ELSEVIER

SERIES IN INTERACTIVE 3D TECHNOLOGY

ESSENTIAL MATHEMATICS FOR GAMES & INTERACTIVE APPLICATIONS

A PROGRAMMER'S GUIDE

JAMES M. VAN VERTH
LARS M. BISHOP

MORGAN KAUFMANN

This excellent volume is unique in that it covers not only the basic techniques of computer graphics and game development, but also provides a thorough and rigorous—yet very readable—treatment of the underlying mathematics. Fledgling graphics and games developers will find it a valuable introduction; experienced developers will find it an invaluable reference. Everything is here, from the detailed numeric issues of IEEE floating point notation, to the correct way to use quaternions and spherical linear interpolation to represent orientation, to the mathematics of collision detection and rigid-body dynamics.

—David Luebke, University of Virginia,
co-author of *Level of Detail for 3D Graphics*

When it comes to software development for games or virtual reality, you cannot escape the mathematics. The best performance comes not from superfast processors and terabytes of memory, but from well-chosen algorithms. With this in mind, the techniques most useful for developing production-quality computer graphics for Hollywood blockbusters are not the best choice for interactive applications. When rendering times are measured in milliseconds rather than hours, you need an entirely different perspective.

Essential Mathematics for Games and Interactive Applications provides this perspective. While the mathematics are rigorous and perhaps challenging at times, Van Verth and Bishop provide the context for understanding the algorithms and data structures needed to bring games and VR applications to life. This may not be the only book you will ever need for games and VR software development, but it will certainly provide an excellent framework for developing robust and fast applications.

—Ian Ashdown, President, ByHeart Consultants Limited

With *Essential Mathematics for Games and Interactive Applications,* Van Verth and Bishop have provided invaluable assistance for professional game developers looking to shore up weaknesses in their mathematical training. Even if you never intend to write a renderer or tune a physics engine, this book provides the mathematical and conceptual grounding needed to understand many of the key concepts in rendering, simulation, and animation.

—Dave Weinstein, Microsoft, Red Storm Entertainment

Geometry, trigonometry, linear algebra, and calculus are all essential tools for 3D graphics. Mathematics courses in these subjects cover too much ground, while at the same time glossing over the bread-and-butter essentials for 3D graphics programmers. In *Essential Mathematics for Games and Interactive Applications*, Van Verth and Bishop bring just the right level of mathematics out of the trenches of professional game development. This book provides an accessible and solid mathematical foundation for interactive graphics programmers. If you are working in the area of 3D games, this book is a "must have."

—Jonathan Cohen, Department of Computer Science,
Johns Hopkins University,
co-author of *Level of Detail for 3D Graphics*

Essential Mathematics for Games and Interactive Applications

A Programmer's Guide

Essential Mathematics for Games and Interactive Applications

A Programmer's Guide

James M. Van Verth
Red Storm Entertainment

Lars M. Bishop
Numerical Design Limited

Amsterdam Boston Heidelberg
London New York Oxford
Paris San Diego San Francisco
Singapore Sydney Tokyo

ELSEVIER

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Senior Editor	*Tim Cox*
Publishing Services Manager	*Simon Crump*
Production Editor	*Troy Lilly*
Editorial Assistant	*Richard Camp*
Cover Design	*Chen Design Associates*
Text Design	*Julio Esperas*
Composition	*Cepha Imaging Pvt. Ltd.*
Technical Illustration	*Dartmouth Publishing, Inc.*
Copyeditor	*Yonie Overton*
Proofreader	*John Bregoli*
Indexer	*Northwind Editorial Services*
Interior printer	*The Maple-Vail Book Manufacturing Group*
Cover printer	*Phoenix Color Corp.*

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data
Van Verth, James M.
Essential mathematics for games and interactive applications : a programmer's guide / James M. Van Verth and Lars M. Bishop.
p. cm. – (The Morgan Kaufmann series in interactive 3D technology)
Includes bibliographical references and index.
ISBN-13: 978-1-55860-863-4 ISBN-10: 1-55860-863-X (hardcover : alk. paper)
1. Computer games–Programming. 2. Three-dimensional display systems–Mathematics. I. Bishop, Lars M. II. Title. III. Series.
QA76.76.C672V47 2004
794.8′1711–dc22

2003028267

ISBN-13: 978-1-55860-863-4
ISBN-10: 1-55860-863-X

For information on all Morgan Kaufmann publications, visit our Web site at *www.mkp.com.*

Printed in the United States of America

05 06 07 08 5 4 3 2

Dedications

To Harry, Mur, and Fiona: my past, present, and future. —Jim

To Dad and Mom (Steve and Helene Bishop); your love and support have always been by my side. Thank you. —Lars

About the Authors

James M. Van Verth is a founding member of Red Storm Entertainment, a division of Ubisoft, where he has been a lead engineer for six years. For the past five years he also has been a regular speaker at the Game Developers Conference, teaching the all-day tutorial, "Essential Math for Programmers," on which this book is based. He began his game industry career at Virtus Corporation, working as a sound and graphics engineer for the title *Tom Clancy: SSN*. His first position at Red Storm was as project lead and designer of *Tom Clancy's Politika*, the first commercial Java game. This was followed by the land warfare game *Force 21* where he acted as lead engineer, focusing on 3D graphics, vehicle physics, and pathfinding. His latest role at Red Storm is as rendering technology lead for a well-known squad combat franchise. His background includes a B.A. in mathematics and computer science from Dartmouth College, an M.S. in computer science from the State University of New York at Buffalo, and an M.S. in computer science from the University of North Carolina at Chapel Hill. This is his first book.

Lars M. Bishop is the Chief Technology Officer for Numerical Design Limited (NDL). Since 1996, he has specialized in real-time 3D game rendering technologies at NDL. He was a founding member of the team that created NDL's popular NetImmerse and Gamebryo 3D game engines, which are used in over 50 games, such as Bethesda Softworks' *Morrowind* and Mythic Entertainment's *Dark Age of Camelot*. Lars is currently working on the development of next-generation NDL products, specifically 3D engines and tools for handheld devices. He holds a B.S. in mathematics and computer science from Brown University and an M.S. in computer science from the University of North Carolina at Chapel Hill.

Contents

Preface

Writing a book is an adventure. To begin with, it is a toy and an amusement; then it becomes a mistress, and then it becomes a master, and then a tyrant. The last phase is that just as you are about to be reconciled to your servitude, you kill the monster, and fling him out to the public. —Sir Winston Churchill

The Adventure Begins

As humorous as Churchill's statement is, there is a certain amount of truth to it; writing this book was indeed an adventure. There is something about the process of writing, particularly a nonfiction work like this, that forces you to test and expand the limits of your knowledge. We hope that you, the reader, benefit from our hard work.

How does a book like this come about? Many of Churchill's books began with his experience—particularly his experience as a world leader in wartime. This book had a more mundane beginning: Two engineers at Red Storm, separately, asked Jim to teach them about vectors. These engineers were 2D game programmers, and 3D was not new, but was starting to replace 2D at that point. Jim's project was in a crunch period, so he didn't have time to do much about it until proposals were requested for the annual Game Developers Conference. Remembering the engineers' request, he thought back to the classic "Math for SIGGRAPH" course from SIGGRAPH 1989, which he had attended and enjoyed. Jim figured that a similar course, at that time titled "Math for Game Programmers," could help 2D programmers become 3D programmers.

The course was accepted, and together with a co-speaker, Marcus Nordenstam, Jim presented it at GDC 2000. The following years (2001–2002) Jim taught the course alone, as Marcus had moved from the game industry to the film industry. The subject matter changed slightly as well, adding more advanced material such as curves, collision detection, and basic physical simulation.

It was in 2002 that the seeds of what you hold in your hand were truly planted. At GDC 2002, another GDC speaker, whose name, alas, is lost to time, recommended that Jim turn his course into a book. This was an interesting idea, but how to get it published? As it happened, Jim ran into Dave Eberly at SIGGRAPH 2002, and he was looking for someone to write just that book for Morgan Kaufmann. At the same time, Lars was presenting some of the basics of rendering on handheld devices as part of a SIGGRAPH course. Jim and Lars discussed the fact that handheld 3D rendering had brought back some of the "lost arts" of 3D programming, and that this might be included in a book on mathematics for game programming.

Thus, a co-authorship was formed. Lars joined Jim in teaching the GDC 2003 version of what was now called "Essential Math for Game Programmers," and simultaneously joined Jim to help with the book, helping to expand the topics covered to include numerical representations. As we began to flesh out the latter chapters of the outline, Lars was finding that the advent of programmable shaders on consumer 3D hardware was bringing more and more low-level lighting, shading, and texturing questions into his office at NDL. Accordingly, the planned single chapter on "texturing and antialiasing" became three, covering a wider selection of these rendering topics.

By early 2003, we were furiously typing the first full draft of what is now before you. The experience was fascinating, sometimes frustrating, but ultimately deeply rewarding. Hopefully, this fascination and respect for the material will be conveyed to you, the reader. The topics in this book can each take a lifetime to study to a truly great depth; we hope you will be convinced to try just that, nonetheless!

Enjoy as you do so, as one of the few things more rewarding than programming and seeing a correctly animated, simulated, and rendered scene on a screen is the confidence of understanding *how* and *why* everything worked. When something in a 3D system goes wrong (and it *always* does), the best programmers are never satisfied with "I fixed it, but I'm not sure how"; without understanding, there can be no confidence in the solution, and nothing new is learned. Such programmers are driven by the desire to understand what went wrong, how to fix it, and learning from the experience. No other tool in 3D programming is quite as important to this process than the mathematical bases[1] behind it.

Those Who Helped Us Along the Road

In a traditional adventure the protagonists are assisted by various characters that pass in and out of the pages. Similarly, while this book bears the names

1. Vector or otherwise.

of two people on the cover, the material between its covers bears the mark of many, many more. We would like to thank a few of them here.

The folks at our publisher, Morgan Kaufman, were extremely patient and helpful, having undertaken the daunting task of leading two authors through the process of finishing their first book. In particular we wish to thank Tim Cox, our editor, and Stacie Pierce and Richard Camp, his assistants over the course of the book, who were patient beyond measure and willing to provide excellent guidance throughout the project. We would also like to acknowledge Troy Lilly of Elsevier and Sean Will of Darmouth Publishing for their invaluable assistance throughout the production process. Special thanks are due to Dave Eberly, our series editor, who read most of the book several times and provided great encouragement (and the occasional scolding) through the entire process, one he's been through firsthand several times.

Our reviewers were top-notch. Ian Ashdown, Steven Woodcock, John O'Brien, J.R. Parker, Neil Kirby, John Funge, and Michael van Lent reviewed the initial proposal document. Peter Norvig, Tomas Akenine-Möller, Steven Woodcock, and John Funge read an early draft of the first few chapters, and provided invaluable comments that significantly improved the direction and content of the material. The entire draft of the book was read by Ian Ashdown, Wes Hunt, Peter Lipson, Jon McAllister, and Travis Young. Despite having a tight deadline, they provided page after page of useful feedback, keeping us honest and helping us generate a better arc to the material. Several of them went *well* above and beyond the call of duty, providing detailed comments and even re-reading sections of the book that required significant changes. Finally, Clark Gibson, Joe Sauder, and Chris Stoy also deserve nods[2] for providing critiques of specific chapters.

Thanks are also due to several groups of people who received early versions of parts of the book via Jim's and/or Lars's lectures, including the attendees and reviewers of the "Essential Mathematics" course at GDC 2000–2003, the reviewers and attendees of the "Dynamic Media" course at SIGGRAPH 2002, and the students of Andy van Dam's CS123 course in the fall of 2002. Marcus Nordenstam (GDC 2000) and David Holmes (SIGGRAPH 2002) were a part of the lecture teams for these course presentations that fed this book, and provided much background that would filter into the outline of the book itself. In addition (being Lars's office mate at NDL), David Holmes provided a weekend and evening sounding board for several of the topics as they came together. Thanks also to Victor Brueggemann and Garner Halloran, who asked the questions that started this whole thing off five years ago.

Jim and Lars would like to acknowledge the folks at their respective jobs, Red Storm Entertainment and Numerical Design Limited, who were *very* understanding with respect to the time-consuming process of creating a book.

2. They've already eaten the cookies.

Also, thanks to the talented engineers at both companies who provided the probing discussions and great questions that led to and continually fed this book.

In addition, Jim would like to thank Mur, his long-suffering wife who dealt with an occasionally absent husband during a pregnancy and the first year of their baby's life; Fiona, the daughter who still has the decency to recognize her daddy; his sister, Liz, who provided illustrations for an early draft of this text; and his parents, Jim and Pat, who gave him the resources to make it in the world and introduced him to the world of computers so long ago.

Lars would like to thank Jen, his wife, who provided more emotional and technical support than anyone could ask for, and who put up with more "lost weekends" and "after the book is done" comments than either of us can count.

And lastly, we would like to thank you, the reader, for joining us on this adventure. May the teeth of this monster find fertile ground in your minds, and yield a new army of 3D programmers.

Introduction

The (Continued) Rise of 3D Games

Over the past decade or so (driven by increasingly powerful computer hardware), 3D games have expanded from custom-hardware arcade machines to the realm of "hardcore" PC games, on to consumer "set top" videogame consoles, and even onto handheld devices such as personal digital assistants (PDAs) and cellular telephones. This explosion in popularity has lead to a corresponding need for programmers with the ability to program these games. As a result, programmers are entering the field of 3D games and graphics by teaching themselves the basics, rather than a "classic" University graphics and mathematics education. At the same time, many University students are looking to move directly from school into the industry. These different groups of programmers each have their own set of skills and needs in order to make the transition. While every programmer's situation is different, we describe some of the more common situations in the paragraphs below.

Many existing, self-taught 3D game programmers have strong game experience and an excellent practical approach to programming, stressing visual results and strong optimization skills that can be lacking in university computer science programs. However, these programmers are sometimes less comfortable with the conceptual mathematics that form the underlying basis of 3D graphics and games. This can make developing, debugging, and optimizing these systems more of a "trial and error" exercise than would be desired.

Programmers who are already established in other specializations in the game industry, such as networking or user interfaces, are now finding that they want to expand their abilities into core 3D programming. While having experience with a wide range of game concepts, these programmers often need to learn or refresh the basic mathematics behind 3D games before continuing on to learn the applications of these principles to rendering and animation.

On the other hand, university students entering (or hoping to enter) the 3D games industry often ask what material they need to know in order to be prepared to work on these games. Younger students often ask what courses they should attend in order to gain the most useful background for a programmer in the industry. Recent graduates, on the other hand, often ask how their computer graphics knowledge best relates to the way games are developed for today's computers and game consoles.

We have designed this book to provide something for each of these groups of readers. We attempt to provide readers with a conceptual understanding of the mathematics needed to create 3D games, as well as an understanding of how these mathematical bases actually *apply* to games and graphics. The book provides not only theoretical mathematical background, but also many examples of how these concepts are used to affect how a game looks (how it is "rendered") and plays (how objects move and react to users). Each type of reader is likely to find sections of the book that, for them, provide mainly "refresher courses," a new understanding of the applications of basic mathematical concepts, or even completely new information. The specific sections that fall into each category for a particular reader will, of course, depend on the reader.

How to Read this Book

As with almost any technical book, how you should read this one depends on two basic questions, What do you know? and What do you want to learn? The twelve core chapters of the book are organized into four parts. The four parts cover core mathematics, rendering, animation, and simulation, respectively.

Part I, Core Mathematics

The basic mathematics (vectors, linear algebra, affine algebra, and numerical representations) are covered in Chapters 1 through 4. These chapters form the mathematical basis for all of the following sections. Some readers will have a passing familiarity with the topics in this section. However, most readers will want to start with these chapters, as many of the topics are covered in more conceptual detail than is often discussed in basic graphics texts. Readers new to the material will want to read in detail, while those who already know some linear algebra can use the chapters to fill in any missing background. All of these chapters form a basis for the rest of the book, and an understanding of these topics, whether existing or new, will be key to successful 3D programming.

Chapter 1 introduces vectors, points, and the operations we apply to them. Vectors and points are the building blocks of the geometry we will use to construct, render, and simulate our 3D objects.

Chapter 2 introduces the matrix, a powerful tool we will use to position, view, and animate objects in our 3D worlds.

Chapter 3 discusses special forms of matrices that define common ways of manipulating points and vectors.

Finally, Part I closes with a detailed look at computer number representations and how they can affect the way we implement 3D games.

