdot h \qquad \text{(Eq. B.122)}$$

$$K = \frac{1}{2} \cdot m \cdot v^2 = m \cdot g \cdot h \qquad \text{(Eq. B.123)}$$

## B.6.14 Conservation of Energy

Potential plus kinetic energy of a closed system is conserved (Equation B.124). This is useful, for example, when solving for an object's current height (current potential energy) when its initial height (initial potential energy), initial velocity (initial kinetic energy), and current velocity (current kinetic energy) are known.

$$U_a + K_a = U_b + K_b \qquad \text{(Eq. B.124)}$$

## B.6.15 Conservation of Momentum

In a closed system, total momentum (mass times velocity) is conserved. This means that it does not change (Equation B.125) and that the momenta of all objects in a closed system always sum to the same amount (Equation B.126). This is useful, for example, when solving for velocities after a collision when the velocities before the collision are known.

$$\frac{d}{dt}((m_1 \cdot v_1 + m_2 \cdot v_2 + \ldots + m_n \cdot v_n)) = 0 \qquad \text{(Eq. B.125)}$$

$$m_1 \cdot v_1 + m_2 \cdot v_2 + \ldots + m_n \cdot v_n = \text{constant} \qquad \text{(Eq. B.126)}$$

## B.6.16 Oscillatory Motion

In some systems, the stability of an object is subject to a linear restoring force, *F*. The force is linearly proportional (using *k* as the constant of proportionality) to the distance *x* the object has been displaced from its equilibrium position (Equation B.127). Associated with this force is the potential energy (Equation B.128) that results from the object's displacement. These systems have the property that, if they are disturbed from equilibrium, the restoring force that acts on them tends to move them back into equilibrium. When disturbed from equilibrium, they tend to overshoot that point when they return, due to inertia. Then the restoring force acts in the opposite direction, trying to return the system to equilibrium. The result is that the system oscillates back and forth like a mass on the end of a spring or the weight at the end of a pendulum.

$$F = -k \cdot x \qquad \text{(Eq. B.127)}$$

$$U = \frac{1}{2} \cdot k \cdot x^2 \qquad \text{(Eq. B.128)}$$

Combining the basic equations of motion (force equals mass times acceleration) with Equation B.127, one can derive the differential equation for oscillatory motion. This equation is satisfied by the displacement function $x(t)$ (Equation B.129).

$$F = m \cdot a$$

$$F = -k \cdot x$$

$$a = \frac{d^2x}{dt}$$

$$m \cdot \frac{d^2x}{dt} = -k \cdot x$$

$$\frac{d^2x}{dt} = (-k \cdot x)/m \qquad \text{(Eq. B.129)}$$

## B.6.17 Damping

The damping force can be modeled after Stokes's law, in which resistance is assumed to be linearly proportional to velocity (Equation B.130). This is usually valid for oscillations of sufficiently small amplitude. The damping force opposes the motion. The constant, $k_d$, is called the *damping coefficient*. Adding the damping force to the spring force produces Equation B.131. Dividing through by $m$ and collecting terms results in Equation B.132, where $b = k_d / m$ and $a^2 = k / m$.

$$F_d = -k_d \cdot \frac{dx}{dt} \tag{Eq. B.130}$$

$$m \cdot \frac{d^2x}{dt} = -k \cdot x - \left(k_d \cdot \frac{dx}{dt}\right) \tag{Eq. B.131}$$

$$\frac{d^2x}{dt} + b \cdot \frac{dx}{dt} + a^2 \cdot x = 0 \tag{Eq. B.132}$$

If there is a spring force but no damping, the general solution can be written as $x = C \cdot \cos(a \cdot t + \theta_0)$. If there is damping but no spring force, the general solution turns out to be $x = C \cdot e^{-b \cdot t} + D$, with $C$ and $D$ constant. If both the spring force and the damping force are present, the solution takes the form shown in Equation B.133.

$$\begin{aligned} &x = C \cdot e^{-b \cdot t} \cdot \cos(d \cdot t + \theta_0) \\ &\text{where } d = \sqrt{a^2 - \frac{b^2}{4}} \\ &\text{for } b < 2 \cdot a \end{aligned} \tag{Eq. B.133}$$

## B.6.18 Angular Momentum

Angular momentum is the rotational equivalent of linear momentum and can be computed by Equation B.134, where $r$ is the vector from the center of rotation and $p$ is momentum (mass times velocity). The temporal rate of change of angular momentum is equal to torque (Equation B.135). Angular momentum, like linear momentum, is conserved in a closed system (Equation B.136).

$$L = r \times p \tag{Eq. B.134}$$

$$\tau = \frac{dL}{dt} \tag{Eq. B.135}$$

$$\sum L_i = \text{constant} \qquad \text{(Eq. B.136)}$$

## B.6.19 Inertia Tensors

An *inertia tensor*, or *angular mass*, describes the resistance of an object to a change in its angular momentum [5][13]. It is represented as a matrix when the angular mass is related to the principal axes of the object (Equation B.137). The terms of the matrix describe the distribution of the mass of the object relative to a local coordinate system (Equation B.138). For objects that are symmetrical with respect to the local axes, the off-diagonal elements are zero (Equation B.139). For a rectangular solid of dimensions *a, b,* and *c* along its local axes, the inertia tensor at the center of mass is given by Equation B.140. For a sphere with radius *R,* the inertia tensor is given by Equation B.141. If the inertia tensor is known at one position by *I,* then the inertia tensor *I′* for parallel axes at a new position (*X, Y, Z*) relative to the original position is calculated by Equation B.142. If the inertia tensor for one set of axes is known by *I,* then the inertia tensor for a rotated frame is calculated by Equation B.143, where *M* is the rotation matrix describing the rotated frame relative to the original frame.

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \qquad \text{(Eq. B.137)}$$

$$\begin{aligned} I_{xx} &= \int (y^2 + z^2)\, dm \\ I_{yy} &= \int (x^2 + z^2)\, dm \\ I_{zz} &= \int (x^2 + y^2)\, dm \\ I_{xy} &= \int xy dm \\ I_{xz} &= \int xz dm \\ I_{yz} &= \int yz dm \end{aligned} \qquad \text{(Eq. B.138)}$$

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \qquad \text{(Eq. B.139)}$$

$$I = \frac{1}{12} \cdot \begin{bmatrix} M \cdot (b^2 + c^2) & 0 & 0 \\ 0 & M \cdot (a^2 + c^2) & 0 \\ 0 & 0 & M \cdot (a^2 + b^2) \end{bmatrix} \tag{Eq. B.140}$$

$$I = \frac{2 \cdot M \cdot R^2}{5} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{Eq. B.141}$$

$$I' = \frac{1}{12} \cdot \begin{bmatrix} I_{xx} + M \cdot (Y^2 + Z^2) & -I_{xy} - (M \cdot X \cdot Y) & -I_{xz} - (M \cdot X \cdot Z) \\ -I_{xy} - (M \cdot X \cdot Y) & I_{yy} + M \cdot (X^2 + Z^2) & -I_{yz} - (M \cdot XY \cdot Z) \\ -I_{xz} - (M \cdot X \cdot Z) & -I_{yz} - (M \cdot XY \cdot Z) & I_{zz} + M \cdot (X^2 + Y^2) \end{bmatrix} \tag{Eq. B.142}$$

$$I' = M \cdot I \cdot M^{-1} \tag{Eq. B.143}$$

## B.7 Numerical Integration Techniques

Numerical integration is useful for finding the arc length of the curve or for updating the position of an object over time. A useful technique for the former is Gaussian quadrature; for the latter, the Runge-Kutta method. As with many numerical techniques, Press et al. [16] is extremely valuable.

### B.7.1 Function Integration

Given a function *f*(*x*), Gaussian quadrature can be used to produce an arbitrarily close approximation to the integral of the function between two values, $\int_a^b f(x)$, if the function is sufficiently well behaved [16]. Gaussian quadrature approximates the integral as a sum of weighted evaluations of the function at specific values (abscissas). The number of evaluations controls the error in the approximation. In its general form, Gaussian quadrature incorporates a multiplicative function, *W*(*x*), which can condition some functions for the approximation (Equation B.144). Gauss-Legendre integration is a special case in which *W*(*x*) = 1.0, and it results in Equation B.145. The code in Figure B.45 for *n* = 10 duplicates that used in Chapter 3 for computing arc length. Figure B.46 gives the code for computing Gauss-Legendre weights and abscissas for arbitrary *n*.

```
/* ------------------------------------------------------------------------
INTEGRATE FUNCTION
use gaussian quadrature to integrate square root of given function in the given
interval
*/
double integrate_func(polynomial_td *func,interval_td *interval)
{
   double   x[5]={.1488743389,.4333953941,.6794095682,.8650633666,.9739065285};
   double   w[5]={.2966242247,.2692667193,.2190863625,.1494513491,.0666713443};
   double   length, midu, dx, diff;
   int      i;
   double  evaluate_polynomial();
   double  u1,u2;

   u1 = interval->u1;
   u2 = interval->u2;

   midu = (u1+u2)/2.0;
   diff = (u2-u1)/2.0;
   length = 0.0;
   for (i=0; i<5; i++) {
      dx = diff*x[i];
      length += w[i]*(sqrt(evaluate_polynomial(func,midu+dx)) +
      sqrt(evaluate_polynomial(func,midu-dx)));
   }
   length *= diff;

   return (length);
}
```

**Figure B.45** Gauss-Legendre integration for $n = 10$

$$\int_a^b (W(x) \cdot f(x))\,dx \cong \sum_{i=1}^{n} w_i \cdot f(x_i) \qquad \textbf{(Eq. B.144)}$$

$$\int_a^b f(x)\,dx \cong \sum_{i=1}^{n} w_i \cdot f(x_i) \qquad \textbf{(Eq. B.145)}$$

## B.7.2 Integrating Ordinary Differential Equations

Integrating ordinary differential equations (ODEs) in computer animation typically means that the derivative function $f'(x)$ is available and that a numerical approximation to the function $f(x)$ is desired. For example, in a physically based simulation, the time-varying acceleration of an object is computed from the object's mass and the forces acting on the object in the environment. From the acceleration (the derivative function), the velocity (the function) is numerically

```
/* GAUSS-LEGENDRE */

#define  EPSILON  0.00000000001
/* calculate the weights and abscissas of the Gauss-Legendre n-point form *
void gaussWeights(float a, float b, float *x, float *w, int n)
{
   int       i,j,m;
   float     p1,p2,p3,p;
   float     z,z1;
   float     xave,xdiff;

   m = (n+1)/2;
   xave = (b+a)/2;
   xdiff = (b-a)/2;
   for (i = 0; i<m; i++) {
      z = cos(PI*((i+1)-0.25)/(n+0.5));
      do {
         p1 = 1.0;
         p2 = 0.0;
         for (j=0; j<n; j++) {
            p3 = p2;
            p2 = p1;
            p1 = ((2*(j+1) - 1.0)*z*p2-j*p3)/(j+1);
         }
         pp = n*(z*p1-p2)/(z*z-1);
         z1 = z;
         z = z1-p1/pp;
      } while (fabs(z-z1) > EPSILON);
      x[i] = xave - xdiff*z;
      x[n-1-i] = xave + xdiff*z;
      w[i] = 2.0*xdiff/((1.0-z*z)*pp*pp);
      w[n-l-i] = w[i];
   }
}
```

**Figure B.46** Computing Gauss-Legendre weights and abscissas

calculated over time. Similarly, once the time-based velocity function is known (the derivative function), the time-varying position of the object (the function) can be calculated numerically.

The simple form of an ordinary differential equation involves a first-order derivative of a function of a single variable. In addition, it is usually the case that conditions at an initial point in time are known and that the numerical integration is used in a simulation of a system as time moves forward. Such problems are referred to as *initial value problems.*

The *Euler method* is the most basic technique used for solving such simple ODE initial value problems. The Euler method is shown in Equation B.146, where $h$ is the time step such that $x_{n+1} = h + x_n$. This method is not symmetrical in that it uses information at the beginning of the time step to advance to the end of the time step. The derivative at the beginning of the time step produces the vector,

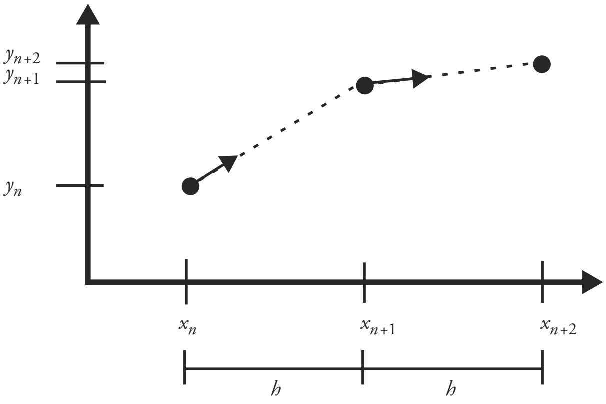


**Figure B.47** The Euler method

which is tangent to the curve representing the function at that point in time. The tangent at the beginning of the interval is used as a linear approximation to the behavior of the function over the entire interval (Figure B.47). The Euler method is neither stable nor very accurate. Other methods, such as Runge-Kutta, are more accurate for equivalent computational cost.

$$y_{n+1} = y_n + h \cdot f'(x_n, y_n) \quad \text{(Eq. B.146)}$$

*Runge-Kutta* is a family of methods that is symmetrical with respect to the interval. The second-order Runge-Kutta, or midpoint method, is presented in Chapter 4 in the discussion of physically based simulations. The most useful method is *fourth-order Runge-Kutta* [16], shown in Equation B.147 and Figure B.48. It is referred to as a fourth-order method because its error term is on the order of the interval $h$ to the fifth power. The advantage of using the method is that although each step requires more computation, larger step sizes can be used, resulting in an overall computational savings.

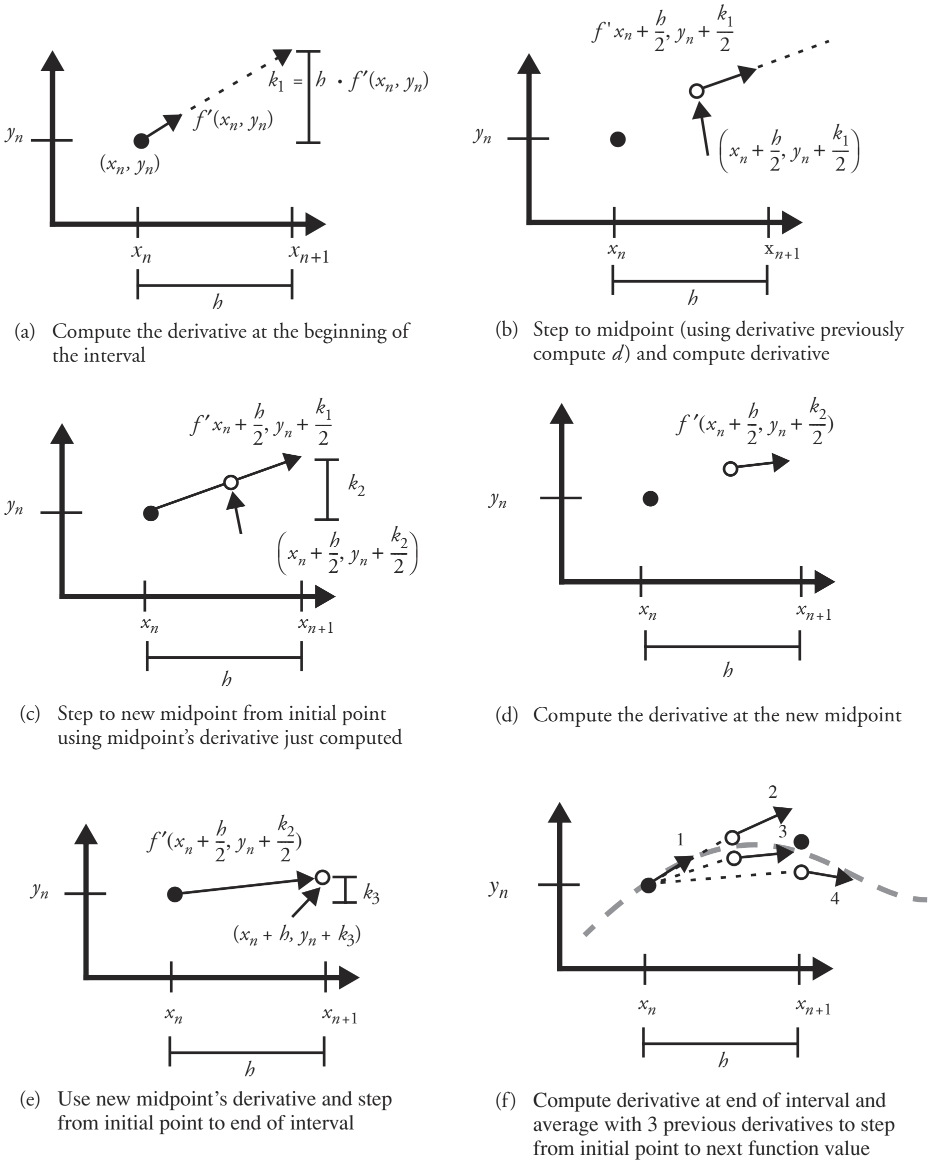
$$k_1 = h \cdot f'(x_n, y_n)$$

$$k_2 = h \cdot f'\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = h \cdot f'\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = h \cdot f'(x_n + h, y_n + k_3)$$

$$y_{n+1} = \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \quad \text{(Eq. B.147)}$$


$y_n$
$(x_n, y_n)$
$f'(x_n, y_n)$
$k_1 = h \cdot f'(x_n, y_n)$
$x_n$
$x_{n+1}$
$h$
$f' x_n + \frac{h}{2}, y_n + \frac{k_1}{2}$
$\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$
$\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$
$k_2$
$f'(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$
$(x_n + h, y_n + k_3)$
$k_3$
1
2
3
4

(a) Compute the derivative at the beginning of the interval

(b) Step to midpoint (using derivative previously compute *d*) and compute derivative

(c) Step to new midpoint from initial point using midpoint's derivative just computed

(d) Compute the derivative at the new midpoint

(e) Use new midpoint's derivative and step from initial point to end of interval

(f) Compute derivative at end of interval and average with 3 previous derivatives to step from initial point to next function value

**Figure B.48** Fourth-order Runge-Kutta

## B.8 Standards for Moving Pictures

When producing computer animation, one must decide what format to use for storing the sequence of images. Years ago the images were captured on film either by taking pictures of refresh vector screens or by plotting the images directly onto the film. This was a long and expensive process requiring single-frame film cameras and the developing of nonreusable film stock. The advent of frame buffers and video encoders made recording on videotape a convenient alternative. With today's cheap disks, memory, and CPU cycles, all-digital desktop video production is a reality. While the entertainment industry is still based on film, most of the rest of computer animation is produced using digital images intended to be displayed on a raster-scan device in such forms as broadcast video, DVD video, animated Web banners, and streaming video. This section is intended to give the reader some idea of the various standards used for recording moving pictures. Standards related to the digital image are emphasized.