Chapter 4 discusses the two common computer representations of the set of real numbers, fixed-point and floating-point. It also explains some issues that can cause either number system to break down and cause incorrect or inaccurate behavior or degraded performance in 3D applications.

Part II, Rendering

Chapters 5 through 8 explain the so-called rendering pipeline, from the way we represent visible objects in 3D games to the methods used to draw these objects to the display. A mixture of concepts, mathematics, and implementations, these chapters begin to show the direct applicability of the mathematics introduced in the first four chapters to the 3D games we see on the market today. While not every 3D programmer will work directly on rendering, concepts in these chapters, especially Chapters 5 and 6, which specialize in geometric representations and transformations, are applicable to other aspects of 3D games.

Chapter 5 applies the concepts of matrices and transformations to the creation of virtual cameras, to be used to view our 3D worlds.

Chapter 6 examines the details of how we will represent our 3D objects visually; how we will break them into simple pieces for rendering and how we will apply colors and images to their surfaces.

Chapter 7 explains how we add realism to our rendering by adding convincing, dynamic lighting.

Chapter 8 covers the basics of how 3D graphics systems actually draw geometry to the display.

The chapters in Part II also provide many small code examples and discussions of how most of these rendering concepts can be implemented via the OpenGL SDK.

Part III, Animation

Chapters 9 and 10 build upon the first four chapters to introduce the basics of animation. These chapters detail the methods used to move 3D objects smoothly over time between sets of desired positions and orientations or *key frames* using different methods of interpolation. The benefits and drawbacks of each interpolation method will be compared along the way.

Chapter 9 introduces the most basic concepts of animation, focusing on animation of the position of objects. Introducing the concepts of parametric curves and splines, it will show how to create smooth curves that allow objects to move in arcs that appear natural and convincing.

Chapter 10 continues with basic animation, this time focusing on animating the orientation of objects. It will introduce the quaternion, an extremely powerful object that can represent orientation and its animation in a flexible and efficient manner.

Part IV, Simulation

Chapters 11 and 12 step beyond prechoreographed animation and describe how to make objects interact dynamically. A key feature of many games, especially action, simulation, and sports games, these methods determine when objects collide and how they should react to one another when they do in order to behave in a convincing manner.

Chapter 11 surveys the wide range of techniques used to determine when a set of objects collide, emphasizing those that are fast and can trade off accuracy and efficiency.

Chapter 12 serves as a basic introduction to the simulation of the laws of physics, allowing games to include realistic motion that is computed on the fly, rather than pre-determined.

Appendices

In addition to the four major sections of the book, we have included two appendices. **Appendix A**, on trigonometry, provides a very brief review of the basic foundations of trigonometric functions, as well as an annotated listing of frequently used trigonometric identities. **Appendix B**, on calculus, provides a review of topics such as limits, derivatives, and integrals. While neither of these appendices can teach these topics to a reader who is unfamiliar with them, they are designed to provide a refresher to readers whose educations in the topics are many years removed from them.

Interactive Demo Applications

Source Code
Demo
Name

Three-dimensional games and graphics are, by their nature, not only visual but dynamic. While figures are indeed a welcome necessity in a book about 3D applications, interactive demos can be even more important. It is difficult to truly understand such topics as lighting, quaternion interpolation, or physical simulation without being able to see them work firsthand and to interact with these complex systems. This book includes a CD-ROM of source code and demonstrations that are designed to illustrate the concepts in a way that is analogous to the static figures in the book itself. Throughout the book, you will find references to interactive demos that may be found on the CD-ROM. Whenever a topic is illustrated with an interactive demo, a special icon like the one seen next to this paragraph will appear in the margin.

Support Libraries

Source Code
Library
Name

In addition to the source code for each of the demos, the CD-ROM includes the supporting libraries used to create the demos, with full source code. Often, code from these supporting libraries is excerpted in the book itself in order to explain how the particular concept is implemented. In such situations, an icon will appear in the margin to note where the library code may be found on the CD-ROM. This source code is designed to allow the reader to modify and experiment themselves, as a way of better understanding the way the code works.

The source code is written entirely in C++, a language that is likely to be familiar to most game developers. C++ was chosen because it is one of the most commonly used languages in 3D game development and because vectors, matrices, quaternions, and graphics algorithms decompose very well into C++ classes. In addition, C++'s support of operator overloading means that the math library can be implemented in a way that makes the code look very similar to the mathematical derivations in the text. However, in some sections of the text, the class declarations as printed in the book are not complete with respect to the code on the CD-ROM. Often, class members that are not relevant to the particular discussion (especially member variable accessor and "housekeeping" functions) have been omitted for clarity. These other functions may be found in the full class declarations/definitions on the CD-ROM.

Note that we have modified our mathematical notation slightly to allow our equations to be as compatible as possible with the code. Mathematicians normally start indexing with 1; for example, $P_1, P_2, \ldots, P_n$. This does not match how indexing is done in C++: `P[0]` is the first element in the array `P`.

To avoid this disconnect, in our equations we will be using the convention that the starting element in a list is indexed as 0; thus $P_0, P_1, \ldots, P_{n-1}$. This should allow for a direct translation from equation to code.

Math Libraries

All of the demos use a shared core math library called `IvMath`, which includes C++ classes that implement vectors and matrices of different dimensions, along with a few other basic mathematical objects discussed in the book. This library is designed to be useful to readers beyond the examples supplied with the book, as the library includes a wide range of functions and operators for each of these objects, some of which are beyond the scope of the book's demos.

The animation demos use a shared library called `IvCurves`, which includes classes that implement spline curves, the basic objects used to animate position, `IvCurves` is built upon `IvMath`, extending this basic functionality to include animation. As with `IvMath`, the `IvCurves` library is likely to be useful beyond the scope of the book, as these classes are flexible enough to be used (along with `IvMath`) in other applications.

Finally, the simulation demos use a shared library called `IvCollision`, which implements basic object intersection (collision) data structures and algorithms. Building on the `IvMath` library, this set of classes and functions forms not only the basis for the later demos in the book but also is an excellent starting point for experimentation with other forms of object collision and physics modeling.

Engine and Rendering Libraries

In addition to the math libraries, the CD-ROM includes a set of classes that implement a simple game-like application framework, basic rendering, input handling, and timer functionality. All of these functions are grouped under the heading of "game engine" functionality, and are located in the `IvEngine` library. The engine's rendering code takes the form of a set of renderer-abstraction classes that simplify the interfaces between the C++ classes in `IvMath` and the C-based, low-level rendering application programmer interface(s), or API(s). This code is included as a part of the engine library, `IvEngine`. It includes renderer setup, basic render-state management, and rendering of simple geometric primitives, such as spheres, cubes, and boxes.

Furthermore, a set of basic classes that implement a simple scene graph are included in the library `IvScene`. The classes in `IvScene` use and depend upon the functionality of the `IvCollision` library. As a result, to avoid unnecessary code dependencies, the scene graph classes were placed in their own library, rather than in `IvEngine`.

Since this book focuses on the mathematics and concepts behind 3D games, we chose not to center the discussion around a large-scale, general 3D rendering engine. Doing so would introduce an extra layer of indirection that would not serve the conceptual requirements of the book. Valuable real estate in the rendering chapters would be spent on background in the use of a particular engine — the one written for the book. For an example and discussion of a full, hierarchical rendering engine, the reader is encouraged to read Dave Eberly's *3D Game Engine Design* [27].

We have opted to implement our rendering system and examples using the multiplatform standard SDK, OpenGL [83]. We also use the OpenGL utility toolkit, GLUT, to implement cross-platform renderer setup and input handling, neither of which are core topics of this book.

Microsoft's DirectX [77] is arguably as popular as (or more popular than) OpenGL for PC game development. However, DirectX was avoided due to its platform dependence. Most of the mathematical content in this book, including the concepts presented in the rendering chapters (Chapters 5 through 8), are independent of the particular rendering API or high-level graphics engine. In addition, DirectX is mentioned in numerous places in Part II, Rendering, generally in places where DirectX provides an interesting contrast to OpenGL.

As mentioned, rendering methods and OpenGL are often not the core purpose of a given demo. In these cases, we use the renderer-abstraction code from `IvEngine` to avoid cluttering the mathematical examples. However, most of the demos in the rendering section of the book are designed to show how specific rendering features are implemented in OpenGL. In these cases, we use some of OpenGL's features and functions directly. These demos include a mixture of `IvEngine` code and direct OpenGL calls in order to show some of the more advanced features of OpenGL not needed in the more mathematically-focused demos.

References and Further Reading

Hopefully, this book will leave readers with a desire to learn even more details and the breadth of the mathematics involved in creating high-performance, high-quality 3D games. Wherever possible, we have included references to other books, articles, papers, and web sites that detail particular subtopics that fall outside the scope of this book. The full set of references may be found at the back of the book.

We have attempted to include references that the vast majority of readers should be able to locate. When possible, we have referenced recent and/or standard industry texts and well-known conference proceedings.

However, in some cases we have included references to older magazine articles and technical reports when we found those references to be particularly complete, seminal, or well-written. In some cases older references can be easier for the less experienced reader to understand, as they often tend to assume less "common knowledge" when it comes to computer graphics and game topics.

In the past, older magazine articles and technical reports were notoriously difficult for the average reader to locate. However, the Internet and digital publishing have made great strides toward reversing this trend. For example, the following sources have made several classes of resources far more accessible:

- The magazine most commonly referenced in this book, *Game Developer*, offers CD-ROMs that contain every issue of the magazine ever published. Copies of these CD-ROMs are available from *www.gdmag.com*. Several other technical magazines also offer such CD-ROMs.
- Technical societies are now placing major historical publications into their "digital libraries," which are often made accessible to members. The Association for Computing Machinery (ACM) has done this via their ACM Digital Library, which is available to ACM members. As an example, the full text of the entire collection of papers from all SIGGRAPH conferences (the conference proceedings most frequently referenced in this book) is available electronically to ACM SIGGRAPH members.
- Other papers and technical reports are often available on the Internet. The two most common methods of finding these resources are via publication portals such as Citeseer (*www.citeseer.com*) and via the authors' personal homepages (if they have them). Most of the technical reports referenced in this book are available online from such sources. Owing to the dynamic nature of the Internet, we suggest using a search engine if the publication portals do not succeed in finding the desired article.

For further reading, we suggest several books that cover topics related to this book in much greater detail. In most cases they assume that the reader is familiar with the concepts discussed in this book. Dave Eberly's *3D Game Engine Design* [27] discusses the design and implementation of a full game engine, focusing mostly on graphics and animation. Books by Gino van den Bergen [109] and Christer Ericson [34] cover topics in interactive collision detection. Finally, Eberly's *Game Physics* [30] provides a more advanced discussion of a wide range of physical simulation topics.

Part I

Core Mathematics

Chapter 1
Vectors and Points

1.1 Introduction

The two building blocks of most objects in our interactive digital world are points and vectors. Points represent locations in space, which can be used either as measurements on the surface of an object to approximate the object's shape (this approximation is called a model), or as simply the position of a particular object. We can manipulate an object indirectly through its position or by modifying its points directly. Vectors, on the other hand, represent the difference or displacement between two points. Both have some very simple properties that make them extremely useful throughout computer graphics and simulation.

In this chapter we'll discuss the properties and representation of vectors and points, as well as the relationship between them. We'll present; how they can be used to build up other familiar entities from geometry classes; in particular, lines, planes, and polygons. Because many problems in computer games boil down to examples in applied algebra, having computer representations of standard geometric objects built on basic primitives is extremely useful.

It is likely that the reader has a basic understanding of these entities from basic math classes but the symbolic representations used by the mathematician may be unfamiliar or forgotten. We will review them in detail here. We will also cover linear algebra concepts — properties of vectors in particular — that are essential for manipulating three-dimensional objects. Without a thorough understanding of this fundamental material, any work in programming 3D games and applications will be quite confusing.

1.2 Vectors

One might expect that we would cover points first since they are the building blocks of our standard model, but in actuality the basic unit of most of the mathematics we'll discuss in this book is the vector. We'll begin by discussing the vector as a geometric entity since that's primarily how we'll be using it and it's more intuitive to think of it that way. From there we'll present a set of vectors known as a vector space and show how using the properties of vector spaces allows us to represent geometric vectors in a form that allows us to manipulate them in the computer. We'll conclude by discussing operations that we can perform on vectors and how we can use them to solve certain problems in 3D programming.

1.2.1 Vectors as Geometry

A geometric *vector* $\mathbf{v}$ is an entity with magnitude (also called length) and direction and is represented graphically as a line segment with an arrowhead on one end (Figure 1.1). The length of the segment represents the magnitude of the vector, and the arrowhead indicates its direction. A vector whose magnitude is 1 is a *unit* or *normalized* vector and is shown as $\hat{\mathbf{v}}$. The zero vector $\mathbf{0}$ has a magnitude of zero but no direction.

Note that a vector does not have a location. To make some geometric calculations easier to understand we may draw two vectors as if they were attached or place a vector relative to a location in space. Despite this, it is important to remember that two vectors with the same magnitude and direction are equal, no matter where drawn on the page. For example, in Figure 1.1 the left-most and right-most vectors are equal.

In games we use vectors in one of two ways. The first is as a representation of direction. For example, a vector may indicate direction toward an enemy, toward a light, or perpendicular to a plane. The second meaning represents change. If we have an object moving through space, we can assign a velocity

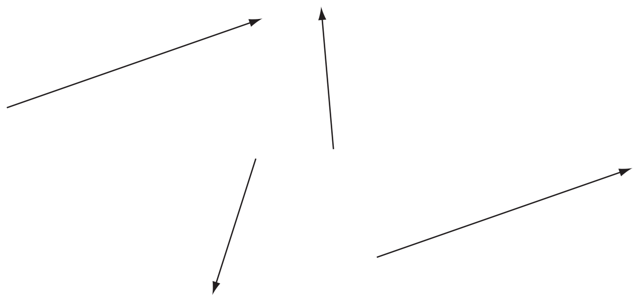


Figure 1.1 Vectors.

vector to the object, which represents change in position. We can displace the object by adding the velocity vector to the object's location to get a new location. Vectors can also be used to represent change in other vectors. For example, we can modify our velocity vector by another over a period of time; the second vector is called acceleration.

We can perform arithmetic operations on vectors just as we can with real numbers. One basic operation is addition. Geometrically, addition combines two vectors together into a new vector. If we think of a vector as an agent that changes position, then the new vector $\mathbf{u} = \mathbf{v} + \mathbf{w}$ combines the position-changing effect of $\mathbf{v}$ and $\mathbf{w}$ into one entity.

As an example, in Figure 1.2 we have three locations P, Q, and R. There is a vector $\mathbf{v}$ that represents the change in position or displacement from P to Q and a vector $\mathbf{w}$ that represents the displacement from Q to R. If we want to know the vector that represents the displacement from P to R, then we add $\mathbf{v}$ and $\mathbf{w}$ to get the resulting vector $\mathbf{u}$.

Figure 1.3 shows another approach, which is to treat the two vectors as the sides of a parallelogram. Then the sum of the two vectors is the diagonal that bisects them. Subtraction, or $\mathbf{v} - \mathbf{w}$, is shown by the other vector crossing the parallelogram. Remember that the difference vector is drawn from the second vector head to the first vector head—the opposite of what one might expect.

The algebraic rules for vector addition are very similar to real numbers:

1. $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ (commutative property)
2. $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ (associative property)
3. $\mathbf{v} + \mathbf{0} = \mathbf{v}$ (additive identity)
4. For every $\mathbf{v}$, there is a vector $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$ (additive inverse)

We can verify this informally by drawing a few test cases. For example, if we examine Figure 1.3 again, we can see that one path along the parallelogram

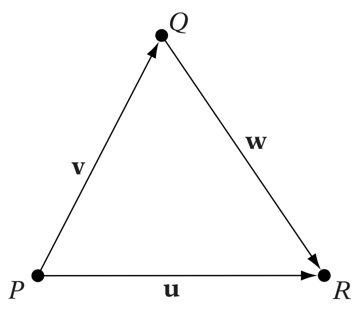


FIGURE 1.2 Vector addition.

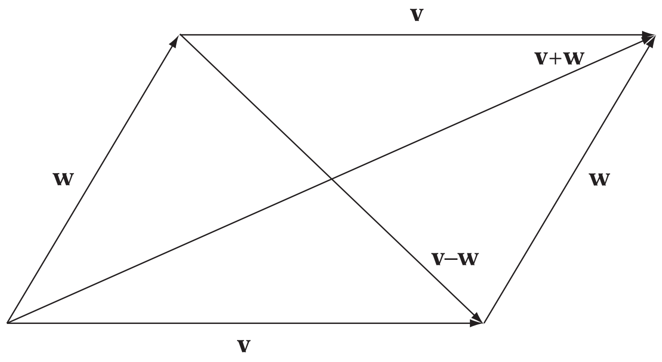


FIGURE 1.3 Vector addition and subtraction.

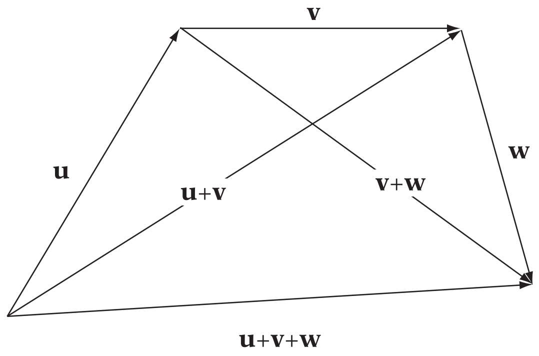


FIGURE 1.4 Associative property of vector addition.

represents $\mathbf{v} + \mathbf{w}$ and the other represents $\mathbf{w} + \mathbf{v}$. The resulting vector is the same in both cases. Figure 1.4 presents the associative property in a similar fashion.

The other basic operation is scalar multiplication, which changes the length of a vector by multiplying it by a single real value (Figure 1.5). Multiplying a vector by 2, for example, makes it twice as long. Multiplying by a negative value changes the length and points the vector in the opposite direction (the length remains nonnegative). Multiplying by 0 always produces the zero vector $\mathbf{0}$.

The algebraic rules for scalar multiplication should also look familiar:

5. $(ab)\mathbf{v} = a(b\mathbf{v})$ (associative property)
6. $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ (distributive property)

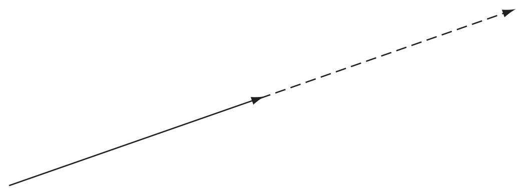


Figure 1.5 Scalar multiplication.

7. $a(\mathbf{v}+\mathbf{w}) = a\mathbf{v} + a\mathbf{w}$ (distributive property)
8. $1 \cdot \mathbf{v} = \mathbf{v}$ (multiplicative identity)

As with the additive rules, diagrams can be created that provide a certain amount of intuitive understanding.

1.2.2 Real Vector Spaces

In symbolic mathematics and (more important for our purposes) in the computer, representing vectors graphically is not convenient. The *linear space* or *vector space* provides a formal means of encapsulating the concepts that we've just covered and allows us to represent our vectors symbolically. This has a few advantages. First of all, this symbolic representation provides a means for storing vectors in the computer. And since it is an abstraction, we can use it for manipulating higher-dimensional vectors than we might be able to conceive of geometrically. It also can be used for representing entities that we wouldn't normally consider as vectors but that follow the same algebraic rules, which can be quite powerful. Finally, there are certain properties of vector spaces that will prove to be quite useful when we cover matrices and linear transformations.

To simplify our approach, we are going to concentrate on a subset of vector spaces known as *real vector spaces*, so called because their fundamental components are drawn from $\mathbb{R}$, the set of all real numbers. We usually say that such a vector space V is *over* $\mathbb{R}$. An element of $\mathbb{R}$ in this case is also known as a *scalar*. As a brief review, real numbers include 0; $\mathbb{Z}$, the set of all integers; $\mathbb{Q}$, the set of all rational numbers (fractions); and irrationals, numbers that can't be represented by fractions, like π and e.

For the most part we'll be representing real numbers in the computer using the floating point format. It is important to note that floating point is really only an approximation. For one thing, it can be used only to represent a finite set of all the real numbers. For another, improper ordering of floating point operations can lead to serious precision problems that don't occur with the infinite precision that real numbers provide. From time to time throughout this chapter and the rest of the book, we will touch on issues that will crop up when using floating point; for more details see Chapter 4.

So what is a real vector space? One example of a real vector space is simply $\mathbb{R}$. At first glance it may be difficult to see the correspondence between a real number and a vector, but as we'll see next, $\mathbb{R}$ does meet the criteria for a vector space.

Another real vector space is the set of all ordered pairs of real numbers, called $\mathbb{R}^2$. For now we can think of this as informally representing two-dimensional space—for example, diagrams on an infinitely extending, flat page. Symbolically, this is represented by

$$\mathbb{R}^2 = \{(x, y) \mid x, y \in \mathbb{R}\}$$

In this context, the symbol | means "such that" and the symbol $\in$ means "is a member of." So we read this as "The set of all possible pairs (x, y), such that x and y are members of the set of real numbers." As mentioned, this is a set of ordered pairs; (1.0, −0.5) is a different member of the set from (−0.5, 1.0).

We define $\mathbb{R}^3$ and $\mathbb{R}^4$ similarly as follows:

$$\mathbb{R}^3 = \{(x, y, z) \mid x, y, z \in \mathbb{R}\}$$
$$\mathbb{R}^4 = \{(w, x, y, z) \mid w, x, y, z \in \mathbb{R}\}$$

Like $\mathbb{R}^2$ these are ordered lists, where two members with the same values but differing orders are not the same. Again informally, we can think of elements in $\mathbb{R}^3$ as representing positions in three-dimensional space, which is where we will be spending most of our time. Correspondingly, $\mathbb{R}^4$ can be thought of as representing fourth-dimensional space, which is difficult to visualize spatially [1] (hence our need for an abstract representation) but is extremely useful for certain computer graphics concepts.

We can extend our definitions to $\mathbb{R}^n$, a generalized n-dimensional space over $\mathbb{R}$:

$$\mathbb{R}^n = \{(x_0, \ldots, x_{n-1}) \mid x_0, \ldots, x_{n-1} \in \mathbb{R}\}$$

The members of $\mathbb{R}^n$ are referred to as an n-tuple.

Up until now we've been casually referring to these real number spaces as vector spaces. For them to be proper vector spaces and not just organized lists of numbers, we need to define two specific operations on the elements that follow certain algebraic rules. The two operations should be familiar from our discussion of geometric vectors: they are addition and scalar multiplication.

1. Unless you are one of a particularly gifted pair of children [85].

We'll define these operations so that the vector space V has *closure* with respect to them, that is

1. For any $\mathbf{u}$ and $\mathbf{v}$ in V, $\mathbf{u} + \mathbf{v}$ is in V (additive closure)
2. For any a in $\mathbb{R}$ and $\mathbf{v}$ in V, $a\mathbf{v}$ is in V (multiplicative closure)

So formally, we define a real vector space as a set V over $\mathbb{R}$ with closure with respect to addition and scalar multiplication on its elements, where the following properties hold:

For all $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, $\mathbf{0}$ in V and all a, b in $\mathbb{R}$:

1. $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ (commutative property)
2. $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ (associative property)
3. There exists an element $\mathbf{0}$ such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$ (additive identity)
4. For every $\mathbf{v}$, there is an element $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$ (additive inverse)
5. $(ab)\mathbf{v} = a(b\mathbf{v})$ (associative property)
6. $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ (distributive property)
7. $a(\mathbf{v} + \mathbf{w}) = a\mathbf{v} + a\mathbf{w}$ (distributive property)
8. $1 \cdot \mathbf{v} = \mathbf{v}$ (multiplicative identity)

These are exactly the properties we stated previously for vector addition and scalar multiplication.

As an example, we can define addition in $\mathbb{R}^2$ as

$$(x_0, y_0) + (x_1, y_1) = (x_0 + x_1, y_0 + y_1)$$

and scalar multiplication as

$$a(x_0, y_0) = (ax_0, ay_0)$$

Using these definitions and the preceding algebraic axioms, it can be shown that $\mathbb{R}^2$ is a vector space. Similar operations can be defined for $\mathbb{R}^3$ and $\mathbb{R}^4$, as well as for $\mathbb{R}$ itself. Generalized over $\mathbb{R}^n$, we have

$$\begin{aligned}\mathbf{u} + \mathbf{v} &= (u_0, \ldots, u_{n-1}) + (v_0, \ldots, v_{n-1}) \\ &= (u_0 + v_0, \ldots, u_{n-1} + v_{n-1})\end{aligned}$$

and

$$\begin{aligned} a\mathbf{v} &= a(v_0, \ldots, v_{n-1}) \\ &= (av_0, \ldots, av_{n-1}) \end{aligned}$$

Suppose we have a subset W of a vector space V. We call W a *subspace* if it is itself a vector space when using the same definition for addition and multiplication operations. In order to show that a given subset W is a vector space, we only need to show that closure under addition and scalar multiplication holds; the rest of the properties are satisfied because W is a subset of V. For example, the subset of all vectors in $\mathbb{R}^3$ with $z = 0$ is a subspace, since

$$\begin{aligned} (x_0, y_0, 0) + (x_1, y_1, 0) &= (x_0 + x_1, y_0 + y_1, 0) \\ a(x_0, y_0, 0) &= (ax_0, ay_0, 0) \end{aligned}$$

The resulting vectors still lie in the subspace $\mathbb{R}^3$ with $z = 0$.

Note that any subspace must contain $\mathbf{0}$ in order to meet the conditions for a vector space. So the subset of all vectors in $\mathbb{R}^3$ with $z = 1$ is not a subspace since $\mathbf{0}$ cannot be represented. And while $\mathbb{R}^2$ is not a subspace of $\mathbb{R}^3$ (since the former is a set of pairs and the latter a set of triples), it can be embedded in a subspace of $\mathbb{R}^3$ by a mapping; for example, $(x, y) \to (x, y, 0)$.

It is important to understand that—despite the name—a vector space does not necessarily have to be made up of geometric vectors. What we have described is a series of sets of ordered lists, possibly with no relation to a geometric construct. As we will see, they *can* be related to the geometry, but the term *vector*, when used in describing members of vector spaces, is an abstract concept. As long as a set of elements can be shown to have the preceding arithmetic properties, we define it as a vector space and any element of a vector space as a vector. It is perhaps more correct to say that the geometric representations of two-dimensional and three-dimensional vectors that we use are visualizations that help us better understand the abstract nature of $\mathbb{R}^2$ and $\mathbb{R}^3$, rather than the other way around.

1.2.3 Linear Combinations and Basis Vectors

Our definitions of vector addition and scalar multiplication can be used to describe some special properties of vector spaces. Suppose we have a set S of n vectors, where $S = \{\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}\}$. We can combine these to create a new vector $\mathbf{v}$, using the function

$$\mathbf{v} = a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1}$$

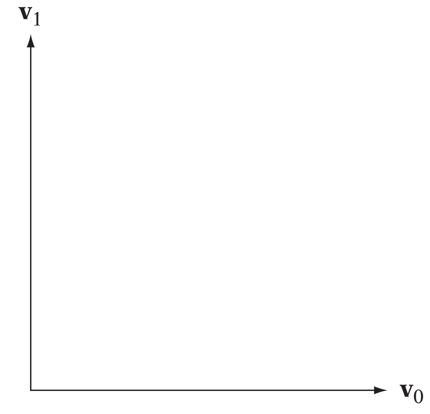


FIGURE 1.6 Two vectors spanning a plane.

for some arbitrary real scalars $a_0, \ldots, a_{n-1}$. This is known as a *linear combination* of all vectors $\mathbf{v}_i$ in S.

If we take all the possible linear combinations of all vectors in S, then the set T of vectors thus created is the *span* of S. We can also say that the set S spans the set T. For example, vectors $\mathbf{v}_0$ and $\mathbf{v}_1$ in Figure 1.6 span the set of vectors that lie on the surface of the page (assuming your book is held flat).

We can use linear combinations to define some properties of our initial set S. Suppose we can find a single vector $\mathbf{v}_i$ in S such that it's equal to a linear combination of other members of S. In other words,

$$\mathbf{v}_i = a_0\mathbf{v}_0 + \cdots + a_{i-1}\mathbf{v}_{i-1} + a_{i+1}\mathbf{v}_{i+1} + \cdots + a_{n-1}\mathbf{v}_{n-1}$$

If such a $\mathbf{v}_i$ exists, then we say that S is *linearly dependent*. If we can't find any such $\mathbf{v}_i$, then the vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}$ are *linearly independent*. An example of a linearly dependent set of vectors can be seen in Figure 1.7. Vector $\mathbf{v}_0$ is equal to the linear combination $-1 \cdot \mathbf{v}_1 + 0 \cdot \mathbf{v}_2$, or just $-\mathbf{v}_1$. Two linearly dependent vectors $\mathbf{v}$ and $\mathbf{w}$ are said to be *parallel*, that is, $\mathbf{w} = a\mathbf{v}$.

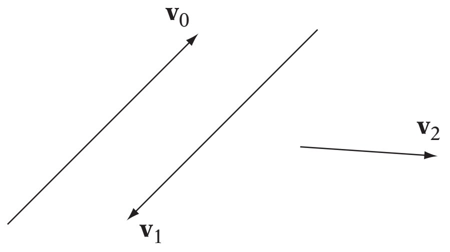


FIGURE 1.7 Linearly dependent set of vectors.

Now suppose that for a given vector space V, we can find a set β of n linearly independent vectors in V that span V. We call that β a *basis* for V, and each element of β is called a *basis vector*. There can be more than one basis for a given vector space, but they will always have the same number of elements. We formally define a vector space's *dimension* as equal to the number of basis vectors required to span it. So, for example, any basis for $\mathbb{R}^3$ will contain three basis vectors, and so it is (as we'd expect) a three-dimensional space. Among the many bases for a vector space, we define one as the *standard basis*. This standard set of basis vectors is represented as $\{\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}\}$, where

$$\begin{aligned}
\mathbf{e}_0 &= (1, 0, \ldots, 0)\\
\mathbf{e}_1 &= (0, 1, \ldots, 0)\\
&\vdots\\
\mathbf{e}_{n-1} &= (0, 0, \ldots, 1)
\end{aligned}$$

One property of a basis β is that for every vector $\mathbf{v}$ in V, there is a unique linear combination of the vectors in β that equal $\mathbf{v}$. So, using a general basis $\beta = \{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}\}$, there is only one list of coefficients $a_0, \ldots, a_{n-1}$ such that

$$\mathbf{v} = a_0\mathbf{b}_0 + a_1\mathbf{b}_1 + \cdots + a_{n-1}\mathbf{b}_{n-1}$$

Because of this, instead of using the full equation to represent $\mathbf{v}$, we can abbreviate it by using only the coefficients $a_0, \ldots, a_{n-1}$ and store them in an ordered n-tuple as $(a_0, \ldots, a_{n-1})$. Note that the coefficient values will be dependent on which basis we're using and will almost certainly be different from basis to basis. The ordering of the basis vectors is important: a different ordering will not necessarily generate the same coefficients for a given vector. For most cases, though, we'll be assuming the standard basis.

Let's take as an example $\mathbb{R}^3$, the vector space we'll be using most often. In this case the standard basis is $\{\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2\}$ or, as this basis is usually represented, $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$, where $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$, and $\mathbf{k} = (0, 0, 1)$. Their corresponding geometric representations can be seen in Figure1.8. Note that these vectors are of unit length and perpendicular to each other (we will define "perpendicular" more formally when we discuss dot products).

Using this basis, we can uniquely represent any vector $\mathbf{v}$ in $\mathbb{R}^3$ by using the formula $\mathbf{v} = a_0\mathbf{i} + a_1\mathbf{j} + a_2\mathbf{k}$. As with the basis vectors, in $\mathbb{R}^3$ we usually replace the general coefficients a_0, a_1, and a_2 with their more common representations x, y, and z, so

$$\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

We can think of x, y, and z as the amounts we move in the $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ directions, from the tail of $\mathbf{v}$ to its tip (see Figure 1.8). Since the $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ vectors are

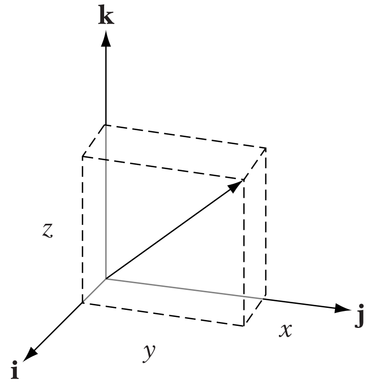


FIGURE 1.8 Standard 3D basis vectors.

known and fixed, we just store the x, y, z values and use them to represent our vector numerically. In this way a three-dimensional vector $\mathbf{v}$ is represented by an ordered triple (x, y, z). We can do the same for $\mathbb{R}^2$ by using as our basis $\{\mathbf{i}, \mathbf{j}\}$, where $\mathbf{i} = (1, 0)$ and $\mathbf{j} = (0, 1)$, and representing a two-dimensional vector as the ordered pair (x, y).