### B.8.1 In the Beginning, There Was Analog

Before the rise of digital video, the two common formats for moving pictures were film and (analog) video. Over the years there have been almost a hundred film formats. The formats differ in the size of the frame, in the placement, size, and number of perforations, and in the placement and type of audio tracks [15]. Silent film is played at 16 frames per second (fps). Some sound film is played at 18 fps, but 24 fps is more common. Film played at 24 fps is typically doubly projected; that is, each frame is displayed twice to reduce the effects of flicker.

Some of the film sizes (widths) are listed in Table B.1. Note that there are often several formats for each film size. Only the most popular film formats are listed here. See the Web pages of Norwood [15] and Rogge [19] for more information. With the rise of desktop video production, film is less of an issue for home-brew computer animation, although it remains useful for conventional animation and, of course, is still the standard medium for display of feature-length films in theaters, although even this is starting to change.

#### Broadcast Video Standard

In 1941 NTSC established 525-line, 60.00-Hz field rate, 2:1 interlaced monochrome television in the United States. In 1953, 525-line, 59.94-Hz field rate, 2:1 interlaced, composite color television signals were established as a standard. The image is displayed top to bottom, with each scanline displayed left to right.

**Table B.1** Film Formats

| Film Width | Notes |
|---|---|
| 8mm | An old format, introduced in 1932, 8mm is used for inexpensive home movies. Cameras for regular 8mm are no longer manufactured. Regular 8mm uses 16mm stock, which is recorded on both sides after flipping the film in the camera. This allows the 16mm film stock to be split down the middle to produce two 8mm reels. The frame is 0.192″ x 0.145″. Super-8 was introduced in 1965 as an improvement over regular 8mm. The perforations (the holes in the film stock used to advance and register the film) were made smaller and the frame size was increased to 0.224″ x 0.163″. The film was placed into cassettes instead of on the reels of regular 8mm film. |
| 16mm | 16mm is used for television and low-budget theatrical productions. It was introduced in 1923 and has a frame size of 0.404″ x 0.295″. |
| 35mm | 35mm has been a standard film size since the turn of the twentieth century [19]. It first became popular because it could be derived from the original 70mm film made by Kodak. It is the standard for theatrical work as well as television. The *standard Academy frame,* the most popular of several 35mm formats, is 0.864″ x 0.630″. |
| 65mm | 65mm is gaining in popularity in venue rides and IMAX theaters. It is run through the projector sideways at 48 fps. The IMAX camera frame is 2.772″ x 2.072″. |
| 70mm | 70mm film is often a blowup print of 35mm film, produced for improved audio, better registration, and less grain of the release print. With better sound technology (e.g., digital) and the advent of multiplex theaters with smaller screen sizes, there is less demand for 70mm film. The frame size is 1.912″ x 0.870″. |

*2:1 interlaced* refers to the scanning pattern, with the information on the odd scanlines followed by the information on the even scanlines. Each set of scanlines is referred to as a *field;* there are two fields per *frame*. This standard is typically referred to by the initials of the committee—NTSC. Broadcast video in the United States must correspond to this standard. The standard sets a specific duration for a horizontal scanline, a frame time, the amplitude and duration of the various synch pulses, and so on. Home video recording units typically generate much sloppier signals and would not qualify for broadcast. There are encoders that can strip old synch signals off a video signal and reencode it so that it conforms to broadcast quality standards.

There are a total of 525 scanline times per frame time in the NTSC format. The number of frames transmitted per second is 29.97. There is a 2:1 interlace of the scanlines in alternate fields. Of the 525 total scanline times, approximately 480 contain picture information. The remainder of the scanline times are occupied by the overhead involved in the scanning pattern: the time it takes the beam to go

**Table B.2** Video Format Comparison [10]

| Standard | Lines | Scan Pattern | Field Rate (Hz) | Aspect Ratio |
|---|---|---|---|---|
| NTSC | 525 | 2:1 interlaced | 59.94 | 4:3 |
| PAL | 625 | 2:1 interlaced | 50 | 4:3 |
| SECAM | 625 | 2:1 interlaced | 50 | 4:3 |

from the end of one scanline to the beginning of the next and the time it takes for the beam to go from the bottom of the image to the top. The aspect ratio of a 525-line television picture is 4:3, so equal vertical and horizontal resolutions are obtained at a horizontal resolution of 480 times 4/3, or 640 pixels per scanline. PAL and SECAM are the other two standards in use around the world (Table B.2). They differ from NTSC in specifics like the number of scanlines per frame, the field rate, and the frequency of the color subcarrier, but both are interlaced raster formats. One of the reasons that television technology uses interlaced scanning is that, when a camera is providing the image, the motion is updated every field, thus producing smoother motion.

### Black-and-White Signal

A black-and-white video signal is basically a single line that has the synch information and intensity signal (luminance) superimposed on one signal. The vertical and horizontal synch pulses are negative with respect to a reference level, with vertical synch being a much longer pulse than horizontal synch. On either side of the synch pulses are reference levels called the front porch and the back porch. The active scanline interval is the period between horizontal synch pulses. During the active scanline interval, the intensity of the signal controls the intensity of the electron beam of the monitor as it scans out the image.

### Incorporating Color into the Black-and-White Signal

When color came on the scene in broadcast television, engineers were faced with incorporating the color information in such a way that black-and-white television sets could still display a color signal and color sets could still display black-and-white signals. The solution is to encode color into a high-frequency component that is superimposed on the intensity signal of the black-and-white video.

A reference signal for the color component, called the *color burst,* is added to the back porch of each horizontal synch pulse, with a frequency of 3.58 Mz. The color is encoded as an amplitude and phase shift with respect to this reference signal. A signal that has separate lines for the color signals is referred to as a *component* signal. A signal such as the color TV signal with all of the information superimposed on one line is referred to as a *composite* signal.

Because of the limited room for information in the color signal of the composite signal, the TV engineers optimized the color information for a particular hue they considered most important: Caucasian skin tone. Because of that, the RGB information has to be converted into a different color space: YUV. *Y* is luminance and is essentially the intensity information found in the black-and-white signal. It is computed by Equation B.148.

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \qquad \textbf{(Eq. B.148)}$$

The *U* and *V* of television are scaled and filtered versions of *B-Y* and *R-Y,* respectively. *U* and *V* are used to modulate the amplitude and phase shift of the 3.58-Mz color frequency reference signal. The phase of this chroma signal, *C,* conveys a quantity related to hue, and its amplitude conveys a quantity related to color saturation. In fact, the *I* and *Q* stand for "in phase" and "quadrature," respectively. The NTSC system mixes *Y* and *C* together and conveys the result on one piece of wire. The result of this addition operation is not theoretically reversible; the process of separating luminance and color often confuses one for the other (e.g., the appearance of color patterns seen on TV shots of people wearing black-and-white seersucker suits).

### Videotape Formats

The size and speed of the tape and the encoding format contribute to the quality that can be supported by a particular video format. Common tape sizes are 1/2", 3/4", 1", and 2". The first two sizes are of cassette tapes, and the other two are of open reel tapes. One-half inch supports consumer-grade formats (e.g., VHS, S-VHS, and the less popular Betamax), 3/4" is industrial strength (U-Matic), and 1" and 2" are professional broadcast-quality formats. One inch is the newer technology and has replaced some 2" systems.

The common 1/2" video formats are VHS and S-VHS. S-VHS is a format in which the *Y* and *C* signals are kept separate when played back, thus avoiding the problems created when the signals are superimposed. All video equipment actually records signals this way, but S-VHS allows the *Y* signal (luminance) to be recorded at a higher resolution than color. The color information is recorded with the same fidelity as on VHS. In addition, the sound is encoded differently than on regular VHS, also resulting in greater fidelity. The advantages of S-VHS are especially pronounced when it is played back on an S-VHS-compatible television.

### High-Definition Television

The basic idea behind high-definition television (HDTV) is to increase the percentage of the visual field occupied by the image [12]. Most HDTV systems approximately double the number of pixels in both the horizontal and vertical directions over the current NTSC standard and change the aspect ratio from 4:3 to 16:9.

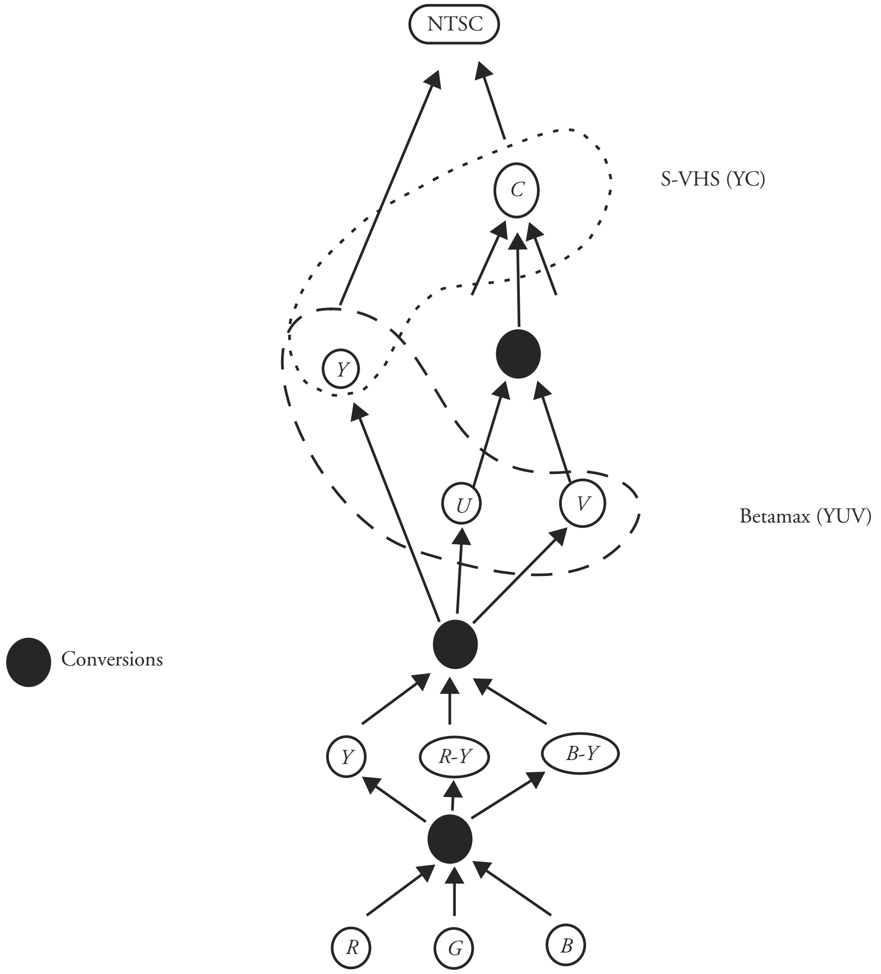


**Figure B.49** Video signal

## B.8.2 In the Digital World

Full-color (24 bits per pixel), video resolution (640x480), video rate (30 fps) animation requires approximately 1.6 gigabytes of information per minute of animation when stored as uncompressed RGB images. The problem is how to store and play back the animation. Various trade-offs must be considered when choosing a format, including the amount of compression, the time it takes for the compression/decompression, and the color and spatial resolution supported. The objective here is to provide an overview of the terminology, the important issues, and the most popular standards in recording animation. The discussion is at the level of consumer-grade technology and is not intended for professionals involved in the production of high-quality digital material.

*Digital video* sometimes refers specifically to digital versions of video to be broadcast for reception by television sets. In some literature this is also referred to as *digital TV* (*DTV*), which is the term used here. On the other hand, *digital video* (*DV*) is also used in the sense of computer-generated moving images intended to be played back on an RGB computer monitor. *DV* is used here to specifically denote material intended to be displayed by a computer. These two categories of digital representations, DTV and DV, have much in common. One of the most important common issues is the use of compression/decompression (codec) technology (described below). Standards related to DTV have two additional concerns. First, the standards are concerned with images that will be encoded for broadcast. As a result, they rarely deal directly with RGB images. When destined for television, digital images are at least partially encoded in a format related to the broadcast video standard (NTSC) soon after they leave the camera (e.g., YUV). Digital images synthesized for playback on a computer are typically generated as RGB images, compressed for storage and transmission, and then decompressed for playback on RGB monitors. Second, DTV standards are concerned with an associated recording format of the images and audio on tape, tape being the common storage medium for broadcast video studios. Because images for computer playback are typically stored digitally on a hard disk or removable disk, DV standards do not cover tape formats. However, there are DV standards for file formats and for digital movies. Digital movie formats organize the image and audio data into tracks with associated timing information.

### Compression/Decompression

The recent explosion in multimedia applications, especially as a result of the Web, has led to the development of a variety of compression/decompression schemes [25]. After the frames of an animation are computed, they are compressed and stored on a hard disk or CD-ROM. When playback of the animation is requested, the compressed animation is sent to the compute box, using disk I/O or over the Web, where the frames are decompressed and displayed. The decompression and playback can take place in real time as the data is being transmitted. In this case it is referred to as *streaming video.* If the decompression is not fast enough to support streaming video, the compressed animation is transmitted in its entirety to the compute box, decompressed into the complete animation, and then played back. In either case, the compression not only saves space on the storage device but also allows animation to be transferred to the computer much faster. Several of the codecs are proprietary and are used in workstation-based video products. The different schemes have various strengths and weaknesses and thus involve trade-offs, the most important of which are compression level, quality of video, and compression/decompression speed [21].

The amount of compression is usually traded off for image quality. With some codecs, the amount of compression can be set by a user-supplied parameter so that, with a trial-and-error process, the best compression level for the particular images at hand can be selected. Codecs with greater compression levels usually incorporate interframe compression as well as intraframe compression. *Intraframe compression* means that each frame is compressed individually. *Interframe compression* refers to the temporal compression possible when one processes a series of similar still images using techniques such as image differencing. However, when one edits a sequence, interframe compression means that more frames must be decompressed and recompressed to perform the edits.

The quality of the video after decompression is, of course, a big concern. The most fundamental feature of a compression scheme is whether it is *lossless* or *lossy.* With lossless compression, in which the final image is identical to the original, only nominal amounts of compression can be realized, usually in the range of 2:1. The codecs commonly used for video are lossy in order to attain the 500:1 compression levels necessary to realize the transmission speeds for pumping animations over the Web or from a CD-ROM. To get these levels of compression, the quality of the images must be compromised. Various compression schemes might do better than others with such image features as edges, large monochrome areas, or complex outdoor scenes. The amount of color resolution supported by the compression is also a concern. Some, such as animated GIF format, support only 8-bit color.

The compression and decompression speed is a concern for obvious reasons, but decompression speed is especially important in some applications, for example, streaming video. On the other hand, real-time compression is useful for applications that store compressed images as they are captured. If compression and decompression take the same amount of time, the codec is said to be *symmetric.* In some applications such as streaming video, it is acceptable to take a relatively long time for compression as long as the resulting decompression is very fast. In the case of unequal times for compression and decompression, the codec is said to be *asymmetric.* To attain acceptable decompression speeds on typical compute boxes, some codecs require hardware support.

A variety of compression techniques form the basis of the codec products. *Run-length encoding* is one of the oldest and most primitive schemes that have been applied to computer-generated images. Whenever a value repeats itself in the input stream, the string of repeating values is replaced by a single occurrence of the value along with a count of how many times it occurred. This was sufficient for early graphic images, which were simple and contained large areas of uniform color. The technique does not perform well with today's complex imagery. The *LZW* (Lempel-Ziv-Welch) technique was developed for compressing text. As the input is read in, a dictionary of strings and associated codes is generated and then used as the rest of the input is read in. *Vector quantization* simply refers to any scheme that uses a sample value to approximate a range of values. *YUV-9* is a technique in which the color is undersampling, so that, for example, a single color value is recorded for each 4x4 block of pixels. *DCT* (discrete cosine transform) is a very popular technique that breaks a signal into a sum of cosine functions of various frequencies at specific amplitudes. The signal can be compressed by throwing away low-amplitude and/or high-frequency components. DCT is an example of the more general *wavelet compression,* in which the form of the base component function can be selected according to its properties. *Fractal compression* is based on the fact that many signals are self-similar under scale. A section of the signal can be composed of transformed copies of other sections of the signal. This is applied to rectangular blocks of the image. In fact, most of the compression schemes break the image into blocks and then compress the blocks.