By doing this, we have also neatly solved the problem of representing our geometric vectors algebraically. By using a standard basis, we can use an ordered triple to represent the same concept as a line segment with an arrowhead. And by setting a correspondence between our algebraic basis and our geometric representation, we can guarantee that the ordered triple we use in one circumstance will be the same as the one we use in the other. Because of this, when working with vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$, we will use the two representations interchangeably.

Using our new knowledge of bases, it is possible to show that our previous definitions of addition and scalar multiplication for $\mathbb{R}^3$ are valid. For example, if we add two vectors in $\mathbb{R}^3$ $\mathbf{v}_0$ and $\mathbf{v}_1$ together:

$$\begin{aligned}\mathbf{v}_0 + \mathbf{v}_1 &= x_0\mathbf{i} + y_0\mathbf{j} + z_0\mathbf{k} + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k} \\ &= x_0\mathbf{i} + x_1\mathbf{i} + y_0\mathbf{j} + y_1\mathbf{j} + z_0\mathbf{k} + z_1\mathbf{k} \\ &= (x_0 + x_1)\mathbf{i} + (y_0 + y_1)\mathbf{j} + (z_0 + z_1)\mathbf{k}\end{aligned}$$

So, as we expect, to add two vectors we take each component in xyz order and add them:

$$(x_0, y_0, z_0) + (x_1, y_1, z_1) = (x_0 + x_1, y_0 + y_1, z_0 + z_1) \tag{1.1}$$

Scalar multiplication works similarly:

$$\begin{aligned} a\mathbf{v} &= a(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) \\ &= a(x\mathbf{i}) + a(y\mathbf{j}) + a(z\mathbf{k}) \\ &= (ax)\mathbf{i} + (ay)\mathbf{j} + (az)\mathbf{k} \end{aligned}$$

Again, this follows what we defined previously:

$$a(x, y, z) = (ax, ay, az) \tag{1.2}$$

1.2.4 Basic Vector Class Implementation

Source Code
Library
IvMath
Filename
IvVector3

Now that we've justified our ordered triple representation, we can talk about how we will store vectors in the computer. As we've mentioned many times, if we know the basis we're using to span our vector space, all we need to represent a vector are the coefficients of the linear combination. In our case we'll assume the standard basis and thus store the coefficients (or *components*) x, y, and z.

The following are some excerpts from the included C++ math library. For a vector in $\mathbb{R}^3$, our bare bones class definition is

```
class IvVector3
{
   inline IvVector3() {}
   inline IvVector3( float _x, float _y, float _z )
   {
       x = _x;
       y = _y;
       z = _z;
   }
   inline ~IvVector3() {}
   IvVector3( const IvVector3& vector );

   IvVector3& operator=( const IvVector3& vector );

   float x,y,z;

   ...
}
```

We can observe a few things about this declaration. First, we declared our member variables as a type `float`. This is the single-precision IEEE floating point representation for real numbers, which is currently standard for computer games. It uses a minimum of space with reasonable accuracy and is also hardware-accelerated on most platforms. Double-precision floating point uses twice as much space and may use a software implementation, which correspondingly can be much slower. On the other hand, because single precision is less precise, we have to be more careful about errors in precision. For more information on floating point, see Chapter 4.

The second thing to notice is that, like many vector libraries, we're making our member variables public. This is not usually recommended practice in C++; usually, the data is hidden and only made available through an inline member function. One motivation for such data hiding is to avoid unexpected side effects when changing a member variable. This is not an issue in the case of a vector since the data is so simple. However, this breaks another motivation for data hiding, which is that you can change your underlying representation without modifying nonlibrary code. This is a downside of what we are doing here, but one most vector libraries consider worthwhile for ease of coding. Consider:

```
v.x = 1.0f;
```

rather than one of the alternatives:

```
v.SetX(1.0f);
v.GetX() = 1.0f;
v.X() = 1.0f;
```

The class has a default constructor and destructor, which do nothing. The constructor could initialize the components to `0.0f` but doing so takes time, which adds up when we have large arrays of vectors (a common occurrence), and in most cases we'll be setting the values to something else anyway. For this purpose, there is an additional constructor which takes three floating point values and uses them to set the components. We can use the copy constructor and assignment operator as well.

Now that we have the data set up for our class, we can add some operations to it. The corresponding operator for vector addition is

```
IvVector3 operator+(const IvVector3& v0, const IvVector3& v1)
{
    return IvVector3( v0.x + v1.x, v0.y + v1.y, v0.z + v1.z );
}
```

Scalar multiplication is also straightforward:

```
IvVector3
operator*( float a, const IvVector3& vector)
```

```
{
    return IvVector3( a*vector.x, a*vector.y, a*vector.z );
}
```

Similar operators for post-multiplication and division by a scalar are also provided within the library; their declarations are

```
IvVector3 operator*( const IvVector3& vector, float scalar );
IvVector3 operator/( const IvVector3& vector, float scalar );
IvVector3& operator*=( IvVector3& vector, float scalar );
IvVector3& operator/=( IvVector3& vector, float scalar );
```

Some vector libraries use an alternative technique for creating vector classes known as template metaprogramming. This approach uses templates to trick the compiler into producing better optimized code. For example, rather than define an `operator+`, this technique creates a general template class called `Sum`, which can perform component-wise operations on our vectors. When we want to add a series of vectors, it creates the appropriate class, which performs the operation and converts it back to an `IvVector3`. A series of operations ends up with a nested set of templatized classes, which — assuming our compiler is any good — ends up as an optimized series of component-wise calculations.

We have decided not to use this for two reasons. First, and mainly, the implementation details tend to be less clear to those unfamiliar with vectors. What was once a simple `operator+()` becomes spread across classes. Second, the purpose of the technique is to minimize the number of operations when computing a complex equation such as

```
IvVector3 v1, v2, v3, v4;
v1 = 2.0f*v2 + 0.5f*v3 + v4;
```

In most cases we won't see equations this complex. For those who are interested, Blinn provides more details on template meta-programming for vector libraries in his collection *Notation, Notation, Notation* [13].

If you do need highly optimized code (in a tight loop, for instance), often it can be better to expand out the terms, which may simplify the equation, or write the assembly yourself. Many modern processors have a platform-specific SIMD instruction set for vectors — for example, SSE on Pentium and 3DNow! on AMD processors — which can perform several floating point operations in parallel. For clarity of code and because ours is a cross-platform library, we have chosen not to implement this, but for a platform-specific application this can be a significant optimization. More information on SSE and 3DNow! can be found in Chapter 4.

Now that we have a numeric representation for vectors and have covered the algebraic form of addition and scaling, we can add some new vector operations as well. As before, we'll focus primarily on the case of $\mathbb{R}^3$. Vectors in $\mathbb{R}^2$ and $\mathbb{R}^4$ have similar properties; any exceptions will be discussed in the particular parts.

1.2.5 Vector Length

We have mentioned that a vector is an entity with length or direction but so far haven't provided any means of measuring or comparing these quantities in two vectors. We'll see shortly how the dot product provides a way to compare vector directions. First, however, we'll consider how to measure a vector's magnitude.

There is a general class of size-measuring functions known as *norms*. A norm $\|\mathbf{v}\|$ is defined as a real-valued function on a vector $\mathbf{v}$ with the following properties:

1. $\|\mathbf{v}\| \geq 0$, and $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = \mathbf{0}$
2. $\|a\mathbf{v}\| = |a|\|\mathbf{v}\|$
3. $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$

We use the $\|\mathbf{v}\|$ notation to distinguish a norm from the absolute value function $|a|$.

An example of a norm is the Manhattan distance, also called the ℓ_1 norm, which is just the sum of the absolute values of the given vector's components:

$$\|\mathbf{v}\|_{\ell_1} = \sum_i |v_i|$$

One that we'll use more often is the Euclidean norm, also known as the ℓ_2 norm or just *length*. If we give no indication of which type of norm we're using, this is usually what we mean.

We derive the Euclidean norm as follows. Suppose we have a two-dimensional vector $\mathbf{u} = x\mathbf{i} + y\mathbf{j}$. Recall the Pythagorean theorem $x^2 + y^2 = d^2$. Since x is the distance along $\mathbf{i}$ and y is the distance along $\mathbf{j}$, then the length d of $\mathbf{u}$ is

$$\|\mathbf{u}\| = d = \sqrt{x^2 + y^2}$$

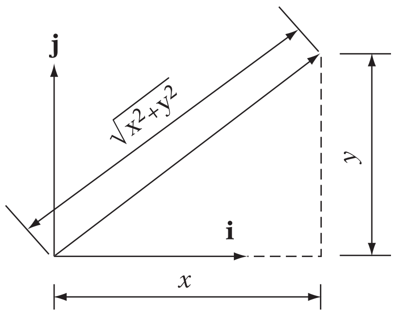


FIGURE 1.9 Length of 2D vector.

as shown in Figure 1.9. A similar formula is used for a vector $\mathbf{v} = (x, y, z)$, using the standard basis in $\mathbb{R}^3$:

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2} \tag{1.3}$$

And the general form in $\mathbb{R}^n$ with respect to the standard basis is

$$\|\mathbf{v}\| = \sqrt{v_0^2 + v_1^2 + \cdots + v_{n-1}^2}$$

We've mentioned the use of unit length vectors as pure indicators of direction; for example, in determining viewing direction or relative location of a light source. Often, though, the process we'll use to generate our direction vector will not automatically create one of unit length. To create a unit vector $\hat{\mathbf{v}}$ from a general vector $\mathbf{v}$, we *normalize* $\mathbf{v}$ by multiplying it by 1 over its length, or

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

This sets the length of the vector to $\|\mathbf{v}\| \cdot 1/\|\mathbf{v}\|$ or, as we desire, 1.

Our implementations of length methods (for $\mathbb{R}^3$) are as follows:

```
float
IvVector3::Length() const
{
    return ::IvSqrt( x*x + y*y + z*z );
}

float
IvVector3::LengthSquared() const
```

```
{
    return x*x + y*y + z*z;
}

IvVector3&
IvVector3::Normalize()
{
    float lengthsq = x*x + y*y + z*z;
    ASSERT( !::IsZero( lengthsq ) );
    if ( ::IsZero( lengthsq ) )
    {
        x = y = z = 0.0f;
        return *this;
    }

    float recip = ::IvInvSqrt( lengthsq );
    x *= recip;
    y *= recip;
    z *= recip;

    return *this;
}
```

Note that in addition to the mathematical operations we've just described, we have defined a `LengthSquared()` method. Performing the square root can be a costly operation, even on systems that have a special machine instruction to compute it. Often we're only doing a comparison between lengths, so it is better and certainly faster in those cases to compute and compare length squared instead. Both length and length squared are increasing functions starting at 0, so the results will be the same.

The `LengthSquared()` method also introduces some new functions which will be useful to us throughout the math library. The function `::IsZero()` is a precision-safe means of testing for near-zero values. We assume that if a floating point number is close enough to zero, it is considered essentially zero. It is much better to use that than to do a direct comparison with `0.0f` because of the inherent precision problems with floating point.

We also use our own square root functions `::IvSqrt()` and `::IvInvSqrt()` instead of `sqrtf()`. There are a number of reasons for this choice. As mentioned, the standard library implementation of square root is often slow. Rather than use it, we can use an approximation on some platforms, which is faster and accurate enough for our purpose. On other platforms there are internal assembly instructions that are not used by the standard library. In particular, there may be an instruction that performs the inverse square root, which is faster than calculating the square root and performing the floating point divide. Defining our own layer of indirection gives us flexibility and ensures that we can guarantee ourselves the best performance.

1.2.6 Dot Product

Now that we've considered vector length, we can look at vector direction. We begin by considering a set of functions known as *inner products*. An inner product is a concept, like a vector space, that is used to abstract away physical notions of geometry while maintaining similar properties.

For all $\mathbf{v}$, $\mathbf{w}$ in a real vector space V, we define an inner product $\langle \mathbf{v}, \mathbf{w} \rangle$ as a function returning a real scalar, with the following properties:

1. $\langle \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{w}, \mathbf{v} \rangle$ (symmetry)
2. $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$ (additivity)
3. $a\langle \mathbf{v}, \mathbf{w} \rangle = \langle a\mathbf{v}, \mathbf{w} \rangle$ (homogeneity)[2]
4. $\langle \mathbf{v}, \mathbf{v} \rangle \geq 0$ (positivity)
5. $\langle \mathbf{v}, \mathbf{v} \rangle = 0$ if and only if $\mathbf{v} = \mathbf{0}$ (definiteness)

A real vector space together with such a function is called an *inner product space*.

There is a particular inner product that can be tied to the physical world in ways that will prove to be very useful to us. It is called the *Euclidean inner product*, or more commonly, the *dot product*. It is probably the most useful vector operation for 3D games and applications. Instead of using the $\langle \cdot, \cdot \rangle$ form, the dot product of two vectors $\mathbf{v}$ and $\mathbf{w}$ is represented by $\mathbf{v} \cdot \mathbf{w}$. However, since it is an inner product, it still follows the same algebraic rules.

Given two vectors $\mathbf{v}$ and $\mathbf{w}$ with an angle θ between them, the dot product is defined as

$$\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\| \|\mathbf{w}\| \cos\theta \tag{1.4}$$

Using this equation, we can find a coordinate-dependent definition in $\mathbb{R}^3$ by examining a triangle formed by $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{v} - \mathbf{w}$ (Figure 1.10). The Law of Cosines[3] gives us

$$\|\mathbf{v} - \mathbf{w}\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{w}\|^2 - 2\|\mathbf{v}\| \|\mathbf{w}\| \cos\theta$$

2. Note that the leading scalar does not apply to both terms on the right-hand side; assuming so is a common mistake.

3. See Appendix A.

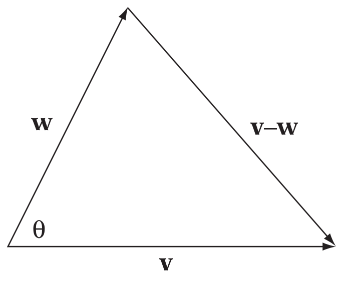


FIGURE 1.10 Law of cosines.

We can rewrite this as

$$-2\|\mathbf{v}\|\,\|\mathbf{w}\|\cos\theta = \|\mathbf{v}-\mathbf{w}\|^2 - \|\mathbf{v}\|^2 - \|\mathbf{w}\|^2$$

Substituting in the definition of vector length in $\mathbb{R}^3$ and expanding, we get

$$\begin{aligned}
-2\|\mathbf{v}\|\,\|\mathbf{w}\|\cos\theta &= (v_x - w_x)^2 + (v_y - w_y)^2 + (v_z - w_z)^2 \\
&\quad - (v_x^2 + v_y^2 + v_z^2) - (w_x^2 + w_y^2 + w_z^2) \\
-2\|\mathbf{v}\|\,\|\mathbf{w}\|\cos\theta &= -2v_x w_x - 2v_y w_y - 2v_z w_z \\
\|\mathbf{v}\|\,\|\mathbf{w}\|\cos\theta &= v_x w_x + v_y w_y + v_z w_z
\end{aligned}$$

So, to compute the dot product in $\mathbb{R}^3$, multiply the vectors component-wise, and then add:

$$\mathbf{v}\cdot\mathbf{w} = v_x w_x + v_y w_y + v_z w_z$$

Note that for this definition to hold, vectors **v** and **w** need to be represented with respect to the standard basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$. The general form for vectors **v** and **w** in $\mathbb{R}^n$, again with respect to the standard basis, is

$$\mathbf{v}\cdot\mathbf{w} = v_0 w_0 + v_1 w_1 + \cdots + v_{n-1} w_{n-1}$$

We can relate the dot product to the length function by noting that

$$\mathbf{v}\cdot\mathbf{v} = \|\mathbf{v}\|^2 \tag{1.5}$$

Whereas we began by defining the length and then the dot product, in more abstract inner product spaces we usually define the norm based on the inner product. This can be done by rewriting the equation as

$$\|\mathbf{v}\| = \sqrt{\langle \mathbf{v}, \mathbf{v}\rangle}$$

Of the two, equation 1.5 will be more useful to us.