Codecs employ one or more of the techniques mentioned above. The names of some of the more popular codecs are Video I, RLE, GIF, Motion JPEG, MPEG, Cinepak, Sorenson, and Indeo 3.2. Microsoft products are Video I and RLE. Video I employs DCT compression, and RLE uses run-length encoding. GIF is an 8-bit color image compression scheme based on the LZW compression. JPEG uses DCT compression. Motion JPEG (MJPEG) is simply the application of JPEG for still images applied to a series of images. The compression/decompression is symmetric and is done in one-thirtieth of a second. JPEG compression can introduce some artifacts into some images with hard edges. The MPEG (Moving Pictures Expert Group) standards were designed specifically for digital video. MPEG uses the same algorithms as JPEG for intraframe compression of individual frames, called I-frames. MPEG then uses interframe compression to create B (bidirectional) and P (predicted) frames. MPEG allows quality settings to specify the amount of compression to use [24] and can be set individually for each frame type (I, P, B). MPEG-1 is designed for high-quality CD video, MPEG-2 is designed for high-quality DVD video, and MPEG-4 is designed for low-bandwidth Web applications [22]. Cinepak and Sorenson are products initially targeted for the MAC world, although Cinepak is now available for the PC. Cinepak uses block-oriented vector quantization. Sorenson uses YUV compression with 4x4 blocks and employs interframe compression [20]. Indeo 3.2 is an Intel product that also uses block-oriented vector quantization. Cinepak and Indeo are highly asymmetric, requiring on the order of three hundred times longer for compression than for their efficient decompression. Indeo also incorporates color blending and run-length encoding into its scheme.

### Digital Video Formats

The codec products (as opposed to the underlying codec techniques) used for DV have an associated file format. Some formats also include timing information, the ability to animate overlay images (sprites), and the ability to loop over a series of frames. MPEG and MJPEG are both common DV formats. GIF89a-based animation (animated GIF) is basically a number of GIF images stored in one file with interframe timing information but no interframe compression. Compressing continuous-tone images can result in color banding. The compression does well on line drawings but not on complex outdoor scenes. GIF animations can use delta frames, which, for example, overlay images on a previously transmitted background. This saves retransmitting static information for some animations. As used here, *movie format* refers to a format that is codec independent and that can handle audio as well as imagery. Both Quicktime (MOV) and Video for Windows (AVI) [2] are movie formats designed as open codec architectures. Any codec can be used with these standards as long as a compatible plug-in is available. Cinepak has been a standard codec used with Quicktime, but Quicktime can also accommodate other codecs such as Sorenson and JPEG [24]. Quicktime organizes data into tracks and includes timing information. Video for Windows from Microsoft uses Video I and RLE as standard codecs but can also accommodate others. Video for Windows allows interleaving of image and audio information.

### Digital Television Formats

Most of the DTV formats are based on sampling scaled versions of the color difference signals (*B-Y, R-Y*). Luminance and the scaled color difference signals are referred to as YUV. Common formats are D1, D2, D3, D5, D6, Digital Betacam, Ampex DCT, Digital8, DV, DVCam, and DVCPRO. When digitally sampled, the YUV signals are referred to as YCrCb. A typical sampling scheme is 4:2:2, in which the color difference signals (Cr, Cb) are sampled at half the sampling rate of luminance (*Y*) in the horizontal direction. 4:1:1 sampling means that the color difference signals are also sampled at half the rate in the vertical direction, resulting in one-quarter of the luminance sampling.

The D1 standard was developed when the broadcast television industry thought it would make the composite-analog-to-component-digital transition in one fell swoop. But that did not happen because the cost was prohibitive. D1 uses YUV coding, so-called 4:2:2, which means that the *U* and *V* components are horizontally subsampled 2:1. Luminance is sampled at 13.5 MHz, resulting in 720 samples per picture width. There is no compression other than undersampling the chroma information. Aggregate data rate is roughly 27 MB/s (megabytes per second). The D2 standard was developed as a low-cost alternative to D1. D2 is a composite NTSC digital format (i.e., digitized NTSC). The composite signal is sampled at four-times-color-subcarrier, about 14.318 MHz at one byte per sample (aggregate data rate, of course, 14.318 MB/s). It has all the impairment of NTSC but the reliability and performance of digital. It uses the same 3/4" cassette as D1. As with D1, D2 uses no compression other than undersampling the chroma information. Other uncompressed digital formats include D3, D5, and D6. D3 is a composite format that uses 1/2" tape. D5 is a component format that uses 1/2" tape. D6 is a component HDTV format.

Common compressed DTV formats include Digital Betacam, Ampex DCT, and Digital8 [9]. Digital Betacam uses 1/2" tapes similar to the Betacam SP format with 2:1 compression based on DCT. Ampex DCT is a proprietary format; the *DCT* in its name stands for *Digital Component Technology* and not the compression scheme. The trio of DV, DVCam, and DVCPRO are similar formats using DCT compression. Depending on image content, the encoder decides whether to compress two fields separately or as a unit. Digital8 is a consumer-grade version of the DV format but uses cheaper Hi8 tapes. Newer formats include W-VHS, Digital S, Betacam SX, Sony HDD-1000, and D-VHS.

In the United States the HDTV standard is being facilitated by the Grand Alliance, a group of manufacturers and research labs. It is based on the MPEG-2 codec and is to be fully available by 2003 [8].

## B.9 Camera Calibration

For digitally capturing motion from a camera image, one must have a transformation from the image coordinate system to the global coordinate system. This requires knowledge of the camera's intrinsic parameters (focal length and aspect ratio) and extrinsic parameters (position and orientation in space). In the capture of an image, a point in global space is projected onto the camera's local coordinate system and then mapped to pixel coordinates. To establish the camera's parameters, one uses several points whose coordinates are known in the global coordinate system and whose corresponding pixel coordinates are also known. By setting up a system of equations that relates these coordinates through the camera's parameters, one can form a least-squares solution of the parameters [23].

Calibration is performed by imaging a set of points whose global coordinates are known and identifying the image coordinates of the points and the correspondence between the two sets of points. This results in a series of five-tuples, ($x_i$, $y_i$, $z_i$, $c_i$, $r_i$) consisting of the 3D global coordinates and 2D image coordinates for each point. The 2D image coordinates are a mapping of the local 2D image plane of the camera located a distance $f$ in front of the camera (Equation B.149). The image plane is located relative to the 3D local coordinate system ($u$, $v$, $w$) of the camera (Figure B.50). The imaging of a 3D point is approximated using a pinhole

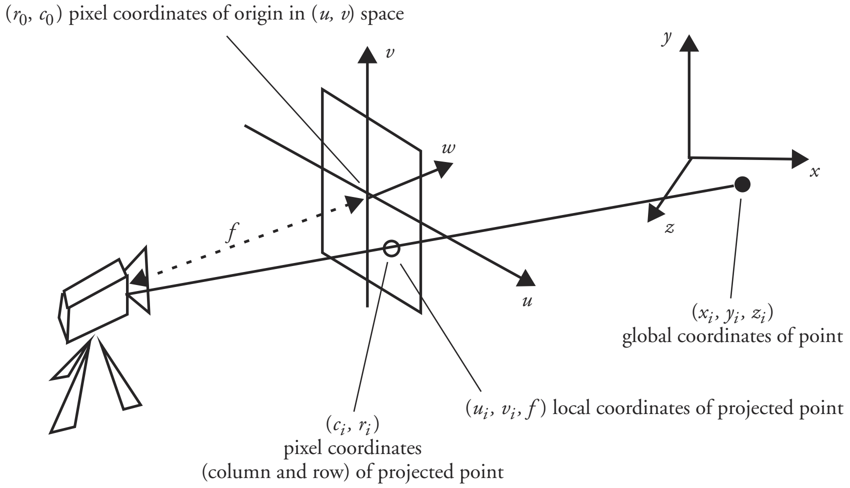

$(r_0, c_0)$ pixel coordinates of origin in $(u, v)$ space
$v$
$y$
$w$
$x$
$f$
$z$
$u$
$(x_i, y_i, z_i)$ global coordinates of point
$(u_i, v_i, f)$ local coordinates of projected point
$(c_i, r_i)$ pixel coordinates (column and row) of projected point

**Figure B.50** Coordinate systems used in the projection of a global point to pixel coordinates

camera model. The 3D local coordinate system of the camera is related to the 3D global coordinate system by a rotation and translation (Equation B.150); the origin of the camera's local coordinate system is assumed to be at the focal point of the camera. The 3D coordinates are related to the 2D coordinates by the transformation to be determined. Equation B.151 expresses the relationship between a pixel's column and row number and the global coordinates of the point. These equations are rearranged and set equal to zero in Equation B.152. They can be put in the form of a system of linear equations (Equation B.153) so that the unknowns are isolated (Equation B.154) by using substitutions common in camera calibration (Equation B.155, Equation B.156). Temporarily dividing through by $t3$ ensures that $t_3 \neq 0.0$ and therefore that the global origin is in front of the camera. This step results in Equation B.157, where $A'$ is the first 11 columns of $A$; $B'$ is the last column of $A$; and $W'$ is the first 11 rows of $W$. Typically, enough points are captured to ensure an overdetermined system. Then a least-squares method, such as Singular Value Decomposition, can be used to find the $W'$ that satisfies Equation B.158. $W'$ is related to $W$ by Equation B.159, and the camera parameters can be recovered by undoing the substitutions made in Equation B.155 by Equation B.160. Because of numerical imprecision, the rotation matrix recovered may not be orthonormal, so it is best to reconstruct the rotation matrix first (Equation B.161), massage it into orthonormality, and then use the new rotation matrix to generate the rest of the parameters (Equation B.162).

$$\begin{aligned} c_i - c_0 &= s_u \cdot u_i \\ r - r_0 &= s_v \cdot v_i \end{aligned} \qquad \text{(Eq. B.149)}$$

$$\begin{bmatrix} u_i \\ v_i \\ f \end{bmatrix} = R \cdot \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} + T$$

$$R = \begin{bmatrix} R_0 \\ R_1 \\ R_2 \end{bmatrix} = \begin{bmatrix} r_{0,0} & r_{0,1} & r_{0,2} \\ r_{1,0} & r_{1,1} & r_{1,2} \\ r_{2,0} & r_{2,1} & r_{2,2} \end{bmatrix}$$

$$T = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \end{bmatrix} \tag{Eq. B.150}$$

$$\frac{u_i}{f} = \frac{c_i - c_0}{s_u \cdot f} = \frac{c_i - c_0}{f_u} = \frac{R_0 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_0}{R_2 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_2}$$

$$\frac{v_i}{f} = \frac{r_i - r_0}{s_v \cdot f} = \frac{r_i - r_0}{f_v} = \frac{R_1 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_1}{R_2 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_2} \tag{Eq. B.151}$$

$$(c_i - c_0) \cdot \left( R_2 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_2 \right) - f_u \cdot \left( R_0 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_0 \right) = 0$$

$$(r_i - r_0) \cdot \left( R_2 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_2 \right) - f_v \cdot \left( R_1 \bullet \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} + t_1 \right) = 0 \tag{Eq. B.152}$$

$$A \cdot W = 0 \tag{Eq. B.153}$$

$$\begin{bmatrix} -x_1 & -y_1 & -z_1 & 0 & 0 & 0 & r_1 \cdot x_1 & r_1 \cdot y_1 & r_1 \cdot z_1 & -1 & 0 & r_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -z_1 & c_1 \cdot x_1 & c_1 \cdot y_1 & c_1 \cdot z_1 & 0 & -1 & c_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -x_n & -y_n & -z_n & 0 & 0 & 0 & r_n \cdot x_n & r_n \cdot y_n & r_n \cdot z_n & -1 & 0 & r_n \\ 0 & 0 & 0 & -x_n & -y_n & -z_n & c_n \cdot x_n & c_n \cdot y_n & c_n \cdot z_n & 0 & -1 & c_n \end{bmatrix} \tag{Eq. B.154}$$

$$\begin{aligned}
W_0 &= f_u \cdot R_0 + c_0 \cdot R_2 = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix}^T \\
W_3 &= f_v \cdot R_1 + r_0 \cdot R_2 = \begin{bmatrix} w_3 & w_4 & w_5 \end{bmatrix}^T \\
W_6 &= R_2 = \begin{bmatrix} w_6 & w_7 & w_8 \end{bmatrix}^T \\
w_9 &= f_u \cdot t_0 + c_0 \cdot t_2 \\
w_{10} &= f_v \cdot t_1 + r_0 \cdot t_2 \\
w_{11} &= t_2
\end{aligned} \tag{Eq. B.155}$$

$$W = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 & w_9 & w_{10} & w_{11} \end{bmatrix}^T \tag{Eq. B.156}$$

$$A' \cdot W' + B' = 0 \tag{Eq. B.157}$$

$$\min_{w} \| A' \cdot W' + B' \| \tag{Eq. B.158}$$

$$W = \begin{bmatrix} W_0 \\ W_3 \\ W_6 \\ w_9 \\ w_{10} \\ w_{11} \end{bmatrix} = \frac{1}{\| W_6' \|} \cdot \begin{bmatrix} W_0' \\ W_3' \\ W_6' \\ w_9' \\ w_{10}' \\ 1 \end{bmatrix} \tag{Eq. B.159}$$

$$
\begin{aligned}
c_0 &= W_0^T \cdot W_6 \\
r_0 &= W_1^T \cdot W_6 \\
f_u &= -\| W_0 - c_0 \cdot W_6 \| \\
f_v &= \| W_3 - r_0 \cdot W_6 \| \\
t_0 &= (w_9 - c_0)/f_u \\
t_1 &= (w_{10} - r_0)/f_v \\
t_2 &= w_{11} \\
R_0 &= (W_0 - c_0 \cdot W_6)/f_u \\
R_1 &= (W_3 - r_0 \cdot W_6)/f_v \\
R_2 &= W_6
\end{aligned}
\tag{Eq. B.160}
$$

$$
\begin{aligned}
R_0 &= (W_0' - c_0 \cdot W_6')/f_u \\
R_1 &= (W_3' - r_0 \cdot W_6')/f_v \\
R_2 &= W_6'
\end{aligned}
\tag{Eq. B.161}
$$

$$
\begin{aligned}
c_0 &= W_0^T \cdot R_2 \\
r_0 &= W_1^T \cdot R_2 \\
f_u &= -\| W_0 - c_0 \cdot R_2 \| \\
f_v &= \| W_3 - r_0 \cdot R_2 \| \\
t_0 &= (w_9 - c_0)/f_u \\
t_1 &= (w_{10} - r_0)/f_v \\
t_2 &= w_{11}
\end{aligned}
\tag{Eq. B.162}
$$

Given an approximation to a rotation matrix, $\tilde{R}$, the objective is to find the closest valid rotation matrix to the given matrix (Equation B.163). This is of the form shown in Equation B.164, where the matrices $C$ and $D$ are notated as shown in Equation B.165. To solve this, define a matrix $B$ as in Equation B.166. If $q = (q_0, q_1, q_2, q_3)^T$ is the eigenvector of $B$ associated with the smallest eigenvalue, $R$ is defined by Equation B.167 [26].

$$min\|\tilde{R} - R\| \tag{Eq. B.163}$$

$$\|R \cdot C - D\| \tag{Eq. B.164}$$

$$\begin{aligned} C &= [C_1, C_2, C_3] \\ D &= [D_1, D_2, D_3] \end{aligned} \tag{Eq. B.165}$$

$$\begin{aligned} B &= \sum_{i=1}^{3} B_i^T \cdot B_i \\ B_i &= \begin{bmatrix} 0 & (C_i - D_i)^T \\ D_i - C_i & [D_i - C_i]_X \end{bmatrix} \\ [(x, y, z)]_X &= \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \end{aligned} \tag{Eq. B.166}$$

$$R = \begin{bmatrix} q_0 - q_1 + q_2 - q_3 & 2 \cdot (q_1 \cdot q_2 - q_0 \cdot q_3) & 2 \cdot (q_1 \cdot q_3 + q_0 \cdot q_2) \\ 2 \cdot (q_1 \cdot q_2 + q_0 \cdot q_3) & q_0 - q_1 + q_2 - q_3 & 2 \cdot (q_3 \cdot q_2 - q_0 \cdot q_1) \\ 2 \cdot (q_1 \cdot q_3 - q_0 \cdot q_2) & 2 \cdot (q_3 \cdot q_1 + q_0 \cdot q_2) & q_0 - q_1 - q_2 + q_3 \end{bmatrix} \tag{Eq. B.167}$$

# References

1. R. Bartels, J. Beatty, and B. Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling,* Morgan-Kaufmann, San Francisco, 1987.
2. D. Dixon, "AVI Formats," http://www.manifest-tech.com/pc_video/avi_formats/avi_formats.htm, January 2001.
3. D. Ebert, ed., *Texturing and Modeling: A Procedural Approach,* 2d ed., AP Professional, Boston, 1998.
4. G. Farin, *Curves and Surfaces for Computer Aided Design,* Academic Press, Orlando, Fla., 1990.
5. S. Frautsch, R. Olenick, T. Apostol, and D. Goodstein, *The Mechanical Universe: Mechanics and Heat, the Advanced Edition,* Cambridge University Press, Cambridge, 1986.
6. S. Gasch, "0.5.12.0 Source Code," http://wannabe.guru.org/alg/node136.html, January 2001.

7. S. Gottschalk, M. Lin, and D. Manocha, “OBB Tree: A Hierarchical Structure for Rapid Interference Detection,” Proceedings of SIGGRAPH 96, Computer Graphics Proceedings, Annual Conference Series, pp. 171–180 (August 1996, New Orleans, La.). Addison-Wesley. Edited by Holly Rushmeier. ISBN 0-201-94800-1.
8. D. Hromas, “Digital Television: The Television System of the Future . . . ,” http://www.cgg.cvut.cz/~xhromas/dtv/, January 2001.
9. M. Iisakkila, “Video Recording Formats,” http://www.hut.fi/u/iisakkil/videoformats.html, January 2001.
10. B. King, “TV Systems: A Comparison,” http://www.ee.surrey.ac.uk/Contrib/WorldTV/compare.html, January 2001.
11. D. Kochanek, “Interpolating Splines with Local Tension, Continuity and Bias Control,” *Computer Graphics* (Proceedings of SIGGRAPH 84), 18 (3), pp. 33–41 (July 1984, Minneapolis, Minn.).
12. K. Kuhn, “HDTV Television—an Introduction: EE 498,” http://www.ee.washington.edu/conselec/CE/kuhn/hdtv/95x5.htm, January 2001.
13. N. Madsen, “Three Dimensional Dynamics of Rigid Bodies,” http://www.eng.auburn.edu/users/gflowers/me232/class_problems/angmom.html, January 2001.
14. M. Mortenson, *Geometric Modeling,* John Wiley & Sons, New York, 1985.
15. S. Norwood, “rec.arts.movies.tech: Frequently Asked Questions (FAQ) Version 2.00,” http://www.redballoon.net/~snorwood/faq2.html, January 2001.
16. W. Press, B. Flannery, S.Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing,* Cambridge University Press, Cambridge, 1986.
17. J. Ritter, “An Efficient Bounding Sphere,” *Graphics Gems* (A. Glassner, ed.), Academic Press, New York, 1990, pp. 301–303.
18. D. Rogers and J. Adams, *Mathematical Elements for Computer Graphics,* McGraw-Hill, New York, 1976.
19. M. Rogge, “More Than One Hundred Years of Film Sizes,” http://www.xs4all.nl/~wichm/filmsize.html, January 2001.
20. Sorenson Media, “Sorenson Video Tutorial: Sorenson Video Compression,” http://www.sorenson.com/Sorenson-Video/tutorial/page03.html, January 2001.
21. L. Speights II, “Video Compression Methods (Codecs),” http://home.earthlink.net/~radse/Page9.html, January 2001.
22. Terrain Interactive, “Index of Codecs for Video, CD, DVD, and Audio,” http://www.terran.com/CodecCentral/Codecs/index.html, January 2001.
23. M. Tuceryan, G. Greer, R. Whitaker, D. Breen, C. Crampton, E. Rose, and K. Ahlers, “Calibration Requirements and Procedures for a Monitor-Based Augmented Reality System,” *IEEE Transactions on Visualization and Computer Graphics,* 1 (3), pp. 255–273 (September 1995).
24. H. Vahlenkamp, “GFDL Visualization Guide: Animation Formats,” http://www.manifest-tech.com/pc_video/avi_formats/avi_formats.htm, January 2001.
25. B. Waggoner, “Technology: A Web Video Guide to Codecs,” http://www.dv.com/webvideo/2000/0900/waggoner0900.html, January 2001.
26. J. Weng, P. Cohen, and M. Herniou, “Camera Calibration with Distortion Models and Accuracy Evaluation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14 (10), pp. 965–980 (October 1992).

# Index

## Numbers

2D grid deformation. *See* grid deformation (2D)
2D morphing. *See* morphing (2D)
3D bucket sort, flocking behavior and, 251
3D shape interpolation. *See* shape interpolation (3D)
4x4 transformation matrix. *See* transformation matrices

## A

AABBs (axis-aligned bounding boxes), 431
*Abyss, The,* 26
acceleration
  average acceleration, 476
  for bodies in free fall, 205, 206–207
  constant (parabolic ease-in/ease-out), 89–92
  equations for, 476–477
  particle animation and, 244
  relative, resting contact and, 230
  sine interpolation and, 87
  sinusoidal pieces for, 87–89
  *See also* ease-in/ease-out
acceleration-time curves, distance-time functions and, 92–94
ACM (Association for Computing Machinery), 1
Action Units (AUs), 347–348
active optical markers for motion capture, 372
actor-based animation languages, 123–124
Adams, J., 458
adaptive approach to arc length computation, 76–77
adaptive Gaussian integration of arc length, 79–83
advanced algorithms
  automatic camera control, 174–175
  controlling groups of objects, 241–261
  enforcing soft and hard constraints, 231–241
  hierarchical kinematic modeling, 175–203
  implicit surfaces, 261–267
  rigid body simulation, 203–231
*Adventures of Andre and Wally B., The,* 25
AFFD objects, 138
affine transformations
  defined, 40
  matrix representation for, 42
  object deformation and, 124–125
  overview, 40–42
  *See also specific transformations*
afterimages, 2
Airy model for waves, 289
alpha channel for compositing, 390, 391–395
Ampex DCT DTV standard, 501, 502
analog standards for moving pictures, 493, 494
analytic approach to arc length computation, 72–73
anatomical terms, 318
angle of view, 34, 35
angular mass. *See* inertia tensors
angular momentum
  of bodies in free fall, 212, 213–214
  overview, 486–487
angular velocities
  of bodies in free fall, 209–211
  circular motion equations, 477–478
  defined, 209, 477

- angular velocities (*continued*)
  - impulse forces and, 226
  - induced by joint axis rotation, 196–197
  - Jacobian and, 195–197
  - proximity of axis of rotation and, 209–210
- ANIMA II, 23, 122
- animated FFDs, 137–143
  - AFFD objects, 138
  - animating the FFD, 140–143
  - deformation tools, 138–140
  - moving the object, 140, 141
  - moving the tool, 138–139
  - overview, 137
- animatic, 13
- animation, defined, 4
- animation languages, 120–124
  - actor-based, 123–124
  - advantages of, 120–121
  - articulation variables in, 122
  - artist-oriented, 121–122
  - general-purpose languages vs., 120–121
  - graphical, 123
  - MEL, 121
  - support for existing languages in, 122
  - typical features, 120
- animation production, 12–21
  - computer animation tasks, 15–17
  - computer vs. conventional animation, 13–14
  - digital editing, 17–20
  - digital video, 20–21
  - hierarchy, 12
  - rendering for previews, 14–15
  - steps, 12–13
- ankle-toe joints, 332, 335
  - optical markers for, 375, 376
- anti-aliasing
  - compositing and, 390–392
  - temporal, 400
- anticipation, 11
- *Ants,* 28
- appeal, 11
- approximation
  - defined, 454
  - interpolation vs., 64–65, 454
  - *See also* curves, interpolating and approximating
- arbitration in autonomous behavior, 260
- arc length
  - defined, 71, 456
  - normalized arc length parameter, 84
  - notation for, 71
  - parameter, arc length pairs (table), 74
  - parameterization by, 456
  - parametric value and, 72
  - as speed control function's output, 84
- arc length computation, 69–84
  - adaptive approach to, 76–77
  - adaptive Gaussian integration for, 79–83
  - analytic approach to, 72–73
  - distance-time function, 70–71
  - finding *u* given *s,* 83–84
  - forward differencing approach, 73–76
  - *LENGTH* function, 71, 72
  - numeric computation, 77–79
  - parametric value and arc length, 72
  - sampling parametric values for, 73–76
  - space curve, 70–71
  - uses for, 69–70
- arc length integral, evaluating, 72–73
- arc length parameterization
  - approaches to, 69
  - defined, 71
  - need for, 68–69
- arcs
  - defined, 11
  - in hierarchical linkages, 177, 178, 179–181
  - *See also* joints
- area calculations
  - polygon, 429, 430
  - triangle, 428
- arm modeling, 318–321
  - basic model, 319
  - coordinated movement and, 324–325
  - degrees of freedom, 318, 319, 320, 321
  - forearm rotation, 318–319
  - inverse kinematics for, 319–320
  - joint limits, 319
  - wrist orientation and, 320
- articulated figures, 317–384
  - anatomical terms, 318
  - arm modeling, 318–321
  - challenges for computer animation, 317
  - cloth and clothing, 361–369
  - comfort metrics, 326–327
  - coordinated movement, 324–325
  - defined, 14, 176
  - facial animation, 339–353
  - hand modeling, 321–324
  - hierarchical modeling of, 175–183
  - layered approach to human figure modeling, 358–361
  - motion capture, 369–378
  - obstacles, reaching around, 325–326
  - reaching and grasping, 318–327
  - shoulder joint modeling, 321, 322
  - strength criteria, 326–327
  - summary, 379
  - transformations for arcs leading to nodes, 179
  - virtual human representation overview, 353–358
  - walking, 327–339
  - *See also* hierarchical modeling
- articulation, 176
- articulation variables (avars), 116, 122
- artist-oriented animation languages, 121–122
- ASAS system, 124
- assemble editing, 19
- Association for Computing Machinery (ACM), 1
- asymmetric vs. symmetric compression/decompression, 499
- attachment point for muscles, 350
- attenuation functions for warping objects, 125, 126
- attributes of particles, 244
- AUs (Action Units), 347–348
- automatic camera control, 174–175
- autonomous behavior, 257–260
  - arbitration between competing intentions, 260
  - characteristics of, 242
  - defined, 257
  - degrees of freedom, 259, 260
  - emergent behavior and, 257–258
  - internal state, 259
  - keeping behavior under control, 259–260
  - knowing the environment, 258–259
  - levels of behavior, 259, 260
  - motor activities, 259, 260
  - movement coordinator, 259, 260
  - overview, 242, 257–258
  - planner, 259, 260
  - reasoning unit, 259, 260
  - strategies, 259, 260
  - uses for, 258
- avars (articulation variables), 116, 122
- average acceleration, 476
- average velocity, 476
- Avery, Tex, 11
- AVI (Video for Windows) standard, 501
- axial slices shape interpolation, 159–160
  - coordinate system, 159
  - interpolating slices, 160, 161
  - multiple 2D slices example, 162
  - parameterization for, 159–160
  - projection lines, 160
- axis-aligned bounding boxes (AABBs), 431

axis-angle representation, 57–58
axis-angle rotations, 449

## B

B-splines
arc length parameterization and, 72
facial models, 342
fitting to position-time pairs, 95–97
NURBS (Nonuniform rational B-splines), 354, 355, 469
overview, 467–469
second-order continuity in, 66
smoothing paths by B-spline approximation, 114
*Babe,* 26
background image, 389
Badler, N., 22, 259, 400
ball-and-socket joints
as complex joints, 176
degrees of freedom, 177, 190, 191
DH notation for, 190–191
shoulder joint modeling, 321, 322
Baraff, D., 228, 231
Barr, A., 130, 236, 237, 238
Bartels, R., 342
barycentric coordinates, 430–431
basic techniques
3D shape interpolation, 153–169
animation languages, 120–124
controlling motion along a curve, 68–97
deforming objects, 124–143
interpolation, 63–68
key-frame systems, 116–120
morphing (2D), 143–153
path following, 102–116
summary, 169–170
*Batman Returns,* 27
BBOP, 23
*Beauty and the Beast,* 28
bends, global deformation for, 132
Betacam SX DTV standard, 502
Betamax videotape format, 479, 496
Bezier curves
composite cubic, 464
constrained, 68
constructing in four-dimensional spherical space, 101–102
de Casteljau construction of, 100–101, 464–465
first-order continuity and, 66
Hermite form vs., 463
between two interpolated points, 100
Bezier interpolation/approximation, 463–464
bias controls for curves, 466–467
bilinear interpolation, 2D grid deformation and, 127, 128
binary subdivision for finding *u* given *s,* 83
*Black and White,* 29
black-and-white video standard, 495
Blackton, J. Stuart, 6
blended parabolas. *See* parabolic blending
blending functions, 455
Bloomenthal, J., 263
Blumberg, J., 259
bodies in contact, 216–231
colliding bodies, 216–227
contact forces, 481–483
friction, 227–228
resting contact, 228–231
types of contact, 216
*See also* colliding bodies
bodies in free fall, 204–216
acceleration function, 204, 205
angular velocity, 209–211
center of mass, 211–212
equations of motion for, 208–216
forces acting on, 205, 208, 212
improving simulation accuracy, 206–208
inertia tensor, 209, 214
momentum, 208, 212–214
numeric approximation and, 206–208
orientation and rotational movement, 209–211
Runge-Kutta integration for, 207–208
simple example, 205–206
standard physics vs. computer animation and, 204
time function, 204, 205
torque, 208, 212
updating orientation for, 215–216
velocity function, 204, 205
boids, 246
bounding boxes, 431–432
bounding shape computations, 431–447
bounding boxes, 431–432
bounding slabs, 432–433
bounding sphere, 433–436
convex hull, 437–447
bounding slabs, 432–433
bounding sphere
for collision avoidance, 252–254
computing, 433–436
bracketed L-systems, 277–278, 279
Bray, John, 6–7
bucket sort, flocking behavior and, 251
*Bud Bowl* commercials, 27
buds, plant, 274
*Bug's Life, A,* 28
*Bunny,* 28
Burtnyk, Nestor, 23, 127, 132
Burton, Tim, 10

## C

CAD system, 343
camera calibration for motion capture, 372–373, 502–507
camera position
3D position reconstruction, 373–376
automatic camera control, 174–175
camera calibration for motion capture, 372–373, 502–507
to check facial expressions and eye movements, 175
defined, 33
for inspecting motion of other objects, 175
orientation along a path, 102–108
*See also* eye position (EYE)
Carlson, W., 156
*Caspar,* 26, 28
catenary curves for cloth draping, 362–364
Catmull, Ed, 22, 23
Catmull-Rom splines
for coordinate grid morphing, 144, 146
first-order continuity and, 66
internal tangent vectors and, 459–461
as interpolating splines, 68
overview, 458–461
cell growth in plants, 274
cellular descent in plants, 274
cels, 7
Center for Human Modeling and Simulation, 22
center of interest (COI)
for automatic camera control, 174
for camera path following, 106–108
constant global offset vector for, 175
for groups of objects, 174, 175
look-to point, 33
separate path for, 107–108
up vector and, 33, 34, 106–107
view direction and, 33
center of mass
for bodies in free fall, 211–212
defined, 211, 479
polyhedra collision and, 220
centrifugal force, 483–484
centripetal force, 481
CFD (computational fluid dynamics), 296, 298
CGRG (Computer Graphics Research Group), 22–23