As mentioned, the dot product has many uses. By equation 1.4, if the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$ in standard Euclidean space is 90 degrees, then $\mathbf{v} \cdot \mathbf{w} = 0$. So we define that two vectors $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, or *orthogonal,* when $\mathbf{v} \cdot \mathbf{w} = 0$. Recall that we stated that our standard basis vectors for $\mathbb{R}^3$ are orthogonal. We can now demonstrate this. For example, taking $\mathbf{i} \cdot \mathbf{j}$ we get

$$\begin{aligned}\mathbf{i} \cdot \mathbf{j} &= (1, 0, 0) \cdot (0, 1, 0) \\ &= 0 + 0 + 0 \\ &= 0\end{aligned}$$

It is possible, although not always recommended, to use equation 1.4 to test whether two unit vectors $\hat{\mathbf{v}}$ and $\hat{\mathbf{w}}$ are pointing generally in the same direction. If they are, $\cos\theta$ is close to 1, so $1 - \hat{\mathbf{v}} \cdot \hat{\mathbf{w}}$ is close to 0 (we use this formula to avoid problems with floating point precision). Similarly, if $1 + \hat{\mathbf{v}} \cdot \hat{\mathbf{w}}$ is close to 0, they are pointing in opposite directions. Performing this test only takes 6 floating point addition and multiplication operations. However, if $\mathbf{v}$ and $\mathbf{w}$ are not known to be normalized, then we need a different test: $\|\mathbf{v}\|^2\|\mathbf{w}\|^2 - (\mathbf{v} \cdot \mathbf{w})^2$. This takes 18 operations.

Note that for unit vectors:

$$\begin{aligned}1 - (\hat{\mathbf{v}} \cdot \hat{\mathbf{w}})^2 &= 1 - \cos^2\theta \\ &= \sin^2\theta\end{aligned}$$

and for non-unit vectors:

$$\begin{aligned}\|\mathbf{v}\|^2\|\mathbf{w}\|^2 - (\mathbf{v} \cdot \mathbf{w})^2 &= \|\mathbf{v}\|^2\|\mathbf{w}\|^2(1 - \cos^2\theta) \\ &= \|\mathbf{v}\|^2\|\mathbf{w}\|^2\sin^2\theta\end{aligned}$$

So assuming we use this, the method we use to test closeness to zero will have to be different for both cases.

In any case, using dot product for this test is not really recommended unless your vectors are pre-normalized *and* speed is of the essence. As $\cos\theta$ gets close to 0, it changes very little. Due to lack of floating point precision, the set of angles that might be considered "zero" is actually broader than one might expect. As we will see, there is another method to test for parallel vectors that is faster with non-unit vectors and has fewer problems with near-zero angles.

A more common use of the dot product is to test the angle between two vectors. If $\mathbf{v} \cdot \mathbf{w} > 0$, then we know the angle is less than 90 degrees. If $\mathbf{v} \cdot \mathbf{w} < 0$, then we know that the angle is greater than 90 degrees, and if $\mathbf{v} \cdot \mathbf{w} = 0$ then the angle is exactly 90 degrees (Figure 1.11). As opposed to testing for parallel vectors, this will work with vectors of *any* length.

For example, suppose that we have an AI agent that is looking for enemy agents in the game. The AI has a view vector **v** and a vector **t** which points toward an object in our scene. If $\mathbf{v} \cdot \mathbf{t} < 0$, then the object is behind us and therefore not visible to our AI (Figure 1.12).

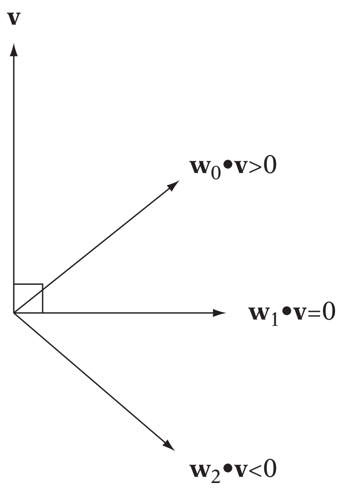

$\mathbf{v}$
$\mathbf{w}_0 \bullet \mathbf{v} > 0$
$\mathbf{w}_1 \bullet \mathbf{v} = 0$
$\mathbf{w}_2 \bullet \mathbf{v} < 0$

FIGURE 1.11 Dot product as measurement of angle.

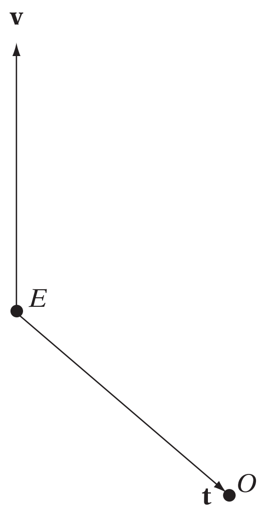


FIGURE 1.12 Measuring angle to target.

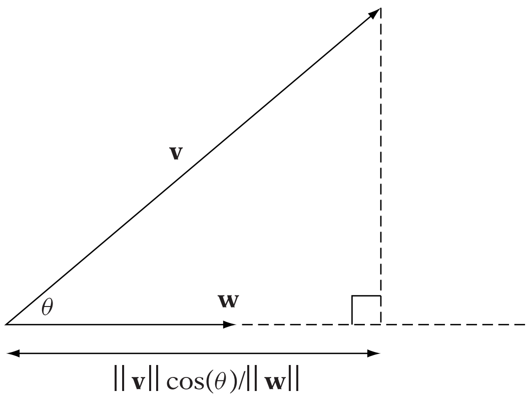


FIGURE 1.13 Dot product as projection.

Equation 1.4 allows us to use the dot product in another manner. Suppose we have two vectors **v** and **w**, where $\mathbf{w} \neq 0$. We define the *projection* of **v** onto **w** as

$$\text{proj}_{\mathbf{w}}\mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{w}\|^2}\mathbf{w}$$

This gives the part of **v** which is parallel to **w**, which is the same as dropping a perpendicular from the end of **v** onto **w** (Figure 1.13).

We can get the part of **v** which is perpendicular to **w** by subtracting the projection:

$$\text{perp}_{\mathbf{w}}\mathbf{v} = \mathbf{v} - \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{w}\|^2}\mathbf{w}$$

Both of these equations will be very useful to us. Note that if **w** is normalized, then the projection simplifies to

$$\text{proj}_{\hat{\mathbf{w}}}\mathbf{v} = (\mathbf{v} \cdot \hat{\mathbf{w}})\hat{\mathbf{w}}$$

The corresponding library implementation of dot product in $\mathbb{R}^3$ is as follows:

```
float
IvVector3::Dot( const IvVector3& other )
{
   return x*other.x + y*other.y + z*other.z;
}
```

1.2.7 Gram-Schmidt Orthogonalization

The combination of dot product and normalization allows us to define a particularly useful class of basis vectors. If a set of basis vectors β are all unit vectors and pairwise orthogonal, we call them an *orthonormal basis*. Our standard basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ is an example of an orthonormal basis.

In many cases we start with a general basis and want to generate the closest possible orthonormal basis. One example of this is when we perform operations on the set of vectors that make up an orthonormal basis. Even if the pure mathematical result should not change their length or relative orientation, due to floating point precision problems the resulting vectors may no longer be orthonormal. The process that allows us to create an orthonormal basis from a possibly non-orthonormal basis is called *Gram-Schmidt Orthogonalization*.

This works as follows. Suppose we have a set of non-orthogonal basis vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}$, and from them we want to create an orthonormal basis $\mathbf{w}_0, \ldots, \mathbf{w}_{n-1}$. We'll use the first vector from our original basis as the starting vector for our new basis so

$$\mathbf{w}_0 = \mathbf{v}_0$$

Now we want to create a vector orthogonal to $\mathbf{w}_0$, which points generally in the direction of $\mathbf{v}_1$. We can do this by computing the projection of $\mathbf{v}_1$ on $\mathbf{w}_0$, which produces the component vector of $\mathbf{v}_1$ parallel to $\mathbf{w}_0$. The remainder of $\mathbf{v}_1$ will be orthogonal to $\mathbf{w}_0$, so

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{v}_1 - \text{proj}_{\mathbf{w}_0}\mathbf{v}_1 \\ &= \mathbf{v}_1 - \frac{\mathbf{v}_1 \cdot \mathbf{w}_0}{\|\mathbf{w}_0\|}\mathbf{w}_0\end{aligned}$$

We perform the same process for $\mathbf{w}_2$: we project $\mathbf{v}_2$ on $\mathbf{w}_0$ and $\mathbf{w}_1$ to compute the parallel components and then subtract those from $\mathbf{v}_2$ to generate a vector orthogonal to both $\mathbf{w}_0$ and $\mathbf{w}_1$:

$$\begin{aligned}\mathbf{w}_2 &= \mathbf{v}_2 - \text{proj}_{\mathbf{w}_0}\mathbf{v}_2 - \text{proj}_{\mathbf{w}_1}\mathbf{v}_2 \\ &= \mathbf{v}_2 - \frac{\mathbf{v}_2 \cdot \mathbf{w}_0}{\|\mathbf{w}_0\|}\mathbf{w}_0 - \frac{\mathbf{v}_2 \cdot \mathbf{w}_1}{\|\mathbf{w}_1\|}\mathbf{w}_1\end{aligned}$$

In general we have

$$\begin{aligned}\mathbf{w}_i &= \mathbf{v}_i - \sum_{j=0}^{i-1} \text{proj}_{\mathbf{w}_j}\mathbf{v}_i \\ &= \mathbf{v}_i - \sum_{j=0}^{i-1} \frac{\mathbf{v}_i \cdot \mathbf{w}_j}{\|\mathbf{w}_j\|}\mathbf{w}_j\end{aligned}$$

Performing this for all n basis vectors will give us an orthogonal basis. To create an orthonormal basis, we can either normalize the resulting $\mathbf{w}_j$ vectors at the end or normalize as we go, the latter of which simplifies the projection calculation to $(\mathbf{v}_i \cdot \mathbf{w}_j)\,\mathbf{w}_j$.

1.2.8 Cross Product

Suppose we have two vectors **v** and **w** and want to find a new vector **u** orthogonal to both. The operation that computes this is the *cross product*, also known as the *vector product*. There are two possible choices for the direction of the vector, each the negation of the other (Figure 1.14); the one chosen is determined by the right-hand rule. Hold your right hand so that your forefinger points forward, your middle finger points out to the left, and your thumb points up. If you roughly align your forefinger with **v**, and your middle finger with **w**, then the cross product will point in the direction of your thumb (Figure 1.15). The length of the cross product is equal to the area of a parallelogram bordered by the two vectors (Figure 1.16). This can be computed using the formula

$$\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\|\|\mathbf{w}\| \sin\theta \tag{1.6}$$

where θ is the angle between **v** and **w**. Note that the cross product is not commutative, so order is important:

$$\mathbf{v} \times \mathbf{w} = -(\mathbf{w} \times \mathbf{v})$$

Also, if the two vectors are parallel, $\sin\theta = 0$, so we end up with the zero vector.

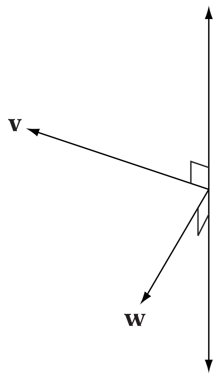


Figure 1.14 Two directions of orthogonal 3D vectors.

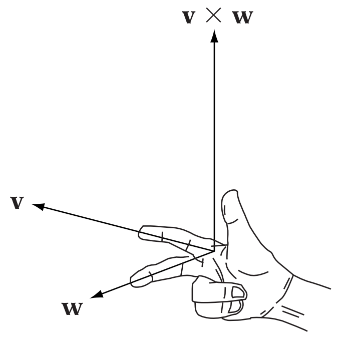


FIGURE 1.15 Cross product direction.

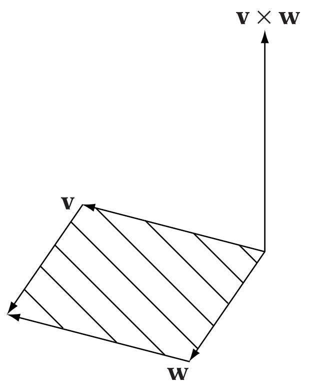


FIGURE 1.16 Cross product length equals area of parallelogram.

It is a common mistake to believe that if **v** and **w** are unit vectors, the cross product will also be a unit vector. A quick look at equation 1.6 shows this is true only if $\sin\theta$ is 1, in which case θ is 90 degrees.

The formula for the cross product is

$$\mathbf{v} \times \mathbf{w} = (v_y w_z - w_y v_z, v_z w_x - w_z v_x, v_x w_y - w_x v_y)$$

Certain processors can implement this as a two-step operation, by creating two vectors and performing the subtraction in parallel:

$$\mathbf{v} \times \mathbf{w} = (v_y w_z, v_z w_x, v_x w_y) - (w_y v_z, w_z v_x, w_x v_y)$$

For vectors **u**,**v**, **w**, and scalar a the following algebraic rules apply:

1. $\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}$
2. $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) + (\mathbf{u} \times \mathbf{w})$

3. $(\mathbf{u} + \mathbf{v}) \times \mathbf{w} = (\mathbf{u} \times \mathbf{w}) + (\mathbf{v} \times \mathbf{w})$
4. $a(\mathbf{v} \times \mathbf{w}) = (a\mathbf{v}) \times \mathbf{w} = \mathbf{v} \times (a\mathbf{w})$
5. $\mathbf{v} \times \mathbf{0} = \mathbf{0} \times \mathbf{v} = \mathbf{0}$
6. $\mathbf{v} \times \mathbf{v} = \mathbf{0}$

There are two common uses for the cross product. The first, and most used, is to generate a vector orthogonal to two others. Suppose we have three points P, Q, and R, and we want to generate a unit vector $\mathbf{n}$ that is orthogonal to the plane formed by the three points (this is known as a normal vector). Begin by computing $\mathbf{v} = (Q - P)$, and $\mathbf{w} = (R - P)$. Now we have a decision to make. Computing $\mathbf{v} \times \mathbf{w}$ and normalizing will generate a normal in one direction, whereas $\mathbf{w} \times \mathbf{v}$ and normalizing will generate one in the opposite direction (Figure 1.17). Usually we'll set things up so that the normal points from the inside toward the outside of our object.

Like the dot product, the cross product can also be used to determine if two vectors are parallel, by checking whether the resulting vector is close to the zero vector. Deciding whether to use this test as opposed to the dot product depends on what your data is. The cross product takes 9 operations. We can test for zero by examining the dot product of the result with itself $((\mathbf{v} \times \mathbf{w}) \cdot (\mathbf{v} \times \mathbf{w}))$. If it is close to 0, then we know the vectors are nearly parallel. The dot product takes an additional 5 operations, or 14 total for our test. Recall that testing for parallel vectors using the dot product of non-normalized vectors takes 18 operations; in this case the cross product test is faster.

The cross product of two vectors is defined only for vectors in $\mathbb{R}^3$. However, in $\mathbb{R}^2$ we can define a similar operation on a single vector $\mathbf{v}$, called the *perpendicular*. This is represented as $\mathbf{v}^\perp$. The result of the perpendicular is the vector rotated 90 degrees. As with the cross product, we have two choices: in this case counterclockwise or clockwise rotation. The standard definition is to rotate counterclockwise (Figure 1.18), so if $\mathbf{v} = (x, y)$, $\mathbf{v}^\perp = (-y, x)$.

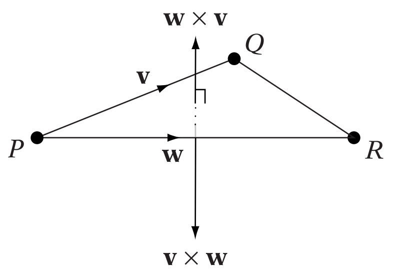


FIGURE 1.17 Computing normal for triangle.