Chadwick, J., 140, 143, 358
chained objects, hierarchical modeling of, 176
channels. *See* articulation variables (avars)
Chen, D., 355
Chen, E., 159, 160
child node, 178
chorus line effect in flocking behavior, 248
Cinepak codec, 500
circular motion, 477–478
cirrus clouds, 299, 306–308
clavicle joint, 322
clipping, 47–48
clipping distance, 34, 35
closest point between two lines in three-space, 427–428
cloth and clothing, 361–369
  catenary curves for, 362–364
  collision detection and response for, 368–369
  modeling dynamics for, 365–368
  relaxation process for, 364–365
  simple draping, 362–365
  virtual human representation, 357
clouds, 299–310
  animating volumetric procedural clouds, 308–310
  basic types, 299
  cirrus, 299, 306–308
  convective, 299, 301
  cumulus, 299, 302, 304–306
  dynamics, 308
  early approaches to modeling, 300–301
  implicit functions for modeling, 303–304
  physics, 299
  rendering issues for, 299–300
  stratus, 299, 308
  visual characteristics of, 299–300
  volumetric modeling, 301–308
codecs, 500–501
Cohl, Emile, 6
COI. *See* center of interest (COI)
colliding bodies, 216–227
  collision detection, 216–217, 220–222
  collision response, 216, 217–227
  impulse force of collision, 222–227
  issues involved, 216
  kinematic response, 217–218
  particle-plane collision, 217–218, 222–224
  penalty method, 217, 218–220
  testing planar polyhedra, 220–222
  *See also* bodies in contact
collision detection
  for cloth and clothing, 368–369
  for colliding bodies, 216–217, 220–222
  flocking behavior collision avoidance, 247, 248–249, 251–255
  for implicit surfaces, 264–265
  by plants, 274
  testing planar polyhedra, 220–222
collision response
  articulated figures and, 325
  for cloth and clothing, 368–369
  implicit surface deforming, 265–267
  impulse force of collision, 222–227
  issues involved, 216
  kinematic, 217–218
  particle-plane collision, 217–218, 222–224
  penalty method, 217, 218–220
color burst, 485
color incorporation into black-and-white signal, 495–496
comfort metrics for articulated figures, 326–327
**comp** operator for compositing, 396–398
competing intentions in autonomous behavior, 260
complex joints, 176–177
complexity
  in arm modeling, 325
  in autonomous behavior modeling, 258
  in cloud modeling, 299
  in facial animation, 339
  in fire modeling, 310
  in hair modeling, 357–358
  in hand modeling, 322
  interpolation issues, 65
  *n*-squared, in flocking behavior, 251
  in natural phenomena modeling, 271–272
  of plant modeling, 272
  of walking, 325, 327
component signal, 495
components of a vector, 420
composite cubic Bezier curves, 464
composite Hermite curves, 457, 458
*composite* operation, 389
composite signal, 495–496
compositing
  alpha channel for, 390, 391–395
  anti-aliasing and, 390–392
  **comp** operator for, 396–398
  defined, 386
  digital, 20–21, 386, 388–398
  FFDs, sequential vs. hierarchical, 136–137, 138
  for foreground animation with static background, 388
  **over** operator for, 389, 391–394, 395–396, 398
  with pixel depth information, 388, 395–398
  without pixel depth information, 389–395
  **zmin** operator for, 396
compound global deformations, 133
compound Hermite curves, 66
compound implicit surfaces, 262, 263
compound transformations
  extracting transformations from a matrix, 45–46
  overview, 39
compressibility of gaseous phenomena, 296
compression. *See* image compression
computational fluid dynamics (CFD), 296, 298
computer animation
  heritage of, 4–12
  history of, 21–29
  principles, 10–12
  production tasks, 15–17
  uses for, 1
  values as candidates for, 1–2
Computer Graphics Research Group (CGRG), 22–23
computer monitors, double buffering for, 385–386, 387
conformational parameters for facial modeling, 349
conforming, 18
connectivity constraints, hierarchical modeling and, 175–176
conservation of energy, 484
conservation of momentum, 484–485
constant acceleration (parabolic ease-in/ease-out), 89–92
constant global offset vector for COI, 175
constant-speed rotation, 98
constraints
  for automatic camera control, 175
  for energy minimization, 236–239
  enforcing soft and hard constraints, 231–241
  floating attachment, 238
  for hierarchical modeling, 175
  locally abutting, 238
  motion, 94, 96
  moving point, 119
  overconstrained systems, 192
  pelvis and feet, 336
  point-to-fixed-point, 238
  point-to-point, 238
  position-time, 96
  on resting contact forces, 228, 229–230
  space-time, 239–241

  for speed control, 85
  underconstrained systems, 192
  *See also* enforcing soft and hard constraints
contact, bodies in. *See* bodies in contact
contact forces, 481–483
  friction, 227–228, 482
  Hooke's law, 481–482
  impulse force of collision, 222–227, 482
  normal, 482
  viscosity, 483
contact shadows, 401
context-free L-systems, 275, 277
context-sensitive L-systems, 280
continuity
  controls for curves, 465–466, 467
  defined, 453
  first-order or tangential, 66, 453–454
  free-form deformation (FFD) and, 135
  geometric, 454
  interpolation and, 65–67
  parametric, 454
  second-order or curvature, 66, 454
  between segments, 66
  zeroth-order or positional, 65–66, 453, 454
continuous approach to motion blur calculation, 399
control points for slerping, 99–100
controlling groups of objects. *See* groups of objects
controlling motion along a curve. *See* motion along a curve
convective clouds, 299, 301
conventional animation
  computer animation vs., 13–14
  defined, 6
  Disney innovations, 7–9
  early days of, 6–7
  fast motion representation in, 399
  studio contributions, 9
convex hull computation, 437–447
convex hull property, 454
convolution kernels for path smoothing, 111–114
  integral calculation, 113
  sample kernels, 113
  step function for, 111–112
  tent kernel, 113–114
coordinate grid deformation
  2D grid deformation, 127–129
  free-form deformation (FFD), 132–143
  global deformations, 130–132
  initial configuration, 125–126
  local coordinate system for, 125–127
  polyline deformation, 129–130
  *See also* free-form deformation (FFD)
coordinate grid morphing, 143–148
  for animated images, 148, 150
  auxilliary grid formation, 144, 146
  column-by-column process (second pass), 148, 149
  cross dissolve for final image generation, 143, 145, 148
  distorting pixels in $x$-direction (first pass), 144, 146–148
  grid definitions, 143, 144
  intermediate image generation, 143–148
  two-pass procedure, 143–148
coordinate space
  camera calibration for motion capture, 372–373, 502–507
  homogenous representations of points, 36–39
  left- vs. right-handed, 32
  orientation along a path and, 102–103
  $y$- vs. $z$-axis as "up," 32
coordinated movement
  in articulated figures, 324–325
  in hand modeling, 322, 324
Coquillart, S., 138
coronal plane, 318
correspondence problem for shape interpolation, 156
cosine rule, 429–430
cross product of two vectors, 422–424
Csuri, Chuck, 22, 23
cubic B-spline curves, second-order continuity in, 66
cubic Bezier curves
  composite, 464
  de Casteljau algorithm for interpolation, 100–101, 464–465
  first-order continuity and, 66
  between two interpolated points, 100
cubic curves
  arc length computation using Gaussian quadrature, 79
  arc length integral and, 73
  equation for, 109
  for path smoothing, 109–111
cubic equations, 453
cubic interpolation for smoothing paths, 109–111
  end point estimation, 110–111
  equation for cubic curves, 109
  geometric construction of a cubic estimate, 109–110
  geometric construction of a parabolic estimate, 111
  parabolic interpolation, 110–111
cumulus clouds, 299, 302, 304–306
  algorithm for, 304–305
curvature (second-order) continuity, 66, 454
curves, interpolating and approximating, 453–470
  B-splines, 467–469
  Bezier interpolation/approximation, 463–464
  bias controls, 466–467
  blended parabolas, 462–463
  Catmull-Rom spline, 458–461
  continuity controls, 465–466, 467
  de Casteljau construction of Bezier curves, 100–101, 464–465
  derivatives, computing, 456
  fitting curves to a given set of points, 469–470
  four-point form, 461–462
  Hermite interpolation, 457–458
  parameterization by arc length, 456
  simple linear interpolation, 454–456
  tension controls, 465, 466, 467
  terminology, 453–454
curves, motion along. *See* motion along a curve

**D**

D-VHS DTV standard, 502
D0L-systems
  described, 275
  geometric interpretation of, 275–277
  predecessor, 275
  production rules, 275
  successor, 275
D1 DTV standard, 501
D2 DTV standard, 501–502
D3 DTV standard, 501, 502
D5 DTV standard, 501, 502
D6 DTV standard, 501, 502
dampers
  PDCs with, 235–236
  spring-mass-damper modeling of flexible objects, 234–235
damping coefficient, 486
damping force, 486
database amplification, bracketed L-systems and, 278
DCT (Digital Component Technology), 502
DCT (discrete cosine transform), 500
de Casteljau construction of Bezier curves, 100–101, 464–465
*Debut of Thomas Cat, The,* 7
deceleration
  constant (parabolic ease-in/ease-out), 89–92

deceleration (*continued*)
- sine interpolation and, 87
- sinusoidal pieces for, 87–89
- *See also* ease-in/ease-out

DeFanti, Tom, 24
default up vector, 34
deformation tools, 138–140
- defined, 138
- moving the object, 140, 141
- moving the tool, 138–139

deforming objects, 124–143
- 2D grid deformation, 127–129
- coordinate grid deformation, 125–143
- free-form deformation (FFD), 125–127, 132–143, 358–361
- geometric deformation for virtual human figures, 356–357
- global deformations, 130–132
- implicit surfaces, as collision response, 265–267
- initial configuration, 125–126
- local coordinate system for, 125–127
- overview, 124–125
- polyline deformation, 129–130
- warping, 125, 126
- *See also* free-form deformation (FFD)

degrees of freedom (DOFs)
- in arm modeling, 318, 319, 320, 321
- in autonomous behavior, 259, 260
- in ball-and-socket joints, 177, 190, 191
- defined, 54
- fixed angle representation and, 53–54
- in hand modeling, 321, 323
- in joints, 176–177
- motion capture and, 377–378
- in shoulder joint modeling, 321
- walking dynamics and, 337

Denavit-Hartenberg (DH) notation
- for ball-and-socket joint, 190–191
- joint angle parameter, 186, 187, 188, 190, 191
- link length parameter, 186, 187, 188, 190, 191
- link offset parameter, 187, 188
- link twist parameter, 187, 188, 190, 191
- overview, 186–190
- screw transformations, 188
- three-joint manipulator example, 188–190

derivatives, computing, 456
DeRose, T., 368
detailed story, 13
determinant of a 3x3 matrix, 410–411
Deussen, O., 283
dextrous workspace, 192
DH notation. *See* Denavit-Hartenberg (DH) notation
Digital Betacam DTV standard, 501, 502
Digital Component Technology (DCT), 502
digital compositing, 20–21
digital editing, 17–20
- assemble, 19
- EDL (edit decision list), 18
- electronic, 17–18
- insert, 19
- nonlinear, 17
- off-line, 18
- on-line, 18–19
- on-line nonlinear, 19–20

Digital S DTV standard, 502
digital television (DTV), 498, 501–502
digital video
- compression/decompression, 498–500
- formats, 21, 501
- overview, 20–21, 497–501

Digital8 DTV standard, 501, 502
discrete approach to motion blur calculation, 399–401
discrete cosine transform (DCT), 500
Disney studio innovations, 7–9
Disney, Walt, 7–9
display pipeline, 32–36
- defined, 32
- eye space, 34–35
- eye space to image space transformation, 35–36, 37, 47
- field of view, 34
- image space, 35–36
- image space to screen space transformation, 36, 37, 47–48
- object space, 33
- object space to world space transformation, 33, 35, 37, 43, 46, 48
- ray casting and, 36, 38
- round-off error considerations, 48–51
- transformations in, 37, 46–48
- up vector, 33–34
- user-supplied up direction, 33
- world space, 33–34
- world space to eye space transformation, 34, 37, 46–47

distal, 318
distance-based implicit surfaces, 262–263
distance function
- for resting contact analysis, 228–229
- for warping objects, 125

distance-time curve. *See* distance-time functions
distance-time functions
- defined, 70–71
- general functions, 92–94
- normalized, 84, 85–86
- for parabolic ease-in/ease-out, 92
- samples, 95
- for speed control, 84–86

Dobashik, Y., 301
DOFs. *See* degrees of freedom (DOFs)
*Don't Touch Me,* 27
dot product of two vectors, 420–422, 424
double buffering, 385–386, 387
- defined, 386
- pseudocode for, 387

downhill motion of water, 292–295
- first-order implicit numerical integration solution, 294–295
- Navier-Stokes equations for, 292–293
- three-dimensional case, 295
- two-dimensional case, 292–295
- volume preservation and, 295

downhill vector for path along a surface, 115–116
drag in flight modeling, 256
*Dragonheart,* 27
draping of clothing, 362–365
*Dream Flight,* 23
drop shadows, 401–406
- approximating, 405–406
- flattening object copy for creating, 403–404
- perspective projection for creating, 401, 403
- placing over ground plane, 405
- usefulness of, 401, 402, 403

DTV (digital television), 498, 501–502
DV. *See* digital video
DV DTV standard, 501, 502
DVCam DTV standard, 501, 502
DVCPRO DTV standard, 501, 502
dynamic control, 174
dynamics
- in cloth and clothing, 365–368
- forward dynamic control, 338–339
- particle system, 308
- qualitative, 308
- for realistic walking motion, 336–338

## E

ease-in/ease-out
- defined, 11
- for motion along a curve, 86–92
- parabolic, 89–92
- sine interpolation for, 86–87

  - sinusoidal, 87–89
  - for speed control, 84
- ease of use, control as trade-off with, 175
- Ebert, D., 301, 302
- Edison, Thomas, 6
- EDL (edit decision list), 18
- Ekman, P., 347, 348
- elbow joint, 322
- electromagnetic sensors for motion capture, 369
- emergent behavior, 241, 246
- Emshwiller, Ed, 23
- end effector
  - arm modeling and, 319–320
  - defined, 176
  - dextrous workspace for, 192
  - for Jacobian applied to linked appendage, 196
  - joint angle and rotational change in, 196
  - reachable workspace for, 192
- energy
  - conservation of, 484
  - kinetic, 484
  - potential, 484
- energy minimization constraints, 236–239
  - floating attachment, 238
  - gradient determination, 237
  - implicit function, 237
  - locally abutting, 238
  - non-negative smooth functions for, 236
  - parametric position function, 237
  - point-to-fixed-point, 238
  - point-to-point, 238
  - soft constraint drawbacks, 238–239
  - surface normal function, 237
- enforcing soft and hard constraints, 231–241
  - difficulties posed by hard constraints, 231
  - for energy minimization, 236–239
  - for flexible objects, 232–235
  - space-time constraints, 239–241
  - for virtual springs, 235–236
- environment
  - autonomous behavior and, 258–259
  - collision avoidance using feelers, 255
  - flocking behavior interactions, 248–249, 255
  - forces in particle systems, 246
  - plant interaction with, 282–283
- equilibrium, 480
- Euler angle representation, 56–57
- Euler method for solving ODEs, 490–491
- Eulerian formulations for gaseous phenomena, 296–297
- Evans, David, 22
- exaggeration, 11
- explicit equations, 453
- exposure sheet, 13
- expression parameters for facial modeling, 349–350
- extension, 318
- exterior vertices of cloth grid, 362
- extracting transformations from matrices, 45–46
- extremes, 13
- eye position (EYE)
  - defined, 33
  - look-from point, 33
  - up vector and, 33
  - view direction and, 33
- eye space
  - defined, 34–35
  - overview, 34–35
  - in this text, 35
  - transformation to image space, 35–36, 37, 47
  - truncated pyramid in, 35
  - world space transformation to, 34, 36–37

## F

- Facial Action Coding System (FACS), 347–348
- facial animation, 339–353
  - creating the model, 342–345
  - difficulties of, 339
  - Facial Action Coding System (FACS), 347–348
  - muscle models, 350–353
  - parameterized models, 348–350
  - textures, 345–347
  - types of facial models, 341–342
  - uses for, 339
- Facial Animation Parameters (FAPs) in MPEG-4, 345
- Facial Definition Parameters (FDPs) in MPEG-4, 345
- facial expressions, camera position for checking, 175
- FACS (Facial Action Coding System), 347–348
- *Fantasia,* 9
- FAPs (Facial Animation Parameters) in MPEG-4, 345
- far (yon) clipping distance, 34, 35
- FDPs (Facial Definition Parameters) in MPEG-4, 345
- feature-based morphing, 148–153
  - code for, 155
  - crossed feature lines and, 153
  - intermediate feature lines, 148, 150, 151, 152
  - intermediate frames, 153
  - mapping feature lines, 148, 150
  - user-supplied parameters for, 153
  - using multiple line pairs, 151–153
  - using single feature line, 150–151, 152
- FFD. *See* free-form deformation (FFD)
- field of view, 34
- film sizes, 493, 494
- fire, 310–311
- first-order (tangential) continuity, 66, 453–454
- fitting curves to a given set of points, 469–470
- Fiume, E., 301, 311
- fixed angle representation
  - degrees of freedom and, 54–55
  - gimbal lock effect, 55
  - interpolation issues, 55–56
  - overview, 42–43, 53–54
- Flanders, M., 320
- Fleischer, Max, 7, 9
- Fleisher, K., 236, 237, 238
- flexible objects, 232–235
  - dampers, 234–235
  - simple spring-mass model, 233–234
  - spring-mass-damper modeling of, 232–235
- flexion
  - defined, 318
  - finger, 324
  - knee, 332
- flicker, 2, 3
- flicker rate, 2
- flight modeling, 255–257
- flipbooks, 4
- floating attachment constraints, 238
- flock centering, 247
- flocking behavior, 246–257
  - boids, 246
  - characteristics of, 242, 246
  - chorus line effect, 248
  - collision avoidance, 247, 248–249, 251–255
  - described, 241–242
  - emergent behavior, 241, 246
  - flight modeling, 255–257
  - flight modules, 250
  - flock centering, 247
  - flock leader, changing, 249
  - global control, 249
  - interacting with the environment, 248–249
  - interacting with other members, 248
  - local control, 247

flocking behavior (*continued*)
  *n*-squared complexity, 251
  navigation module, 249, 250
  negotiating the motion, 249–250
  perception and, 247, 248
  pilot module, 250
  reasoning and reaction, 247
  splitting and rejoining, 255
  uses for, 246
*Flubber,* 27
fluid dynamics
  computational (CFD), 296, 298
  overview, 296
  *See also* gaseous phenomena
Foldes, Peter, 23, 127
Foley, J., 34
follow through/overlapping action, 11
foot-sliding, 376
force field collision avoidance, 251–252, 253
forces
  on bodies in free fall, 205, 208, 212
  centrifugal force, 483–484
  centripetal force, 481
  collision response, 216, 217–227
  contact, 228–231, 481–483
  dampers, 234–235
  damping, 486
  in flight modeling, 256
  friction, 227–228, 482
  gravity, 256, 257, 282, 292–295, 480–481
  Hooke's law, 481–482
  impulse force, 222–227, 482
  linear, 212
  normal force, 482
  particle system, 246
  resting contact, 228–231, 481–483
  soft constraints as, 231–232
  spring-mass-damper modeling of flexible objects, 232–235
  torque, 208, 212, 220, 230, 235, 326–327, 480
  viscosity, 483
  *See also* colliding bodies; *specific forces*
forearm rotation, 318–319
foreground image, 389
*Forrest Gump,* 26
Forsey, D., 342
forward differencing, arc length computation using, 73–76
forward kinematics, 184–185
Foster, N., 301
four-point form, 461–462
Fourier synthesis for cloud modeling, 300–301
fractal compression, 500
fractal synthesis for cloud modeling, 300
frames
  context and meaning of, 103
  defined, 12
  key, 13, 116
  key-frame systems, 116–120
  local coordinate, 185–192
  in robotics, 176
  speed/quality trade-offs, 14
  in storyboard, 12
free fall. *See* bodies in free fall
free-form deformation (FFD), 132–143
  AFFD objects, 138
  animated FFDs, 137–143
  for Bezier solids, 135
  compositing FFDs, 136–137, 138
  continuity and, 135
  cylindrical grid for, 135–136
  deformation by passing through FFD space, 140, 141
  deformation tools, 138–140
  deforming a surface around an articulated joint, 140–143
  grid of control points for, 134
  hierarchical organization, 136–137, 138
  in human figure modeling, 358–359, 360
  initial configuration, 125–126
  interpolation functions for, 133, 135
  local coordinate system for, 125–127, 133–134
  moving control points, 134–135
  reconstructing a point's position, 134
  sequential organization, 136, 137
Frenet frames, 103–106
  calculating, 103–104
  defined, 103
  interpolating for undefined segments, 104–105
  segments without curvature and, 104–105
  unnaturalness of motions using, 105–106
friction
  kinetic, 228, 482
  overview, 482
  static, 227–228, 482
Friesen, W., 347, 348
function integration, 488–489
*Future World,* 24