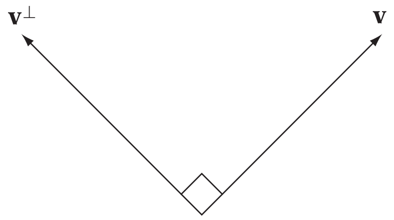


FIGURE 1.18 Perpendicular vector.

The perpendicular has similar properties to the cross product. First, it produces a vector orthogonal to the original. Also, when used in combination with the dot product in $\mathbb{R}^2$ (also known as the *perpendicular dot product*):

$$\mathbf{v}^\perp \cdot \mathbf{w} = \|\mathbf{v}\|\,\|\mathbf{w}\| \sin\theta$$

where θ is the *signed* angle between **v** and **w**. That is, if the shortest rotation to get from **v** to **w** is in a clockwise direction, then θ is negative. And similar to the cross product, the absolute value of the perpendicular dot product is equal to the area of a parallelogram bordered by the two vectors.

It is possible to take cross products in dimensions greater than 3 by using $n-1$ vectors to take an n-dimensional cross product, but in general they won't be useful to us.

Our IvVector3 cross product method is

```
IvVector3
IvVector3::Cross( const IvVector3& other )
{
    return IvVector3( y*other.z - other.y*z,
                      z*other.x - other.z*x,
                      x*other.y - other.x*y );
}
```

1.2.9 Triple Products

In $\mathbb{R}^3$ there are two extensions of the two single operation products called triple products. The first is the *vector triple product*, which returns a vector and is computed as $\mathbf{u} \times (\mathbf{v} \times \mathbf{w})$.

A special case is $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$ (Figure 1.19). Examining this, $\mathbf{v} \times \mathbf{w}$ is perpendicular both to **v** and **w**. The result of $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$ is a vector perpendicular to

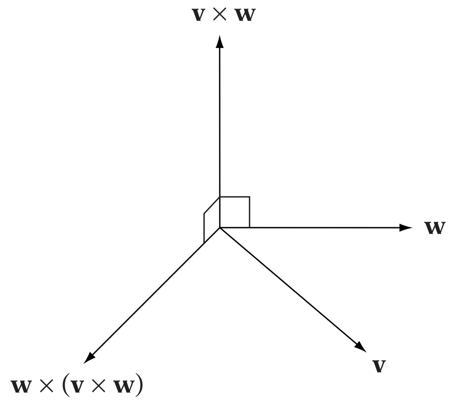

$\mathbf{v} \times \mathbf{w}$
$\mathbf{w}$
$\mathbf{v}$
$\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$

FIGURE 1.19 Using the vector triple product.

both $\mathbf{w}$ and $(\mathbf{v} \times \mathbf{w})$. Therefore, if we combine normalized versions of $\mathbf{w}$, $(\mathbf{v} \times \mathbf{w})$ and $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$, we have an orthonormal basis (all are perpendicular and of unit length).

The second triple product is called the scalar triple product. It (naturally) returns a scalar, and its formula is $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})$. To understand this geometrically, suppose we treat these three vectors as the edges of a slanted box, or parallelopiped (Figure 1.20). Then the area of the base equals $\|\mathbf{v} \times \mathbf{w}\|$, and $\|\mathbf{u}\| \cos \theta$ gives the height of the box. So

$$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \|\mathbf{u}\| \|\mathbf{v} \times \mathbf{w}\| \cos \theta$$

or area times height equals volume of the box.

In addition to computing volume, the scalar triple product can be used to test the direction of the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$, relative to

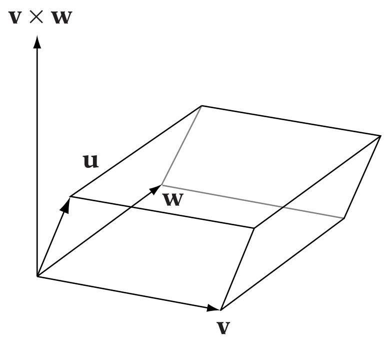


FIGURE 1.20 Scalar triple product equals volume of parallelopiped.

a third vector **u** that is linearly independent to both. If $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) > 0$, then the shortest rotation from **v** to **w** is in a counterclockwise direction (assuming our basis vectors are right-handed as we will discuss shortly) around **u**. Similarly, if $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) < 0$, the shortest rotation is in a relative clockwise direction.

For example, suppose we have a tank with current velocity **v** and desired direction **d** of travel. Our tank is oriented so that its current up direction points along a vector **u**. We take the cross product $\mathbf{v} \times \mathbf{d}$ and dot it with **u**. If the result is positive, then we know that **d** lies to the left of **v** (counterclockwise rotation) and we turn left. Similarly, if the value is less than zero, then we know we must turn right to match **d** (Figures 1.21 and 1.22).

If we know that the tank is always oriented so that it lies on the xy-plane, we can simplify this considerably. Vectors **v** and **d** will always have z values

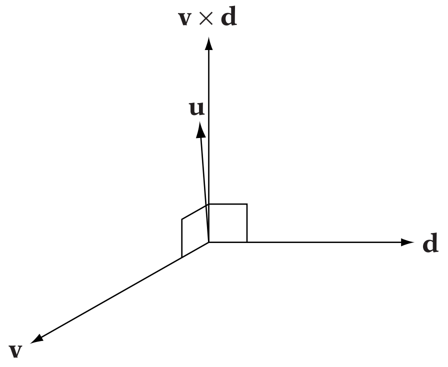


FIGURE 1.21 Scalar triple product indicates left turn.

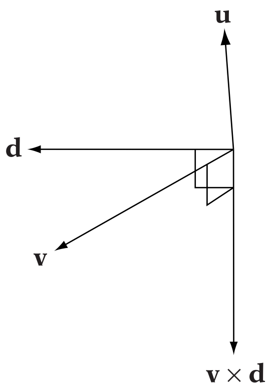


FIGURE 1.22 Scalar triple product indicates right turn.

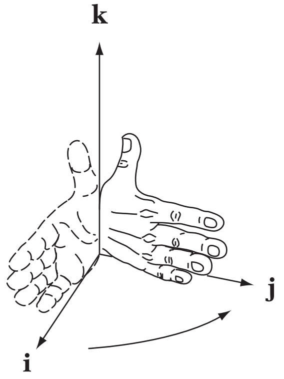


FIGURE 1.23 Right-handed rotation.

of 0, and $\mathbf{u}$ will always point in the same direction as the standard basis vector $\mathbf{k}$. In this case the result of $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{d})$ is equal to the z value of $\mathbf{v} \times \mathbf{d}$. So the problem simplifies to taking the cross product of $\mathbf{v}$ and $\mathbf{d}$ and checking the sign of the resulting z value to determine our turn direction.

Finally, we can use the scalar triple product to test whether our ordered basis vectors in $\mathbb{R}^3$ are left-handed or right-handed. We can test this informally for our standard basis by using the right-hand rule. Take your right hand and point the thumb along $\mathbf{k}$ and your fingers along $\mathbf{i}$. Now, rotating around your thumb, sweep your fingers counterclockwise into $\mathbf{j}$ (Figure1.23). This 90 degree rotation of $\mathbf{i}$ into $\mathbf{j}$ shows that the basis is right-handed. We can do the same trick with the left hand rotating clockwise to show that a basis is left-handed.

Formally, if we have three basis vectors $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$, then they are right-handed if $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) > 0$, and left-handed if $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) < 0$. If $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) = 0$, we've got a problem — our vectors are linearly dependent and thus are not a basis.

While the scalar triple product only applies to vectors in $\mathbb{R}^3$, we can use the perpendicular dot product to test vectors in $\mathbb{R}^2$ for both turning direction and right or left handedness. For example, if we have two basis vectors $\{\mathbf{v}_0, \mathbf{v}_1\}$ in $\mathbb{R}^2$, then they are right-handed if $\mathbf{v}_0^{\perp} \cdot \mathbf{v}_1 > 0$, and left-handed if $\mathbf{v}_0^{\perp} \cdot \mathbf{v}_1 < 0$.

For vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ in $\mathbb{R}^3$ the following algebraic rules regarding the triple products apply:

1. $\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \cdot \mathbf{w})\mathbf{v} - (\mathbf{u} \cdot \mathbf{v})\mathbf{w}$
2. $(\mathbf{u} \times \mathbf{v}) \times \mathbf{w} = (\mathbf{u} \cdot \mathbf{w})\mathbf{v} - (\mathbf{v} \cdot \mathbf{w})\mathbf{u}$
3. $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \mathbf{w} \cdot (\mathbf{u} \times \mathbf{v}) = \mathbf{v} \cdot (\mathbf{w} \times \mathbf{u})$

1.3 Points

Now that we have covered vectors and vector operations in some detail, we turn our attention to a related entity, the point. While the reader probably has some intuitive notion of what a point is, in this part we'll provide a mathematical representation and discuss the relationship between vectors and points. We'll also discuss some special operations that can be performed on points and alternatives to the standard Cartesian coordinate system.

Within this part it is also assumed that the reader has some general sense of what lines and planes are. More information on these topics follows in subsequent parts.

1.3.1 Points as Geometry

Everyone who has been through a first-year geometry course should be familiar with the notion of a *point*. Euclid describes the point in his work *Elements* [35] as "that which has no part." They have also been presented as the cross-section of a line, or the intersection of two lines. A less vague but still not satisfactory definition is to describe them as an infinitely small entity which has only the property of location. In games we use points for two primary purposes: to represent the position of game objects and as the basic building block of their geometric representation. Points are represented graphically by a dot.

Euclid did not present a means for representing position numerically, although later Greek mathematicians used latitude, longitude, and altitude. The primary system we use now—Cartesian coordinates—was originally published by Rene Descartes in his 1637 work *La geometrie* [26] and further revised by Newton and Leibniz.

In this system we measure a point's location relative to a special, anchored point, called the *origin*, which is represented by the letter O. In $\mathbb{R}^2$ we informally define two perpendicular real number lines or axes—known as the x- and y-axes—which pass through the origin. We indicate the location of a point P by a pair (x, y) in $\mathbb{R}^2$, where x is the distance from the point to the y-axis, and y is the distance from the point to the x-axis. Another way to think of it is that we count x units along the x-axis and then y units up parallel to the y-axis to reach the point's location. This combination of origin and axes is called the *Cartesian coordinate system* (Figure 1.24).

For $\mathbb{R}^3$ three perpendicular coordinate axes—x, y, and z—intersect at the origin. There are corresponding coordinate planes xy, yz, and xz that also intersect at the origin. Take the room you're sitting in as our space, with one corner of the room as the origin, and think of the walls and floor as the three coordinate planes (assume they extend infinitely). The edges where the walls and floor join together correspond to the axes. We can think of a three-dimensional position as being a real number triple (x, y, z)

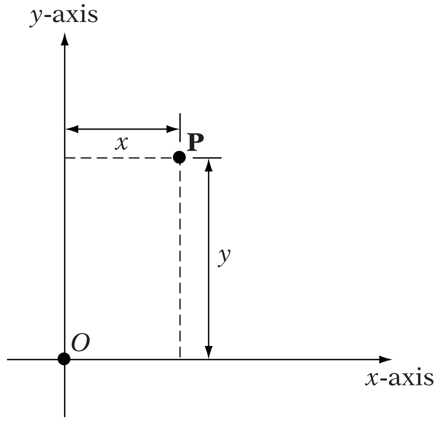


FIGURE 1.24 2D Cartesian coordinate system.

corresponding to the distance of the point to the three planes, or counting along each axis as before.

In Figure 1.25 you can see an example of a three-dimensional coordinate system. Here the axis pointing up is called the z-axis, the one to the side is

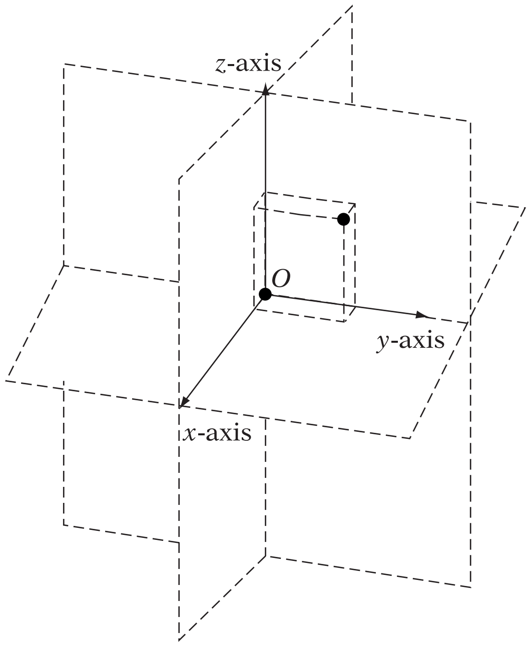


FIGURE 1.25 3D Cartesian coordinate system.

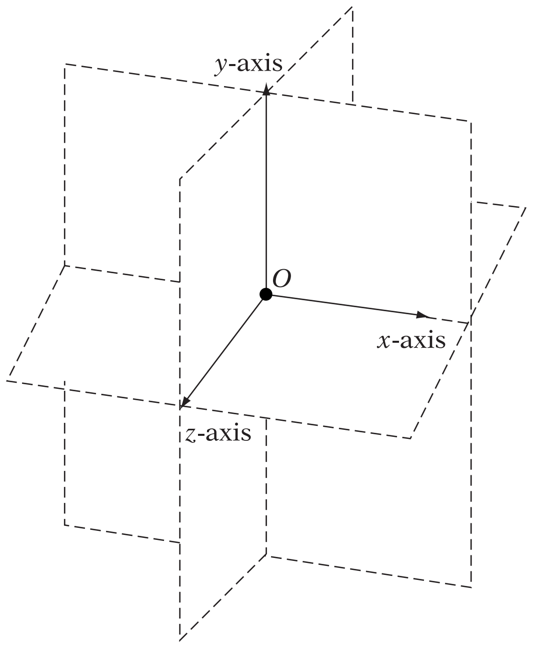


FIGURE 1.26 Alternate 3D Cartesian coordinate system.

the y-axis, and the one aimed slightly out of the page is the x-axis. Another system that is commonly used in graphics books has the y-axis pointing up, the x-axis to the right, and the z-axis out of the page (Figure 1.26). Some graphics developers favor this because the x- and y-axis match the relative axes of the two-dimensional screen, but most of the time we'll be using the former convention for this book.

Both of the three-dimensional coordinate systems we have described are right-handed. As before, we can test this via the right-hand rule. This time point your thumb along the z-axis, your fingers along the x-axis, and rotate counterclockwise into the y-axis. As with left-handed bases, we can have left-handed coordinate systems (and will be using them later in this book), but the majority of our work will be done in a right-handed coordinate system because of convention.

1.3.2 Affine Spaces

We can provide a more formal definition of coordinate systems based on what we already know of vectors and vector spaces. Before we can do so, though, we need to define the relationship between vectors and points. Points can be related to vectors by means of an *affine space*. An affine space consists of a set of points W and a vector space V. The relation between the points and vectors is defined using the following two operations:

For every pair of points P and Q in W, there is a unique vector $\mathbf{v}$ in V such that

$$\mathbf{v} = Q - P$$

Correspondingly, for every point P in W and every vector $\mathbf{v}$ in V, there is a unique point Q such that

$$Q = P + \mathbf{v} \tag{1.7}$$

This relationship can be seen in Figure 1.27. We can think of the vector $\mathbf{v}$ as acting as a displacement between the two points P and Q. To determine the displacement between two points, we subtract one from another. To displace a point, we add a vector to it and that gives us a new point.

We can define a fixed point O in W, known as the *origin*. Then using equation 1.7, we can represent any point P in W as

$$P = O + \mathbf{v}$$

or, expanding our vector using n basis vectors that span V:

$$P = O + a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1} \tag{1.8}$$

Using this, we can represent our point using an n-tuple $(a_0, \ldots, a_{n-1})$ just as we do for vectors. The combination of the origin O and our basis vectors $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$ is known as a *coordinate frame*.

Note that we can use any point in W as our origin and—for an n-dimensional affine space—any n linearly independent vectors as our basis. Unlike the Cartesian axes, this basis does not have to be orthonormal, but using an orthonormal basis (as with vectors) does make matching our

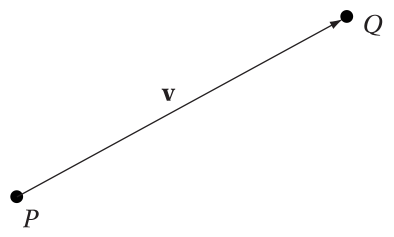


FIGURE 1.27 Affine relationship between points and vectors.

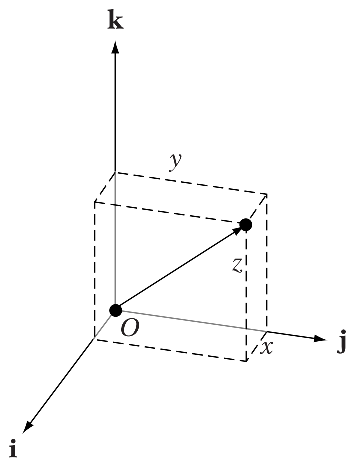


FIGURE 1.28 Relationship between points and vectors in Cartesian affine frame.

physical geometry with our abstract representation more straightforward. Because of this, we will work with the standard origin $(0, 0, \ldots, 0)$, and the standard basis $\{(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$. This is known as the *Cartesian frame*.

In $\mathbb{R}^3$ our Cartesian frame will be the origin $(0, 0, 0)$ and the standard ordered basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ as before. Our basis vectors will lie along the x-, y-, and z-axes, respectively. By using this system, we can use the same triple (x, y, z) to represent a point and the corresponding vector from the origin to the point (Figure 1.28).

To compute the distance between two points we use the length of the vector that is their difference. So if we have two points $P_0 = (x_0, y_0, z_0)$ and $P_1 = (x_1, y_1, z_1)$ in $\mathbb{R}^3$, the difference is

$$\mathbf{v} = P_1 - P_0 = (x_1 - x_0, y_1 - y_0, z_1 - z_0)$$

and the distance between them is

$$\text{dist}(P_1, P_0) = \|v\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

This is also known as the *Euclidean distance*. In the $\mathbb{R}^3$ Cartesian frame, the distance between a point $P = (x, y, z)$ and the origin is

$$\text{dist}(P, 0) = \sqrt{x^2 + y^2 + z^2}$$

1.3.3 Affine Combinations

So far the only operation that we've defined on points alone is subtraction, which results in a vector. However, there is a limited addition operation that we can perform on points that gives us a point as a result. It is known as an *affine combination*, and has the form

$$P = a_0 P_0 + a_1 P_1 + \cdots + a_k P_k \tag{1.9}$$

where

$$a_0 + a_1 + \cdots + a_k = 1 \tag{1.10}$$

So an affine combination of points is like a linear combination of vectors, with the added restriction that all the coefficients need to add up to 1. We can show why this restriction allows us to perform this operation by rewriting equation 1.10 as

$$a_0 = 1 - a_1 - \cdots - a_k$$

and substituting into equation 1.9 to get

$$\begin{aligned} P &= (1 - a_1 - \cdots - a_k) P_0 + a_1 P_1 + \cdots + a_k P_k \\ &= P_0 + a_1(P_1 - P_0) + \cdots + a_k(P_k - P_0) \end{aligned} \tag{1.11}$$

If we set $\mathbf{u}_1 = (P_1 - P_0)$, $\mathbf{u}_2 = (P_2 - P_0)$, and so on, we can rewrite this as

$$P = P_0 + a_1\mathbf{u}_1 + a_2\mathbf{u}_2 + \cdots + a_k\mathbf{u}_k$$

So by restricting our coefficients in this manner, it allows us to rewrite the affine combination as a point plus a linear combination of vectors, a perfectly legal operation.

Looking back at our coordinate frame equation 1.8, we can see that it too is an affine combination. Just as we use the coefficients in a linear combination of basis vectors to represent a general vector, we can use the coefficients of an affine combination of origin and basis vectors to represent a general point.

An affine combination spans an affine space, just as a linear combination spans a vector space. If the vectors in equation 1.11 are linearly independent, we can represent any point in the spanned affine space using the coefficients of the affine combination, just as we did before with vectors. In this case we say that the points $P_0, P_1, \ldots, P_k$ are *affinely independent*, and the ordered points are called a *simplex*. The coefficients are called *barycentric coordinates*. For example, we can create an affine combination of a simplex made of three

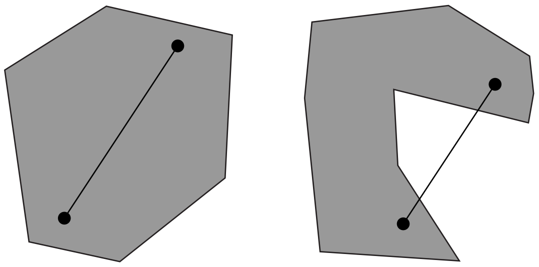


Figure 1.29 Convex versus non-convex set of points.

affinely independent points P_0, P_1, and P_2. The affine space spanned by the affine combination $a_0P_0 + a_1P_1 + a_2P_2$ is a plane, and any point in the plane can be specified by the coordinates (a_0, a_1, a_2).

We can further restrict the set of points spanned by the affine combination by considering properties of convex sets. A *convex set* of points is defined such that a line drawn between any pair of points in the set remains within the set (Figure 1.29). The *convex hull* of a set of points is the smallest convex set that includes all the points. If we restrict our coefficients $(a_0, \ldots, a_{n-1})$ such that $0 \leq a_0, \ldots, a_{n-1} \leq 1$, then we have a *convex combination*, and the span of the convex combination is the convex hull of the points. For example, the convex combination of three affinely independent points spans a triangle. We will discuss the usefulness of this in more detail when we cover triangles in Part 1.6.

If the barycentric coordinates in a convex combination of n points are all $1/n$, then the point produced is called the *centroid*, which is the mean of a set of points.

1.3.4 Point Implementation

Source Code
Library
IvMath
Filename
IvVector3

Using the Cartesian frame and standard basis in $\mathbb{R}^3$, the x, y, z values of a point P in $\mathbb{R}^3$ match the x, y, z values of the corresponding vector $P - O$, where O is the origin of the frame. This also means that we can use one class to represent both, since one can be easily converted to the other. Because of this, many math libraries don't even bother implementing a point class and just treat points as vectors.

Other libraries indicate the difference by treating them both as 4-tuples and indicate a point as $(x, y, z, 1)$ and a vector as $(x, y, z, 0)$. In this system if we subtract a point from a point, we automatically get a vector:

$$(x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1) = (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0)$$

Similarly, a point plus a vector produces a point:

$$(x_0, y_0, z_0, 1) + (x_1, y_1, z_1, 0) = (x_0 + x_1, y_0 + y_1, z_0 + z_1, 1)$$

Even affine combinations give the expected results:

$$\begin{aligned}\sum_{i=1}^{n} a_i(x_i, y_i, z_i, 1) &= \left(\sum_{i=1}^{n} a_i x_i,\ \sum_{i=1}^{n} a_i y_i,\ \sum_{i=1}^{n} a_i z_i,\ \sum_{i=1}^{n} a_i\right)\\ &= \left(\sum_{i=1}^{n} a_i x_i,\ \sum_{i=1}^{n} a_i y_i,\ \sum_{i=1}^{n} a_i z_i, 1\right)\end{aligned}$$

OpenGL uses this form when specifying the difference between a point light, which casts light rays in all directions from a given position, and a directional light, which only casts light rays in one direction. Both are specified by a single call:

```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

If the final value of `light_position` is 0, then it is treated as a directional light; otherwise, it is treated as a point light.

In our case we will not be using a separate class for points. There would be a certain amount of code duplication, since the `IvPoint3` class would end up being very similar to the `IvVector3` class. Also to be considered is the performance cost of converting points to vectors and back again. Further, to maintain type correctness we may end up distorting equations unnecessarily; this obfuscates the code and can lead to a loss in performance as well. Finally, most production game engines don't make the distinction, and we wish to remain compatible with the overall state of the industry.

Despite not making the distinction in the class structure, it is important to remember that points and vectors are not the same. One has direction and length and the other position, so not all operations apply to both. For example, we can add two vectors together to get a new vector. As we've seen, adding two points together is only allowed in certain circumstances. So while we will be using a single class, we will be maintaining mathematical correctness in the text and writing the code to reflect this.

As mentioned, most of what we need for points is already in the `IvVector3` class. The only additional code we'll have to implement is for distance and distance squared operations:

```
float
Distance( const IvVector3& point1,
          const IvVector3& point2 )
{
    float x = point1.x - point2.x;
    float y = point1.y - point2.y;
    float z = point1.z - point2.z;

    return IvSqrt( x*x + y*y + z*z );
}

float
DistanceSquared( const IvVector3& point1,
                 const IvVector3& point2 )
{
    float x = point1.x - point2.x;
    float y = point1.y - point2.y;
    float z = point1.z - point2.z;

    return ( x*x + y*y + z*z );
}
```

1.3.5 Polar and Spherical Coordinates

Cartesian coordinates are not the only way of measuring location. We've already mentioned latitude, longitude, and altitude, and there are other, related systems. Take a point P in $\mathbb{R}^2$ and compute the vector $\mathbf{v} = P - 0$. We can specify the location of P using the distance r from P to the origin—which is the length of $\mathbf{v}$—and the angle θ between $\mathbf{v}$ and the positive x-axis, where $\theta > 0$ corresponds to a counterclockwise rotation from the axis. The components (r, θ) are known as *polar coordinates*.

It is easy to convert from polar to Cartesian coordinates. We begin by forming a right triangle using the x-axis, a line from P to O, and the perpendicular from P to the x-axis (Figure 1.30). The hypotenuse has the length r and is θ degrees from the x-axis. Using simple trigonometry, the lengths of the other two sides of the triangle x and y can be computed as

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned} \tag{1.12}$$

From Cartesian to polar coordinates, we reverse the process. It's easy enough to generate r by computing the distance between P and O. Finding θ

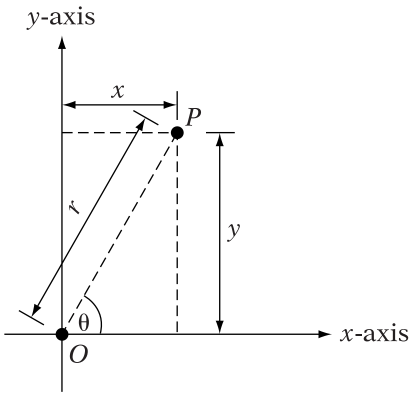


FIGURE 1.30 Relationship between polar and Cartesian coordinates.

is not as straightforward. The naive approach is to solve equation 1.12 for θ, which gives us $\theta = \arccos(x/r)$. However, the `acos()` function under C++ only returns an angle in the range of $[0, \pi)$, so we've lost the sign of the angle. Since

$$\begin{aligned}\frac{y}{x} &= \frac{r\sin\theta}{r\cos\theta}\\ &= \frac{\sin\theta}{\cos\theta}\\ &= \tan\theta\end{aligned}$$

an alternate choice would be $\arctan(y/x)$, but this doesn't handle the case when $x = 0$. To manage this, C++ provides a library function called `atan2()`, which takes y and x as separate arguments and computes $\arctan(y/x)$. It has no problems with division by 0 and maintains the signed angle with a range of $[-2\pi, 2\pi]$. We'll represent the use of this function in our equations as $\arctan 2(y, x)$. The final result is

$$\begin{aligned}r &= \sqrt{x^2 + y^2}\\ \theta &= \arctan 2(y, x)\end{aligned}$$

If r is 0, θ may be set arbitrarily.

The system that extends this to three dimensions is called *spherical coordinates*. In this system we call the distance from the point to the origin ρ instead of r. We create a sphere of radius ρ centered on the origin and define where the point lies on the sphere by two angles, ϕ and θ. If we take a vector

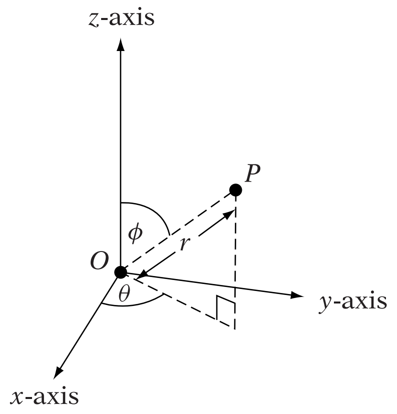


FIGURE 1.31 Spherical coordinates.

v from the origin to the point and project it down to the xy plane, θ is the angle between the x-axis and rotating counterclockwise around z. The other quantity, ϕ, measures the angle between **v** and the z-axis. The three values, ρ, ϕ, and θ, represent the location of our point (Figure 1.31).

Spherical coordinates can be converted to Cartesian coordinates as follows. Begin by building a right triangle as before, except with its hypotenuse along ρ and base along the z-axis (Figure 1.32). The length z is then $\rho \cos \phi$. To compute x and y, we project the vector **v** down onto the xy plane, and then use polar coordinates. The length r of the projected vector $\mathbf{v}'$ is $\rho \sin \phi$, so we have

$$x = \rho \sin \phi \cos \theta \tag{1.13}$$

$$y = \rho \sin \phi \sin \theta \tag{1.14}$$

$$z = \rho \cos \phi \tag{1.15}$$

To convert from Cartesian to spherical coordinates, we begin by computing ρ, which again is the distance from the point to the origin. To find ϕ, we need to find the value of $\rho \sin \phi$. This is equal to the projected xy length r since

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ &= \sqrt{(\rho \sin \phi \cos \theta)^2 + (\rho \sin \phi \sin \theta)^2} \\ &= \sqrt{(\rho \sin \phi)^2 (\cos^2 \theta + \sin^2 \theta)} \\ &= \rho \sin \phi \end{aligned}$$

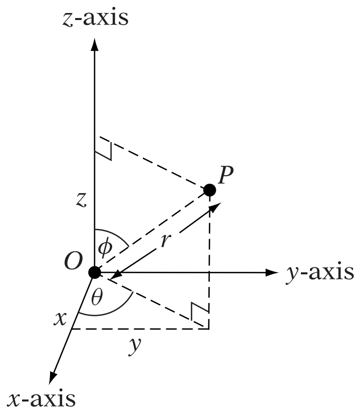


FIGURE 1.32 Relationship between spherical and Cartesian coordinates.

And since, as with polar coordinates,

$$\begin{aligned}\frac{r}{z} &= \frac{\rho \sin \phi}{\rho \cos \phi} \\ &= \tan \phi\end{aligned}$$

we can compute $\phi = \arctan 2(r, z)$. Similarly, $\theta = \arctan 2(y, x)$. Summarizing:

$$\begin{aligned}\rho &= \sqrt{x^2 + y^2 + z^2} \\ \phi &= \arctan 2\left(\sqrt{x^2 + y^2}, z\right) \\ \theta &= \arctan 2(y, x)\end{aligned}$$

1.4 Lines

1.4.1 Definition

As with the point, a *line* as a geometric concept should be familiar. Euclid defines a line as “breadthless length” and a *straight line* as that “which lies evenly with the points on itself.” A straight line has also been referred to as the shortest distance between two points, although in non-Euclidean geometry this is not necessarily true.

From first-year algebra, we know that a line in $\mathbb{R}^2$ is represented by the formula

$$y = mx + b \tag{1.16}$$

where m is the slope of the line (it describes how y changes with each step of x), and b is the coordinate location where the line crosses the y axis (called the y-intercept). In this case x varies over all values and y is represented in terms of x. This general form works for all lines in $\mathbb{R}^2$ except for those that are vertical, since in that case the slope is infinite and the y-intercept is either nonexistent or is all values along the y-axis.

Equation 1.16 has a few problems. First of all, as mentioned, we can't easily represent a vertical line—it has infinite slope. And, it isn't obvious how to transform this equation into one useful for three dimensions. We will need a different representation.

1.4.2 Parameterized Lines

One possible representation is known as a parametric equation. Instead of representing the line as a single equation with a number of variables, each coordinate value is calculated by a separate function. This allows us to use one form for a line that is generalizable across all dimensions. As an example, we will take equation 1.16 and parameterize it.

To compute the parametric equation for a line, we need two points on our line. We can take the y-intercept $(0, b)$ as one of our points, and then take one step in the positive x direction, or $(1, m+b)$, to get the other. Subtracting point 1 from point 2, we get a 2D vector $\mathbf{d} = (1, m)$, which is oriented in the same direction as the line (Figure 1.33). If we take this vector and add all the possible scalar multiples of it to the starting point $(0, b)$, then the points generated will lie along the line. We can express this in one of the following forms:

$$\begin{aligned} L(t) &= P_0 + t(P_1 - P_0) && (1.17)\\ &= (1-t)P_0 + tP_1 && (1.18)\\ &= P_0 + t\mathbf{d} && (1.19) \end{aligned}$$

The variable t in this case is called a *parameter*.

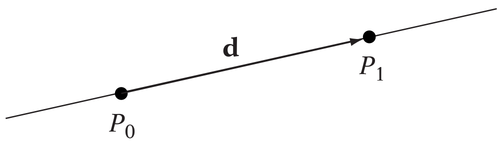


Figure 1.33 Line.

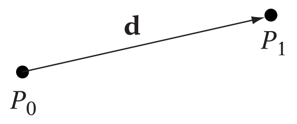


FIGURE 1.34 Line segment.

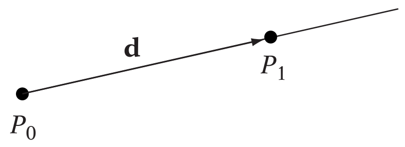


FIGURE 1.35 Ray.

We started with a two-dimensional example, but the formulas we just derived work beyond two dimensions. As long as we have two points, we can just substitute them into the preceding equations to represent a line. More formally, if we examine equation 1.17, we see it matches equation 1.11. The affine combination of two unequal or noncoincident points span a line. Equation 1.19 makes this even clearer. If we think of P_0 as our origin and **d** as a basis vector, they span a one-dimensional affine space—which is the line.

Since our line is spanned by an affine combination of our two points, the logical next question is, What is spanned by the convex combination? The convex combination requires that t and $(1 - t)$ lie between 0 and 1, which holds only if t lies in the interval [0, 1]. Clamping t to this range gives us a *line segment* (Figure 1.34). The edges of polygons are line segments, and we'll also be using line segments when we talk about bounding objects and collision detection.

If we clamp t to only one end of the range, usually specifying that $t \geq 0$, then we end up with a *ray* (Figure 1.35) that starts at P_0 and extends infinitely along the line in the direction of **d**. Rays are useful for intersection and visibility tests. For example, P_0 may represent the position of a camera, and **d** is the viewing direction.

SOURCE CODE
LIBRARY
IvMath
FILENAME
IvLine3
IvLineSegment3
IvRay3

In code we'll be representing our lines, rays, and line segments as a point on the line P and a vector **d**; so for example, the class definition for a line in $\mathbb{R}^3$ is