**G**

gains, pseudo inverse Jacobian and, 202
Galyean, T., 259
Gardner, G., 300
gaseous phenomena, 296–311
  clouds, 299–310
  compressibility issues, 296
  computational fluid dynamics, 298
  fire, 310–311
  general approaches to modeling, 296–298
  grid-based method, 296–297
  hybrid method, 298
  particle-based method, 297–298
  steady state flow, 296
  vortices, 296
  *See also* clouds
Gauss-Legendre integration, 488, 489, 490
Gaussian integration, adaptive, for arc length computation, 79–83
Gaussian quadrature for arc length computation, 77–79
generally perpendicular, 115
generating particles, 243
genesis effect, 310
genetic algorithms for path planning, 325
geometric computations, 420–447
  area calculations, 428–429
  barycentric coordinates, 430–431
  bounding shapes, 431–447
  closest point between two lines in three-space, 427–428
  components of a vector, 420
  cosine rule, 429–430
  cross product of two vectors, 422–424
  dot product of two vectors, 420–422, 424
  length of a vector, 420, 425
  matrix routines, 425–427
  vector routines, 424–425
geometric construction
  of cubic estimate, 109–110
  of parabolic estimate, 111
geometric continuity, 454
geometric flight, 256
geometric information, shape interpolation and, 157
geometric interpretation of L-systems, 275–277
geometric replacement for L-systems, 275
geotropism, 274
*Geri's Game,* 25
*Gertie the Dinosaur,* 6
*Getting into Art,* 340
GIF standard, 500, 501
GIF89a standard, 501
gimbal lock effect, 55
Girard, M., 322, 336
global control
  defined, 67
  of flocking behavior, 249
  interpolation and, 67

global deformations, 130– 132
- bend operation, 132
- compound operations, 133
- taper operation, 131
- transformation matrix for, 130
- twist about an axis, 131

global offset vector, constant, for center of interest, 175
gradient for energy minimization constraints, 237
Grand Alliance, 502
graphical animation languages, 123
Graphical Symbiosis System (GRASS), 24
grasping. *See* reaching and grasping
GRASS (Graphical Symbiosis System), 24
gravity
- downhill motion of water, 292–295
- in flight modeling, 256, 257
- Newton's law of universal gravitation, 480– 481
- overview, 480– 481
- in plant modeling, 282

Greenberg, Jim, 23
grid-based method for gaseous phenomena, 296–297
grid deformation (2D), 127–129
- bilinear interpolation, 127, 128
- initial configuration, 127
- local coordinate system, 127
- moving grid vertices, 127–129

ground plane, drop shadow placement over, 405
groups of objects, 241–261
- autonomous behavior, 257–260
- center of interest for, 174, 175
- characteristics of group types, 242
- emergent behavior, 241
- flocking behavior, 246–257
- particle systems, 242–246
- *See also* autonomous behavior; flocking behavior; particle systems

## H

hair, 357–358
hand modeling, 321–324
- coordinating movement for, 322, 324
- degrees of freedom, 321, 323
- grasping and, 322, 324
- opposable thumb and, 322

handedness of coordinate spaces, 32
hard constraints, 231. *See also* enforcing soft and hard constraints
Harryhausen, Ray, 10
Haumann, D., 140, 358
HCI (human-computer interaction), 339
HDTV (high-definition television), 496
herbaceous plants, 273
heritage of animation, 4–12
- Disney innovations, 7–9
- early "conventional" animation, 6–7
- early devices, 4–6
- principles of computer animation, 10– 12
- stop-motion techniques, 9–10
- studio contributions, 9
- video art, 10

Hermite curves, 66
Hermite interpolation, 457– 458
hierarchical FFD organization, 136–137, 138
hierarchical kinematic modeling, 175–203
- forward kinematics, 184–185
- hierarchical modeling overview, 175–177
- inverse kinematics, 185, 192–203
- local coordinate frames, 185–192
- representing hierarchical models, 177–183
- summary, 203
- *See also* inverse kinematics; local coordinate system

hierarchical modeling, 175–183
- defined, 175
- degrees of freedom, 176–177
- joints, 176–177
- simple example (two-dimensional, three-link), 179–183
- transformations, 179–185
- tree structure for, 177–179, 181
- types of, 175–176
- world space and, 180– 181

hierarchy of animation levels, 12
high-definition television (HDTV), 496
Hill's muscle model, 353
hips. *See* pelvis
history of computer animation, 21–29
- coming of age, 26–29
- early activity, 21–24
- middle years, 24–26

hither (near) clipping distance, 34, 35
Hooke's law, 481– 482
Houdini system, 123
*Howard the Duck,* 26
human-computer interaction (HCI), 339
human representation. *See* articulated figures; virtual human representation
human visual system, 2–3
*Hummingbird,* 23
*Hunger,* 23, 28, 127
Hurd, Earl, 7
hybrid method for gaseous phenomena, 298

## I

identity matrix
- 4x4 matrix, 38
- defined, 410, 411

IK. *See* inverse kinematics
image compression
- codecs, 500– 501
- compression/decompression standards, 498–500
- DCT (discrete cosine transform), 500
- for digital video, 21
- fractal, 500
- interframe, 499
- intraframe, 499
- lossless vs. lossy, 499
- LZW, 500
- run-length encoding (RLE), 499–500, 501
- symmetric vs. asymmetric, 499
- vector quantization, 500
- wavelet, 500

image space
- eye space transformation to, 35–36, 37, 47
- overview, 35–36
- transformation to screen space, 36, 37, 47– 48

implicit equations, 453
implicit functions
- for cloud modeling, 303–304, 305–306
- density functions, 303, 304, 305
- for energy minimization, 237
- for implicit surfaces, 261

implicit pivoting, 415
implicit primitives
- for cloud modeling, 303–304, 305–306, 308
- implicit surfaces and, 261–263
- metaball, 261–262

implicit surfaces, 261–267
- animation using implicitly defined objects, 263–264
- basic formulation, 261–263
- collision detection, 264–265
- compound, 262, 263
- defined, 261
- deforming as collision response, 265–267
- distance-based, 262–263
- implicit function, 261
- implicit primitives, 261–263

- implicit surfaces (*continued*)
  - metaball implicit primitive, 261–262
  - penetration region, 265–266
  - propagation region, 265, 267
  - for virtual human representation, 355
- impulse, 224
- impulse force of collision, 222–227
  - backing up to point of impact, 222–224
  - computing impulse forces, 224–227
  - defined, 482
  - for point-plane collision, 222–224
- in-betweening, 13
- Indeo 3.2 codec, 500
- inertia tensors
  - angular momentum and, 208–209, 214
  - equations for, 214
  - overview, 487–488
- inertial frame, 479
- initial configuration for deforming objects, 125–126
- inking, 13
- Inman, V., 333
- inner product of two vectors, 420–422, 424
- insert editing, 19
- insertion point for muscles, 350, 351
- integrating ordinary differential equations (ODEs), 489–492
- integration, numerical. *See* numerical integration techniques
- intelligence, group types and, 242
- interframe compression, 499
- interior vertices of cloth grid, 362
- interlacing in NTSC format, 3
- internal state in autonomous behavior, 259
- interpolated points, 99
- interpolation
  - approximation vs., 64–65, 454
  - of arc length from parameter/arc-length pair table, 73–76
  - bilinear, 127, 128
  - choosing the appropriate function, 64–68
  - complexity issues, 65
  - continuity issues, 65–67
  - of curves, interpolating and approximating, 453–470
  - defined, 454
  - global vs. local control, 67
  - Hermite, 457–458
  - magnitude and, 97
  - nontriviality of, 63–64
  - orientation representation issues, 53, 55–56
  - overview, 63–68
  - polynomials and, 65
  - of rotations represented by quaternions, 97–102
  - shape interpolation (3D), 153–169
  - shape interpolation in key-frame systems, 117–120
  - sine interpolation for ease-in/ease-out, 86–87
  - for smoothing paths, 108–111
  - spherical linear interpolation (slerp), 99
  - summary, 169–170
  - varying the parameter at a constant rate, 68, 69
  - *See also* curves, interpolating and approximating
- *Interview with a Vampire,* 26
- intraframe compression, 499
- inverse Jacobian, 199–201
  - arm modeling and, 319–320
  - linkage changes and accuracy of, 201
  - pseudo inverse, 200–203
  - singularities and, 199–200
- inverse kinematics, 192–203
  - adding more control, 201–203
  - advantages of, 185
  - analytic solution, 193–194
  - arm modeling and, 319–320
  - defined, 185
  - dextrous workspace, 192
  - inverse Jacobian solution, 199–201
  - Jacobian, the, 194–201
  - local coordinate frames for, 185–192
  - number of solutions for, 192
  - overconstrained systems, 192
  - reachable workspace, 192
  - simple Jacobian solution, 197–199
  - skating counteracted by, 376
  - underconstrained systems, 192
- inverse of a matrix, 411–419, 426–427
- inverse of a quaternion, 59, 60
- Iwerks, Ub, 9

## J

- *Jack,* 22
- Jackson, Michael, 29
- Jacobian, the, 194–201
  - arm modeling and, 319–320
  - control expression with pseudo inverse, 201–203
  - defined, 193, 195
  - inverse, 199–201, 319–320
  - as mapping of velocities, 195
  - overview, 194–197
  - pseudo inverse, 200–203
  - simple example, 197–199
- *James and the Giant Peach,* 10, 386, 388
- jerky motion (strobing), 3
- Jodoin, Rene, 23
- joint angles
  - DH notation parameter, 186, 187, 188, 190, 191
  - end effector rotational change and, 196
  - for Jacobian applied to linked appendage, 196
  - for simple linkages, 194
- joints
  - ankle-toe joints, 332, 335
  - arm modeling, 318–321
  - ball-and-socket, 176, 177, 190–191
  - complex, 176–177
  - coordinated movement and, 324–325
  - degrees of freedom in, 176–177
  - DH notation for, 186–191
  - displacement in motion capture, 376–377
  - FFDs for, 358–359, 360
  - geometric deformation for virtual human figures, 356–357
  - hand modeling, 321–324
  - hierarchical modeling of, 177–183
  - human joint limits, 319
  - knee, 332, 334
  - local coordinate system for, 185–192
  - overview, 176–177
  - pelvic, 329–331
  - planar, 176–177, 186
  - prismatic, 176, 183, 192
  - revolute, 176, 181–183, 186, 192
  - in robotics, 176
  - shoulder joint modeling, 321, 322
  - telescoping, for leg dynamics, 337–338
  - tree structure for, 178–179
  - typical in computer animation, 176
  - variable rotations at, 181, 182
  - *See also* arcs; articulated figures; optical markers for motion capture; *specific types*
- Jones, Chuck, 9
- JPEG compression, 500, 501
- *Jumanji,* 27
- *Jurassic Park,* 27

## K

- Kajiya, J., 301
- Kass, M., 239, 240, 313, 368
- Kent, J., 156
- key. *See* occlusion mask
- key-frame systems, 116–120
  - articulation variables (avars), 116
  - linear interpolation for, 117–119

moving point constraints for immediate curves, 119
shape interpolation, 117–120
simple interface for interpolation of values, 116
simple key and intermediate frames, 117
as track based, 116
key frames, 13, 116
kinematic collision response, 217–218
kinematic control, 173–174
kinematic modeling, hierarchical. *See* hierarchical kinematic modeling
kinematics of walking, 333–336
joint angles, 333–335
lateral displacement of pelvis, 333
overview, 328
pelvis and feet constraints, 336
segment lengths and, 336
kinetic energy, 484
kinetic friction, 228, 482
*King Kong,* 10
Kluyskens, T., 301
knees
angles, 334
flexion, 332
optical markers for, 375, 376
*Knick Knack,* 25
Knowlton, Ken, 23
Kochanek, D., 465
Koga, Y., 320
Kondo, K., 320
Korein, J., 259, 400

**L**

L-systems
for animating plant growth, 281–283
bracketed, 277–278, 279
context-free, 275, 277
context-sensitive, 280
D0L-systems, 275
database amplification, 278
deterministic, 275
geometric interpretation of, 275–277
geometric replacement, 275
global time variable, 281, 282
local age value, 282
for modeling plants, 275–280
nondeterministic, 277–278
open, 283
parametric, 281
stochastic, 278–279
summary, 283
timed, 281–282
turtle graphics, 276–277
Lacquaniti, F., 320
Lagrangian formulations for gaseous phenomena, 296, 297–298
Lantz, Walter, 7, 9
laser scanners for facial modeling, 343, 346
Lasseter, J., 10
*Last Starfighter, The,* 25
lateral displacement of pelvis, 333
*Lawnmower Man,* 27
laws of motion, Newton's, 478–479
layered approach to human figure modeling, 358–361
leaf node, 178
leaves, plant, 274
LEDs for motion capture, 372
left-handed coordinate space, 32
left stance phase
of run cycle, 328–329, 330
of walk cycle, 328, 329
*LENGTH* function, 71, 72
length of a vector, 420, 425
levels of autonomous behavior, 259, 260
life span of particles, 242, 243, 244, 246
lift in flight modeling, 256–257
Lindenmayer, A., 276
lineage, plant, 274
linear acceleration, resting contact and, 230
linear equations
defined, 453
LU decomposition for solving, 411–419
linear force, bodies in free fall and, 212
linear interpolation
algebraic form, 455
geometric form, 454–455
for key-frame systems, 117–119
matrix representation, 455–456
for smoothing paths, 108–109
for wave modeling, 291
linear least squares problems, 419–420
linear momentum of bodies in free fall, 212–213
linear velocities
impulse forces and, 226
induced by joint axis rotation, 196–197
Jacobian and, 195–197
particle animation and, 244
link length parameter (DH notation), 186, 187, 188, 190, 191
link offset parameter (DH notation), 187, 188
link twist parameter (DH notation), 187, 188, 190, 191
links
in prismatic joints, 176
in revolute joints, 176
in robotics, 176
tree structure for, 177–179
*Lion King,* 27
lip-synch animation
facial models and, 339
sampling rate for, 3
sound track for, 13
*Little Nemo,* 6
local control
defined, 67
of flocking behavior, 247
interpolation and, 67
local coordinate system
for 2D grid deformation, 127
for ball-and-socket joint, 190–191
constructing the frame description, 191–192
for deforming objects, 125–127, 133–134
Denavit-Hartenberg (DH) notation, 186–190
for free-form deformation (FFD), 133–134
for hierarchical kinematic modeling, 185–192
for intermediate joints, 192
offset vector relative to, 175
for three-joint manipulator, 188–190
locomotion mechanisms. *See* mechanisms of locomotion
look-from point, 33. *See also* eye position (EYE)
look-to point, 33. *See also* center of interest (COI)
*Looker,* 24
lossless vs. lossy image compression, 499
low-albedo illumination, clouds and, 299–300
LU decomposition, 411–419
Lucasfilm, 25
**LUdecomp** procedure, 416, 417–418
**LUsubstitute** subroutine, 416, 418–419
*Luxo Jr.,* 25
LZW compression, 500

**M**

Maciejewski, A., 336
MAGI (Mathematical Application Group Inc.), 24
magic lanterns, 6
Magnenat-Thalmann, Nadia, 23, 120
magnetic tracking for motion capture, 369
manipulators, 176

map to sphere shape interpolation, 160– 166
- associating intersections with edges, 163–164
- configurations possible, 166
- current face, 164–165
- by fixing key vertices, 162–166
- generating new polyhedral definitions, 165
- intersection-candidate list for, 163–164
- locating the initial vertex, 163
- mapping back to the original object, 165
- projection method, 161
- recursive subdivision for, 166–169

*Mars Attacks,* 27
mass
- center of, 211–212, 220, 479
- penalty method for collision response, 218–219
- spring-mass-damper modeling of flexible objects, 232–235

matching topology shape interpolation, 157
Mathematical Application Group Inc. (MAGI), 24
matrices
- defined, 409
- determinant of a 3x3 matrix, 410– 411
- of direction cosines, 43–45
- identity matrix, 38, 410, 411
- inverse, 411– 419, 426– 427
- LU decomposition, 411– 419
- matrix representation of linear interpolation, 455–456
- matrix-vector multiplication, 426
- multiplying, 410, 425–426
- routines, 425–427
- singular value decomposition (SVD), 419– 420
- transpose of, 410, 411
- vector-matrix multiplication, 426
- *See also* transformation matrices

matrix-vector multiplication, 426
matte. *See* occlusion mask
Max, Nelson, 23, 24
May, Steve, 120, 122
Maya animation system, 301, 308–310, 320
McCay, Winsor, 6
Mech, R., 283
mechanisms of locomotion, 328–332
- ankle-toe joints, 332
- knee flexion, 332
- pelvic list, 331
- pelvic rotation, 329–331
- pelvic transport, 329, 330, 331
- run cycle, 328–329, 330
- walk cycle, 328, 329

MEL animation language, 121, 308
Méliès, Georges, 6
memory, modeling for autonomous behavior, 259
*Men in Black,* 26
Menache, A., 375
*merge* operation, 389
message passing, flocking behavior and, 251
Messmer, Otto, 7
metaball implicit primitive, 261–262
Metaxas, D., 301
Miller, G., 293, 294
MJPEG compression, 500, 501
model sheet, 13
momentum
- angular, 212, 213–214, 486– 487
- of bodies in free fall, 212–214
- conservation of, 484–485
- impulse and, 224

momentum of bodies in free fall, 212–214
- angular momentum, 212, 213–214
- conservation of, 208, 212–213
- inertia tensor and, 208–209, 214
- linear momentum, 212–213

monitors, double buffering for, 385–386, 387
morphing (2D), 143–153
- coordinate grid approach, 143–148
- defined, 143
- feature-based, 148–153
- films using, 28–29
- *See also* coordinate grid morphing; feature-based morphing

Mortenson, M., 456, 458, 464
motion along a curve, 68–97
- arc length computation, 69–84
- constant acceleration, 89–92
- curve fitting to position-time pairs, 94–97
- distance-time functions, 92–94
- ease-in/ease-out, 86–89
- parabolic ease-in/ease-out, 89–92
- specifying positions and speeds at specific times, 94
- speed control, 84–86

motion blur
- calculating, 3, 399– 401
- camera operation and, 399
- continuous approach, 399
- discrete approach, 399– 401
- human vision and, 3
- perception and, 399
- ray tracing and, 399– 400
- rendering issues, 398– 401
- temporal anti-aliasing and, 400

motion capture, 369–378
- 3D position reconstruction, 373–376
- active optical markers for, 372
- camera calibration, 372–373, 502–507
- defined, 369
- electromagnetic sensors for, 369
- fitting to the skeleton, 376–378
- modifying motion capture data, 378
- noise issues, 372
- objective of, 370
- optical markers for, 369–370, 371–372
- processing the images, 371–372
- summary, 378

motion constraints, 94, 96
Motion JPEG compression, 500, 501
motion of space-time particles, 239–240
motion perception
- motion blur, 3, 398– 401
- persistence of vision and, 2–3
- strobing, 3

motor activities in autonomous behavior, 259, 260
MOV (Quicktime) standard, 501
movement coordinator for autonomous behavior, 259, 260
moving picture standards. *See* standards for moving pictures
moving point constraints for immediate curves, 119
MPEG-1 standard, 500
MPEG-2 standard, 500
MPEG-4 standard, 345, 500
multiplane camera animation, 7–8
multiplicative identity, 59
multiplying matrices
- for compound transformations, 39
- matrix-vector multiplication, 426
- overview, 410
- premultiplying vs. postmultiplying, 38–39
- vector-matrix multiplication, 426

Muraki, S., 343
muscle models for facial animation, 350– 353
Muybridge, Eadweard, 6

## N

*n*-squared complexity in flocking behavior, 251
Nakamae, E., 301
National Television Standards Committee (NTSC) format
- interlacing, 3
- on-line editing for, 18–19

overview, 493–495
playback rate, 3
natural phenomena, 271–315
difficulties of modeling, 271–272
gaseous phenomena, 296–311
plants, 272–283
water, 283–296
Navier-Stokes equations, 292, 298
near (hither) clipping distance, 34, 35
negative velocities, 93
negotiating flocking motion, 249–250
New York Institute of Technology (NYIT), 23
Newton-Raphson iteration, 83
Newton's laws
of motion, 478– 479
of universal gravitation, 480– 481
Neyret, F., 301
*Nightmare Before Christmas,* 10
Nishita, T., 301
nodes in hierarchical linkages, 177–178
noise, motion capture issues, 372
noise functions
for cloud modeling, 303
overview, 471– 473
Nolan, Bill, 9
Noll, A. M., 23
nondeterministic L-systems, 277–278
nonlinear editing
defined, 17
on-line, 19–20
Nonuniform rational B-splines (NURBS), 354, 355, 469
nonuniform scale transformations, 40, 42
normal force, 482
normal vector perturbation for small-amplitude waves, 284
normalized arc length parameter, 84
normalizing a vector, 425
NS (Navier-Stokes) equations, 292, 298
NTSC format. *See* National Television Standards Committee (NTSC) format
numeric arc length computation, 77–79
numerical integration techniques, 488– 492
Euler method, 490– 491
function integration, 488– 489
Gauss-Legendre integration, 488, 489, 490
integrating ordinary differential equations (ODEs), 489– 492
Runge-Kutta methods, 491– 492
*Numerical Recipes: The Art of Scientific Computing,* 207, 416, 420
NURBS (Nonuniform rational B-splines), 354, 355, 469
NYIT (New York Institute of Technology), 23

## O

OBBs (oriented bounding boxes), 431
object space
defined, 33
overview, 33
transformation to world space, 33, 35, 37, 43, 46, 48
object stretching, 399
O'Brien, Willis, 10
observer position
defined, 33
view frustum and, 34, 35
*See also* eye position (EYE)
obstacles
plants' response to, 274
reaching around, 325–326
wave displacement from, 285–287
occlusion mask
alpha channel for, 390, 391–395
defined, 389
one-bit, 390, 391
ocean waves
Airy model for, 289
anatomy of, 288–289
breaking, 289
modeling, 289–291
shoreline and, 289
spray and foam modeling, 291
transport of water and, 288
wave refraction, 289
ODEs (ordinary differential equations), integrating, 489– 492
off-line editing, 18
offset vector, for automatic camera control, 175
on-line editing
nonlinear, 19–20
overview, 18–19
opaquing, 13
open L-systems, 283
optical markers for motion capture
3D position reconstruction, 373–376
active markers, 372
fitting to the skeleton, 376–378
number required, 375, 376
overview, 369–370
processing the images, 371–372
ordinary differential equations (ODEs), integrating, 489– 492
orientation along a path, 102–108
COI for camera path following, 106–108
Frenet frame for, 103–106
quality of motion vs., 173
orientation of bodies in free fall, 209–211
orientation representation, 51–60
4x4 transformation matrix for, 51–53
along a path, 102–108
axis-angle, 57–58
of bodies in free fall, 209–211
Euler angle, 56–57
fixed angle, 42– 43, 53–56
interpolation issues, 53
quality of motion vs., 173
quaternions, 58–60, 97–102
oriented bounding boxes (OBBs), 431
orthonormalization, 50– 51, 52
oscillatory motion, 485
outer product of two vectors, 422– 424
**over** operator for compositing
with pixel depth information, 395–396, 398
without pixel depth information, 389, 391–394
overconstrained systems, 192
overlapping action, 11
overlay plane, 389

## P

painting, 13
PAL standard, 495
panning, 7
parabolic blending
first-order continuity and, 66
interpolation and, 68
overview, 462– 463
parabolic ease-in/ease-out, 89–92
parabolic interpolation for smoothing paths, 110– 111
parallax effect, multiplane camera and, 7–8
parameterization by arc length, 456
parameterized models for facial animation, 344–345, 348–350
parametric continuity, 454
parametric curves, derivative computation for, 456
parametric equations, 453
parametric L-systems, 281
parametric position function, 237
parametric values
parameter, arc length pairs (table), 74
sampling to estimate arc length, 73–76
parent node, 178
Parent, R., 140, 156, 159, 160, 358
Park, Nick, 10
Parke, F., 22, 340, 344–345, 349
partial pivoting, 415

particle-based method for gaseous phenomena, 297–298
particle pair forces, 246
particle-plane collision
  impulse force for, 222–224
  kinesthetic response for, 217–218
particle systems, 242–246
  animating particles, 244
  assumptions for, 242
  attributes of particles, 244
  characteristics of, 242
  cloud animation and, 308
  defined, 241
  emergent behavior, 241
  films using, 27
  for fire modeling, 310–311
  forces, 246
  frame of motion computation for, 242–243
  for fuzzy objects, 243
  generating particles, 243
  life span of particles, 242, 243, 244, 246
  overview, 241
  randomness in, 242, 243
  rendering particles, 244–245
  representation, 245
  terminating particles, 244
  updating particle states, 245
  updating system status, 245
patch virtual human representations, 354, 355
path following, 102–116
  center of interest for, 106–108
  determining a path along a surface, 114–116
  Frenet frame for, 103–106
  issues for, 102
  orientation along a path, 102–108
  smoothing a path, 108–114
path planning for reaching around obstacles, 325–326
PDCs (proportional derivative controllers), 235–236
Peachey, D., 289
pelvis
  constraints satisfied by inverse kinematics, 336
  hip angles, 334
  lateral displacement of, 333
  pelvic list, 331
  pelvic rotation, 329–331
  pelvic transport, 329, 330, 331
penalty method for collision response, 218–220
  described, 217
  disadvantages of, 219
  mass and, 218–219
  penalty spring, 218–220
  polyhedra objects and, 220
  velocity and, 219
  as virtual spring, 235
pencil tests, 13, 14
penetration region of implicit surfaces, 265–266
perception
  flocking behavior and, 247, 248
  motion blur and, 399
  overview, 2–3
persistence of vision
  defined, 2
  early animation devices based on, 4–6
  motion perception and, 2–3
perspective divide, 47
perspective foreshortening, 36
perspective matrix multiply, 47
perspective transformation
  defined, 35–36
  perspective divide, 47
  perspective foreshortening, 36
  perspective matrix multiply, 47
phenakistoscopes, 4, 6
photographic approaches to facial modeling, 343–344, 347
phototropism, 274
physics
  of clouds, 299
  in flocking behavior, 247
  group types and, 242
  motion equations, computer animation and, 204
physics primer, 476–488
  angular momentum, 486–487
  center of mass, 479
  centrifugal force, 483–484
  centripetal force, 481
  circular motion, 477–478
  conservation of energy, 484
  conservation of momentum, 484–485
  contact forces, 481–483
  damping force, 486
  equilibrium, 480
  gravity, 480–481
  inertia tensors, 487–488
  inertial frame, 479
  kinetic energy, 484
  Newton's laws of motion, 478–479
  oscillatory motion, 485
  position, velocity, and acceleration, 476–477
  potential energy, 484
  torque, 480
  work, 484
piecewise properties of curves, 454
pinhole camera model, 372–373
pitch in flight modeling, 255, 256, 257
Pixar, 10, 15, 17, 25
pixel depth information
  compositing with, 388, 395–398
  compositing without, 389–395
planar joints
  defined, 176–177
  DH notation for, 186
planar polyhedra
  collision detection for, 220–222
  impulse force of collision for, 222–227
  particle-plane collision, 217–218, 222–224
  penalty method for collision response and, 220
  swept volume for, 222, 223
planes relative to the human figure, 318
planetary systems, hierarchical modeling of, 175–176
planner for autonomous behavior, 259, 260
plants, 272–283
  animating growth, 280–283
  botany background for, 273–274
  branching topology of, 272–273
  buds, 274
  cell growth in, 274
  cellular descent, 274
  challenges for computer animation, 272
  herbaceous, 273
  interacting with the environment, 282–283
  L-systems, 275–280
  leaves, 274
  lineage, 274
  obstacles, response to, 274
  roots, 273
  stems, 274
  structural components of, 273–274
  systems for modeling, 272
  tropisms, 274
  woody, 273
  *See also* L-systems
playback rate, 3
point of attachment for muscles, 350
point of insertion for muscles, 350, 351
point-plane collision. *See* particle-plane collision
point-to-fixed-point constraints, 238
point-to-point constraints, 238
point transformation using vector-matrix multiplication, 447
points, fitting curves to a given set, 469–470
polygon, area of, 429, 430
polygon mesh surfaces, determining paths along, 114–116
polygonal facial models, 341–342
polygonal human figures, 354
polyline, 129, 454

polyline deformation, 129–130
  mapping to the polyline, 129
  repositioning the polyline, 129–130
polynomial equations, 453
polynomials, interpolation ease and, 65
pose, 185
pose-to-pose technique, 11–12
pose vector
  calculating analytically, 192, 193–194
  defined, 185
position
  equations for, 476
  *See also* camera position; eye position (EYE); observer position
position-time constraints, 96
position-time pairs, curve fitting to, 94–97
positional (zeroth-order) continuity, 65–66, 453, 454
positive afterimage, 2
postmultiplying
  for compound transformations, 39
  premultiplying vs., 38–39
potential energy, 484
Potts, W., 248
praxinoscopes, 4, 6
predecessor in D0L-systems, 275
preliminary story, 12
premultiplying
  alpha channel and, 395
  arc transformations, 184
  for compound transformations, 39
  postmultiplying vs., 38–39
Press, W., 207
*Prince of Egypt,* 28
principles of computer animation, 10–12
prismatic joints
  defined, 176
  frames for intermediate joints, 192
  in human figures, 183
production. *See* animation production
production rules for D0L-systems, 275
propagation region of implicit surfaces, 265, 267
proportional derivative controllers (PDCs), 235–236
proximal, defined, 318
Prusinkiewicz, P., 276, 283
pseudo inverse Jacobian, the
  control expression with, 201–203
  for matrices not square, 200–201

## Q

quadratic equations, 453
quadratic programming, resting contact and, 231
qualitative dynamics, 308
quality of motion, orientation and positioning vs., 173
quaternions, 58–60, 450–452
  addition, 59
  arithmetic, 450
  axis-angle representation and, 58–59
  conversions between rotation matrices and, 451–452
  defined, 58
  interpolation of rotations represented by, 97–102
  inverse of, 59, 60
  multiplication, 59
  multiplicative identity, 59
  rotating vectors using, 59–60, 450–451
  rotation representation using, 60
  unit, 59, 97–99
Quicktime (MOV) standard, 501

## R

radially symmetric standing waves, 284, 285
radioulnar joint, 321, 322
Ralson, H., 333
randomness, 470–475
  noise functions, 303, 471–473
  in particle systems, 242, 243
  random number generator, 474–475
  turbulence functions, 303, 474
  uses for, 470–471
raster displays, 21
ray casting
  defined, 36
  display pipeline transformations vs., 36, 38
ray tracing, motion blur and, 399–400
reachable workspace
  defined, 192
  for simple linkage, 193
reaching and grasping, 318–327
  arm modeling, 318–321
  comfort metrics, 326–327
  coordinated movement, 324–325
  hand modeling, 321–324
  obstacles, reaching around, 325–326
  shoulder joint modeling, 321, 322
  strength criteria, 326–327
reaction in flocking behavior, 247
reasoning in flocking behavior, 247
reasoning unit for autonomous behavior, 259, 260
recursive subdivision for shape interpolation, 166–169
  mapping between vertices, 167–169
  overview, 166–167
  recursively dividing the meshes, 169
  splitting objects into front and back meshes, 167
  vertex-to-vertex interpolation, 169
*Red's Dream,* 25
Reeves, W., 119
rejoining (flocking behavior), 255
relative acceleration, resting contact and, 230
*render* function, 389
*RenderFarm,* 17
rendering issues
  for clouds, 299–300
  compositing, 386, 388–398
  double buffering, 385–386, 387
  drop shadows, 401–406
  motion blur, 398–401
  for particles, 244–245
  shortcuts for previewing, 14–15
*Rendez-vous à Montréal,* 23, 27
representation of particle systems, 245
resting contact, 228–231
  constraints on forces, 228, 229–230
  defined, 227
  difficulty of computing forces for, 228
  distance function for analysis, 228–229
  normal forces for, 228
  quadratic programming for, 231
  relative acceleration and, 230
revolute joints
  animating, 181–183
  defined, 176
  DH notation for, 186
  frames for intermediate joints, 192
Reynolds, C., 124, 246, 247, 249, 250, 251
right-handed coordinate space, 32
right stance phase
  of run cycle, 328–329, 330
  of walk cycle, 328, 329
right swing phase of walk cycle, 328
rigid body simulation, 203–231
  bodies in contact, 216–231
  bodies in free fall, 204–216
  overview, 203–204
  standard physics vs. computer animation and, 204
  update cycle, 204
  *See also* bodies in contact; bodies in free fall
rigid transformations
  defined, 41–42
  orthonormalization, 50–51, 52
Rijpkema, H., 322
RLE (run-length encoding), 499–500, 501

- robotics, hierarchical modeling and, 176
- Rogers, D., 458
- roll in flight modeling, 255, 256–257
- root arc, 178
- root node, 177
- roots, plant, 273
- rotate transformations
  - applying incremental rotation matrices to points, 48– 49
  - for arcs leading to nodes, 179
  - axis-angle representation, 57–58
  - axis-angle rotations, 449
  - basic, 40
  - constant-speed rotation, 98
  - converting between matrices and quaternions, 451– 452
  - direction of rotation, 41
  - Euler angle representation, 56–57
  - fixed angle representation, 42– 43, 53–56
  - forming the rotation matrix anew for each frame, 49–50
  - in hierarchical modeling, 179, 181–183
  - incrementally updating the rotation matrix, 49, 50
  - interpolation of rotations represented by quaternions, 97–102
  - left- vs. right-handed coordinate space and, 32
  - with nonuniform scale transformations, 42
  - quaternions for, 58–60, 97–102, 450– 451
  - rigid transformations and, 41
  - round-off error considerations, 48–51
  - in screw transformations, 188
  - shear transformations and, 42
  - with translate transformations, 41– 42
- rotoscoping, 7
- round-off error considerations, 48–51
- route sheet, 13
- run-length encoding (RLE), 499–500, 501
- Runge-Kutta integration for bodies in free fall, 207–208
- Runge-Kutta methods for solving ODEs, 491– 492

## S

- S-VHS videotape format, 496, 497
- sagittal plane, 318
- sampling rate, 3
- scale transformations
  - nonuniform, 40, 42
  - rigid transformations and, 41– 42
  - uniform, 40– 42, 46
- scapula, 322
- Scheepers, F., 321, 322, 359
- Schwartz, Lillian, 23
- scratch track, 13
- screen space, image space transformation to, 36, 37, 47– 48
- screw transformations, 188
- script, 12
- SECAM standard, 495
- second-order (curvature) continuity, 66, 454
- secondary actions, 11
- Sederberg, T., 125, 133
- sequences, 12
- sequential FFD organization, 136, 137
- *Sextone for President,* 27
- shadows
  - contact, 401
  - drop, 401– 406
- shape interpolation (3D), 153–169
  - axial slices, 159–160
  - control mechanism for, 156–157
  - correspondence problem for, 156
  - interpolation problem for, 156
  - in key-frame systems, 117–120
  - map to sphere, 160– 166
  - matching topology, 157
  - recursive subdivision, 166–169
  - star-shaped polyhedra, 157–159
  - summary, 169
  - surface-based techniques, 153, 155
  - terminology, 156
  - topological vs. geometrical information, 157
  - volume-based techniques, 156
- shear transformations, 42
- Shoemake, K., 99
- shots
  - defined, 12
  - test shots, 13, 14
- shoulder joint modeling, 321, 322
- SIGGRAPH (Special Interest Group on Computer Graphics), 1, 23, 24
- *Simpsons, The,* 28
- sine interpolation for ease-in/ease-out, 86–87
- singular value decomposition (SVD), 419– 420
- sinusoidal ease-in/ease-out, 87–89
- skating, 376
- skeleton, motion capture fitting to, 376–378
- *Skeleton Dance,* 9
- skin
  - muscle models for facial animation and, 350– 352
  - textures for facial animation, 345–347
  - in virtual human representation, 353, 358, 359, 361
- slerp (spherical linear interpolation), 99, 101–102
- slow in & slow out, 11. *See also* ease-in/ ease-out
- smell, modeling for autonomous behavior, 258–259
- Smith, Alvy Ray, 23
- smoke, volumetric modeling of, 301
- smoothing paths, 108–114
  - for automatic camera control, 175
  - by B-spline approximation, 114
  - with convolution kernels, 111–114
  - with cubic interpolation of adjacent values, 109–111
  - with linear interpolation of adjacent values, 108–109
- smoothness, interpolation and, 65–67
- Society of Motion Picture and Television Engineers (SMPTE) time code, 19
- Soechting, J. F., 320
- soft constraints
  - as additional forces, 231–232
  - defined, 231
  - for energy minimization, 237–239
  - for flexible objects, 232–235
  - *See also* enforcing soft and hard constraints
- "solids of revolution" objects, defined, 14
- Sony HDD-1000 DTV standard, 502
- Sorenson codec, 500, 501
- sound
  - importance of, 13
  - for lip-synch animation, 13
  - modeling for autonomous behavior, 258–259
  - scratch track, 13
- source point for waves, 284
- space curves
  - arc length integral and, 72–73
  - arc length parameterization of, 71
  - defined, 70– 71
  - finding *u* given *s,* 83–84
  - parameterized interpolation function as, 70
  - speed control curve independence from, 94
- space-time constraints, 239–241
  - numerical solution for, 240– 241
  - space-time particles, 239–240
- space-time particles, 239–240
- spaces (overview)
  - coordinate space, 32
  - eye space, 34–35
  - image space, 35–36
  - naming convention for, 33
  - object space, 33

ray casting and, 36, 38
screen space, 36
transformations in the display pipeline, 37, 46–48
world space, 33–34
*See also* display pipeline; *specific spaces*
Special Interest Group on Computer Graphics (SIGGRAPH), 1, 23, 24
*Species,* 26
speed
controlling for motion along a curve, 84–86
of curve tracing, 68
flocking behavior perception and, 248
parameterizing interpolating functions by arc length and, 68–69
*See also* angular velocities; linear velocities
speed control
constraints, 85
defined, 84
distance-time function for, 84–86
by fixing velocities at key points, 93
*See also* ease-in/ease-out
speed lines, 399
spherical coordinate system for arm modeling, 320
spherical linear interpolation (slerp), 99, 101–102
spline facial models, 341, 342
splitting (flocking behavior), 255
spray from waves, 291
springs
in cloth modeling, 366–367
Hooke's law, 481–482
modeling external forces, drawbacks for, 232–233
penalty, 218–220
spring-mass-damper modeling of flexible objects, 232–235
spring mesh skin model, 351, 352
virtual, 235–236
squash & stretch, 11
staging, 11
Stallings, George, 7, 9
Stam, J., 301, 311
stance
defined, 328
pelvic rotation and, 329–330, 331
pelvic transport and, 329, 330
run cycle phases, 328–329, 330
walk cycle phases, 328, 329
walking dynamics and, 337–338
standards for moving pictures, 493–502
analog, 493, 494
black-and-white video, 495
color incorporation into black-and-white signal, 495–496
compression/decompression, 498–500
digital television (DTV), 498, 501–502
digital video, 497–501
high-definition television (HDTV), 496
NTSC, 3, 18–19, 493–495
PAL, 495
SECAM, 495
videotape formats, 496, 497
Stanford University, 231
star-shaped polygons, 157
star-shaped polyhedra
defined, 157
shape interpolation, 157–159
*Star Trek: Deep Space Nine,* 27
*Star Trek II: The Wrath of Khan,* 27, 310
*Star Trek IV: The Voyage Home,* 28
*Star Wars,* 24
*Star Wars: Episode I,* 28
*Starship Troopers,* 27
static cameras, tracking objects with, 174
static friction, 227–228, 483
Staudhammer, John, 23
steady state flow, 296
steam, volumetric modeling of, 301
*Steamboat Willie,* 7
steer-to-avoid procedure for collision avoidance, 252–253
stems, plant, 274
stochastic L-systems, 278–279
stochastic perturbation for spray and foam modeling, 291
stop-motion techniques, 9–10
storage tubes, 21
story reel, 13
storyboard, 12–13
straight ahead technique, 11
strategies for autonomous behavior, 259, 260
stratus clouds, 299, 308
streaming video, 498
strength criteria for articulated figures, 326–327
stretching
modeling in cloth, 366
squash & stretch, 11
stride, 328
strobing, 3
subdivision surfaces
for facial models, 341, 342
for virtual human representation, 355
successor in D0L-systems, 275
*Sunstone,* 23
surface normal function, 237
surfaces
deforming around an articulated joint, 140–143
determining paths along, 114–116
implicit, 261–267
subdivision, 341, 342, 355
surface-based techniques for shape interpolation, 153, 155
virtual human representation, 358
*See also* implicit surfaces
Sutherland, Ivan, 22
SVD (singular value decomposition), 419–420
swept volume, 222, 223
symmetric vs. asymmetric compression/decompression, 499
synthetic actors, films using, 27–28

## T

tangential (first-order) continuity, 66, 453–454
tapering, global deformation for, 131
*Tarzan,* 28
technical background, 31–61
orientation representation, 51–60
spaces and transformations, 31–51
*Technological Threat,* 25, 28
telescoping joint for leg dynamics, 337–338
temporal anti-aliasing, 400
tension controls for curves, 465, 466, 467
terminating particles, 244
*Terminator II,* 26, 28
Terry, Paul, 7, 9
test shots, 13, 14
textures
cloud, 301
for facial animation, 345–347
for fire modeling, 311
Thalmann, Daniel, 23, 120
thaumatropes, 4, 5
three-dimensional computer graphics, films using, 28
thrust in flight modeling, 256
timed L-systems, 281–282
timing, 11
*Tin Toy,* 10, 25
*Titanic,* 27
Todd, F., 333
toe and ankle joints, 332, 335
optical markers for, 375, 376
*Tony de Peltrie,* 23, 27
topological information, shape interpolation and, 157
torque
bodies in free fall and, 212
defined, 208, 212, 480
overview, 480

torque (*continued*)
  penalty method of collision response and, 220
  resting contact and, 230
  in spring-mass-damper modeling of flexible objects, 235
  strength criteria for articulated figures and, 326–327
touch, modeling for autonomous behavior, 258
*Toy Story,* 15, 17, 28
*Toy Story 2,* 28
tracks. *See* articulation variables (avars)
transcendental equations, 453
transformation matrices
  4x4 matrix for, 32, 37–38
  for compound transformations, 39
  extracting transformations from, 45–46
  fixed angle representation, 42–43
  for global deformations, 130
  homogenous coordinates and, 36–39
  identity matrix (4x4), 38
  matrix of direction cosines, 43–45
  multiplying, 39
  for orientation representation, 51–53
  perspective matrix multiply, 47
  premultiplying vs. postmultiplying and, 38–39
  round-off error considerations, 48–51
  *See also* matrices
transformations, 31–51, 447–452
  affine, 40–42
  axis-angle rotations, 449
  basic, 40–42
  compounding, 39
  display pipeline, 32–36, 37, 46–48
  extracting from a matrix, 45–46
  eye space to image space, 35–36, 37, 47
  fixed angle representation for, 42–43
  in hierarchical modeling, 179–185
  homogenous coordinates for, 36–39
  image space to screen space, 36, 37, 47–48
  matrix of direction cosines for, 43–45
  object space to world space, 33, 35, 37, 43, 46, 48
  orthonormalization, 50–51, 52
  overview, 31–32
  perspective transformation, 35–36
  point, using vector-matrix multiplication, 447
  quaternions for, 58–60, 450–452
  ray casting for, 36, 38
  representing orientations, 42–45
  round-off error considerations, 48–51
  screw, 188
  types of, 31–32
  vector, using vector-matrix multiplication, 448–449
  world space to eye space, 34, 37, 46–47
  *See also* display pipeline; *specific types*
translate transformations
  basic, 40
  in hierarchical modeling, 179, 181, 183
  rigid transformations and, 41
  with rotate transformations, 41
  in screw transformations, 188
transpose of vectors or matrices, 410, 411
transverse plane, 318
tree structure for hierarchical modeling, 177–179, 181
triangle, area of, 428
*TRON,* 24–25
tropisms, plant, 274
*True Lies,* 26
truncated pyramid, 35
Truong, T., 368
turbulence functions
  for cloud modeling, 303
  overview, 474
turtle graphics, 276–277
*Twister,* 27
twists, global deformation for, 131

## U

unary forces in particle systems, 246
underconstrained systems, 192
uniform scale transformations
  extracting transformations from matrices and, 46
  overview, 40–41
  as rigid transformations, 41–42
unit quaternions, 59, 97–99
University of Utah, 22
up vector
  center of interest and, 33, 34, 106–107
  computing, 33
  default, 34
  defined, 33
  for orientation along a path, 103
  path for, 108
  user-supplied up direction vs., 33
updating
  orientation for bodies in free fall, 215–216
  particle state, 245
  particle system status, 245
  rigid body simulation update cycle, 204
  rotation matrix incrementally, 49, 50
user-supplied up direction, 33

## V

Van Der Beek, S., 23
vector displays, 21–22
vector-matrix multiplication
  routine for, 426
  transforming a point using, 447
  transforming a vector using, 448–449
vector quantization, 500
vectors
  components of, 420
  context and meaning of, 36
  cross product of, 422–424
  defined, 409
  dot product of, 420–422, 424
  length of, 420, 425
  matrix-vector multiplication, 426, 447–449
  normalizing, 425
  rotating using quaternions, 59–60, 450–451
  routines, 424–425
  transforming using vector-matrix multiplication, 448–449
  transpose of, 410, 411
  vector-matrix multiplication, 426
  view vector, 103, 107
velocities. *See* angular velocities; linear velocities
velocity-time curves
  distance-time functions and, 92–94
  nonintuitive results for user-specified values, 93
VHS videotape format, 496, 497
video
  digital, 20–21, 497–501
  streaming, 498
video art, 10
Video for Windows (AVI) standard, 501
Video I codec, 500, 501
video sequences for facial modeling, 345
videotape formats, 496, 497
view direction
  for camera path following with COI, 107
  overview, 33
  view frustum and, 34, 35
view frustum, 34, 35
view vector
  for camera path following with COI, 107

for orientation along a path, 103
virtual human representation, 353–358
body geometry, 354–356
clothing, 357
computation time required for, 353
free-form deformation (FFD) for, 358–359, 360
geometry data acquisition, 356
geometry deformation, 356–357
hair, 357–358
implicit surfaces for, 355
patch representations, 354, 355
polygonal representations, 354
subdivision surfaces for, 355
surface detail, 358
volumetric modeling for, 355
*See also* articulated figures
virtual springs, 235–236
PDCs, 235–236
penalty method for collision response, 218–220
uses for, 235–236
with zero rest lengths, 235
viscosity, 483
vision
modeling for autonomous behavior, 258
obstacle avoidance and, 325
visual characteristics of clouds, 299–300
Voight viscoelastic model, 351–352
volume-based techniques for shape interpolation, 156
volumetric cloud modeling, 301–310
animating volumetric procedural clouds, 308–310
cirrus clouds, 306–308
cumulus clouds, 302, 304–306
example system, 302–308
overview, 301–302
stratus clouds, 308
volumetric modeling for virtual human representation, 355
vortices in fluid dynamics, 296
Voss, R., 300
*Voyager,* 23

**W, X**

W-VHS DTV standard, 502
walk cycle, 328, 329
walking, 327–339
ankle-toe joints, 332
complexity of, 325, 327
cyclic and acyclic components of, 327
dynamics for realistic motion, 336–338
forward dynamic control, 338–339
kinematics of, 328, 333–336
knee flexion, 332
mechanisms of locomotion, 328–332
pelvic list, 331
pelvic rotation, 329–331
pelvic transport, 329, 330, 331
reaching vs., 327
run cycle, 328–329, 330
state transition diagrams for, 327–328
summary, 339
walk cycle, 328, 329
whole-body actions during, 325
*Wallace and Gromit,* 10
warping objects, 125, 126
water, 283–296
anatomy of waves, 288–289
challenges for computer animation, 283–284
downhill motion of, 292–295
ocean waves, 289–291
still waters and small-amplitude waves, 284–287
Waters, K., 340
wavelet compression, 500
waves
Airy model for, 289
anatomy of, 288–289
breaking, 289
displacement from obstructions, 285–287
ocean, 289–291
overlapping ripples, 284–285
radially symmetric standing waves, 284, 285
shoreline and, 289
small-amplitude, 284–287
source point for, 284
spray and foam modeling, 291
transport of water and, 288
wave refraction, 289
wind waves, 288
Weil, J., 362
Weiler, K., 156
Wein, Marceli, 23, 127
Wessler, Barry, 22
Whitney, John, Sr., 23
Whitted, Turner, 24
*Willow,* 26, 28
wire, 454
Witkin, A., 236, 237, 238, 239, 240
Wolberg, G., 148
woody plants, 273
work, equation for, 484
*Works, The,* 23
world space
camera calibration for motion capture, 372–373, 502–507
camera position, 33
defined, 33
eye position (EYE), 33
field of view, 34
hierarchical modeling and, 180–181
object space transformation to, 33, 35, 37, 43, 46, 48
observer position, 33
overview, 33–34
round-off error considerations, 48–51
transformation to eye space, 34, 37, 46–47
up vector, 33–34
user-supplied up direction, 33
view frustum, 34
visible area of, 34

**Y**

*y*-axis, as “up,” 32
yaw in flight modeling, 255, 256
yon (far) clipping distance, 34, 35
*Young Sherlock Holmes,* 26

**Z**

*z*-axis, as “up,” 32
Zeltzer, D., 259, 355
zeroth-order (positional) continuity, 65–66, 453, 454
**zmin** operator for compositing, 396
zoetropes, 4, 5
zoopraxinoscopes, 6

# About the Author

Rick Parent is an Associate Professor at Ohio State University, where he teaches computer graphics and computer animation. His research focuses on techniques for modeling and animating the human figure and on geometric modeling, with a current emphasis on implicit surfaces. Rick earned his Ph.D. in computer science from Ohio State University and his bachelor's degree, also in computer science, from the University of Dayton. Rick was awarded one of four "Outstanding Ph.D. Thesis" awards given nationally by the NCC in 1977. He has served on numerous SIGGRAPH committees as well as on the Computer Graphics International 2000 Program Committee and the Computer Animation 1999 Program Committee.