```
class IvLine3
{
public:
  IvLine3( const IvVector3& direction, const IvPoint3& origin );
```

```
    IvVector3 mDirection;
    IvPoint3  mOrigin;
};
```

1.4.3 Generalized Line Equation

There is another formulation of our two-dimensional line which can be useful. Let's start by writing out the equations for both x and y in terms of t:

$$x = P_x + td_x$$
$$y = P_y + td_y$$

Solving for t in terms of x:

$$t = \frac{(x - P_x)}{d_x}$$

Substituting this into the y equation we get

$$y = d_y \frac{(x - P_x)}{d_x} + P_y$$

We can rewrite this as

$$\begin{aligned} 0 &= \frac{(y - P_y)}{d_y} - \frac{(x - P_x)}{d_x} \\ &= (-d_y)x + (d_x)y + (d_y P_x - d_x P_y) \\ &= ax + by + c \end{aligned} \tag{1.20}$$

where

$$\begin{aligned} a &= -d_y \\ b &= d_x \\ c &= d_y P_x - d_x P_y = -aP_x - bP_y \end{aligned}$$

We can think of a and b as the components of a two-dimensional vector **n**, which is the perpendicular to the direction vector **d**, and so is orthogonal to the direction of the line (Figure 1.36). This gives us a way of testing where a 2D point lies relative to a 2D line. If we substitute the coordinates of the point into the x, y values of the equation, then a value of 0 indicates it's on the line,

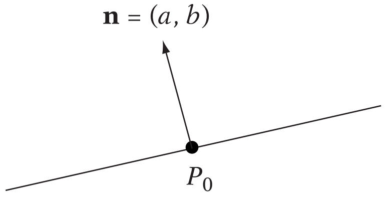


FIGURE 1.36 Normal form of 2D line.

a positive value indicates that it's on the side where the vector is pointing, and a negative value indicates that it's on the opposite side. If we normalize our vector, we can use the value returned by the line equation to indicate the distance from the point to the line.

To see why this is so, suppose we have a test point Q. We begin by constructing the vector between Q and our line point P, or $Q - P$. There are two possibilities. If Q lies on the side of the line where $\mathbf{n}$ is pointing, then the distance between Q and the line is

$$d = \|Q - P\| \cos\theta$$

where θ is the angle between $\mathbf{n}$ and $Q - P$. But since $\mathbf{n} \cdot (Q - P) = \|\mathbf{n}\|\|Q - P\| \cos\theta$, we can rewrite this as

$$d = \frac{\mathbf{n} \cdot (Q - P)}{\|\mathbf{n}\|}$$

If Q is lying on the opposite side of the line, then we take the dot product with the negative of $\mathbf{n}$, so

$$\begin{aligned} d &= \frac{-\mathbf{n} \cdot (Q - P)}{\| - \mathbf{n}\|} \\ &= -\frac{\mathbf{n} \cdot (Q - P)}{\|\mathbf{n}\|} \end{aligned}$$

Since d is always positive, we can just take the absolute value of $\mathbf{n} \cdot (Q - P)$ to get

$$d = \frac{|\mathbf{n} \cdot (Q - P)|}{\|\mathbf{n}\|} \tag{1.21}$$

If we know that $\mathbf{n}$ is normalized, we can drop the denominator. If $Q = (x, y)$ and (as we've stated) $\mathbf{n} = (a, b)$, we can expand our values to get

$$\begin{aligned} d &= a(x - P_x) + b(y - P_y) \\ &= ax + by - aP_x - bP_y \\ &= ax + by + c \end{aligned}$$

If our $\mathbf{n}$ is not normalized, then we need to remember to divide by $\|\mathbf{n}\|$ to get the correct distance.

1.4.4 Collinear Points

Three or more points are said to be *collinear* if they all lie on a line. Another way to think of this is that despite there being more than two points, the affine space that they span is only one-dimensional.

To determine whether three points P_0, P_1, and P_2 are collinear, we take the cross product of $P_1 - P_0$ and $P_2 - P_0$ and test whether the result is close to the zero vector. This is equivalent to testing whether basis vectors for the affine space are parallel.

1.5 Planes

Euclid defines a surface as "that which has length and breadth only" and a plane surface, or just a *plane*, as "a surface which lies evenly with the straight lines on itself." Another way of thinking of this is that a plane is created by taking a straight line and sweeping each point on it along a second straight line. It is a flat, limitless, infinitely thin surface.

1.5.1 Parameterized Planes

As with lines, we can express a plane algebraically in a number of ways. The first follows from our parameterized line. From basic geometry we know that two noncoincident points form a line and three noncollinear points form a plane. So if we can parameterize a line as an affine combination of two points, then it makes sense that we can parameterize a plane as an affine combination of three points P_0, P_1, and P_2, or

$$P(s, t) = (1 - s - t)P_0 + sP_1 + tP_2$$

Alternatively, we can represent this as an origin point plus the linear combination of two vectors:

$$\begin{aligned} P(s,t) &= P_0 + s(P_1 - P_0) + t(P_2 - P_0) \\ &= P_0 + s\mathbf{u} + t\mathbf{v} \end{aligned}$$

As with the parameterized line equation, if our points are of higher dimension, we can create planes in higher dimensions from them. However, in most cases our planes will be firmly entrenched in $\mathbb{R}^3$.

1.5.2 Generalized Plane Equation

We can define an alternate representation for a plane in $\mathbb{R}^3$, just as we did for a line in $\mathbb{R}^2$. In this form a plane is defined as the set of points perpendicular to a normal vector $\mathbf{n} = (a, b, c)$ which also contains the point $P_0 = (x_0, y_0, z_0)$ as shown in Figure 1.37. If a point P lies on the plane, then the vector $\mathbf{v} = P - P_0$ also lies on the plane. For $\mathbf{v}$ and $\mathbf{n}$ to be orthogonal, then $\mathbf{n} \cdot \mathbf{v} = 0$. Expanding this gives us the *normal-point* form of the plane equation, or

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

We can pull all the constants into one term to get

$$\begin{aligned} 0 &= ax + by + cz - (ax_0 + by_0 + cz_0) \\ &= ax + by + cz + d \end{aligned}$$

So extending equation 1.20 to three dimensions gives us the equation for a plane in $\mathbb{R}^3$.

This is the *generalized plane equation*. As with the generalized line equation, this equation can be used to test where a point lies relative to either side of a plane. Again, comparable to the line equation, it can be proved that if $\mathbf{n}$ is normalized, $|ax + by + cz + d|$ returns the distance from the point to the plane.

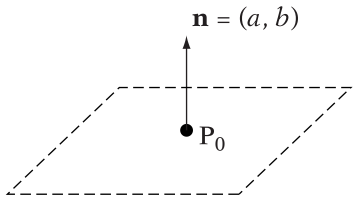


Figure 1.37 Normal form of plane.

Testing points versus planes using the general plane equation happens quite often. For example, to detect whether a point lies inside a convex polyhedron, you can do a plane test for every face of the polyhedron. Assuming the plane normals point away from the center of the polyhedron, if the point is on the negative side of all the planes then it lies inside. We may also use planes as culling devices that cut our world into half-spaces. If an object lies on one half of a plane, we consider it (say, for rendering purposes); otherwise, we ignore it. The distance property can be used to test whether a sphere is intersecting a plane. If the distance between the sphere's center and the plane is less than the sphere's radius, then the sphere is intersecting the plane.

Given three points in $\mathbb{R}^3$ P, Q, and R, we generate the generalized plane equation as follows. First we compute two vectors **u** and **v**, where

$$\mathbf{u} = Q - P$$
$$\mathbf{v} = R - P$$

Now we take the cross product of these two vectors to get the normal to the plane:

$$\mathbf{n} = \mathbf{u} \times \mathbf{v}$$

We usually normalize **n** at this point so that we can take advantage of the distance measuring properties of the plane equation. This gives us our values a, b, and c. Taking P as the point on the plane, we compute D by

$$d = -(aP_x + bP_y + cP_z)$$

We can also use this to convert our parameterized form to the generalized form by starting with the cross product step.

Source Code
Library
IvMath
Filename
IvPlane

Since we'll be working in $\mathbb{R}^3$ most of the time and because of its useful properties, we'll be using the generalized plane equation as the basis for our class:

```
class IvPlane
{
public:
  IvPlane( float a, float b, float c, float d );

  IvVector3 mNormal;
  float    D;
};
```

However, from time to time we'll be making use of the parameterized form, so it's good to keep it in mind.

1.5.3 Coplanar Points

Four or more points are said to be *coplanar* if they all lie on a plane. Another way to think of this is that despite the number of points being greater than three, the affine space that they span is only two-dimensional.

To determine whether four points P_0, P_1, P_2, and P_3 are coplanar, we create vectors $P_1 - P_0$, $P_2 - P_0$, and $P_3 - P_0$, and compute their triple scalar product. If the result is near zero, then they may be coplanar, if they're not collinear. To determine if they are collinear, take the cross products $(P_1 - P_0) \times (P_2 - P_0)$, and $(P_1 - P_0) \times (P_3 - P_0)$. If both results are near zero, then the points are collinear instead.

1.6 Polygons and Triangles

Source Code
Library
IvMath
Filename
IvTriangle

The current class of graphics processors wants their geometric data in primarily one form: points. However, having just a collection of points is not enough. We need to organize these points into smaller groups, for both rendering and computational purposes.

A *polygon* is made up of a set of *vertices* (which are represented by points) and *edges* (which are represented by line segments). The edges define how the vertices are connected together. A *convex polygon* is one where the set of points enclosed by the vertices and edges is a convex set; otherwise, it's a *concave polygon*.

The most commonly used polygons for storing geometric data are *triangles* (three vertices) and *quadrilaterals* (four vertices). While some rendering systems accept quadrilaterals (also referred to as just quads) as data, most want geometry grouped in triangles, so we'll follow that convention throughout the remainder of the book. One advantage triangles have over quadrilaterals is that three noncollinear vertices are guaranteed to be coplanar, so they can be used to define a single plane. If the three vertices of a triangle are collinear, then we have a *degenerate* triangle. Degenerate triangles can cause problems on some hardware and with some geometric algorithms, so it's good to cull them by checking for collinearity of the triangle vertices, by using the technique described previously.

If the points are not collinear, then as we've stated, the three vertices P_0, P_1, and P_2 can be used to find the triangle's incident plane. If we set $\mathbf{u} = P_1 - P_0$ and $\mathbf{v} = P_2 - P_0$, then we can define this via the parameterized plane equation $P(s, t) = P_0 + s\mathbf{u} + t\mathbf{v}$. Alternately, we can compute the generalized plane equation by computing the cross product of $\mathbf{u}$ and $\mathbf{v}$, normalizing to get the normal $\hat{\mathbf{n}}$, and then computing d as described in Section 1.5.2.

It's often necessary to test whether a 3D point lying on the triangle plane is inside or outside of the triangle itself (Figure 1.38). We begin by computing

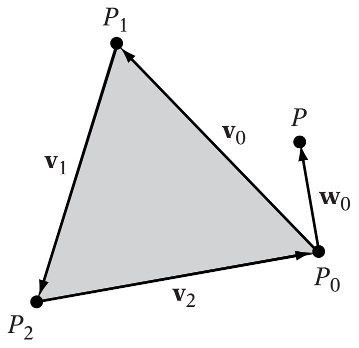


Figure 1.38 Point in triangle test.

three vectors $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$, where

$$\begin{aligned}\mathbf{v}_0 &= P_1 - P_0\\ \mathbf{v}_1 &= P_2 - P_1\\ \mathbf{v}_2 &= P_0 - P_2\end{aligned}$$

We take the cross product of $\mathbf{v}_0$ and $\mathbf{v}_1$ to get a normal vector $\mathbf{n}$ to the triangle. We then compute three vectors from each vertex to the test point:

$$\begin{aligned}\mathbf{w}_0 &= P - P_0\\ \mathbf{w}_1 &= P - P_1\\ \mathbf{w}_2 &= P - P_2\end{aligned}$$

If the point lies inside the triangle, then the cross product of each $\mathbf{v}_i$ with each $\mathbf{w}_i$ will point in the same direction as $\mathbf{n}$, which we can test by using a dot product. If the result is negative, then we know they're pointing in opposite directions, and the point lies outside. For example, in Figure 1.38, the normal vector to the triangle, computed as $\mathbf{v}_0 \times \mathbf{v}_1$, points out of the page. But the cross product $\mathbf{v}_0 \times \mathbf{w}_0$ points into the page, so the point lies outside.

We can speed up this operation by projecting the point and triangle to one of the xy, xz, or yz planes and treating it as a 2D problem. To improve our accuracy, we'll choose the one which provides the maximum area for the projection of the triangle. If we look at the normal $\mathbf{n}$ for the triangle, one of the coordinate values (x, y, z) will have the maximum absolute value; that is, the normal is pointing generally along that axis. If we drop that coordinate and keep the other two, that will give us the maximum projected area. We can then throw out a number of zero terms and end up with a considerably faster test.

This is equivalent to using the perpendicular dot product instead of the cross product. More detail on this technique can be found in Section 11.3.5.

Another advantage that triangles have over quads is that (again, assuming the vertices aren't collinear) they are convex polygons. In particular, the convex combination of the three triangle vertices spans all the points that make up the triangle. Given a point P inside the triangle and on the triangle plane, it is possible to compute its particular barycentric coordinates (s, t), as used in the parameterized plane equation $P(s,t) = P_0 + s\mathbf{u} + t\mathbf{v}$. If we compute a vector $\mathbf{w} = P - P_0$, then we can rewrite the plane equation as

$$\begin{aligned} P &= P_0 + s\mathbf{u} + t\mathbf{v} \\ \mathbf{w} &= s\mathbf{u} + t\mathbf{v} \end{aligned}$$

If we take the cross product of $\mathbf{v}$ with $\mathbf{w}$, we get

$$\begin{aligned} \mathbf{v} \times \mathbf{w} &= \mathbf{v} \times (s\mathbf{u} + t\mathbf{v}) \\ &= s(\mathbf{v} \times \mathbf{u}) + t(\mathbf{v} \times \mathbf{v}) \\ &= s(\mathbf{v} \times \mathbf{u}) \end{aligned}$$

Taking the length of both sides gives

$$\|\mathbf{v} \times \mathbf{w}\| = |s|\|\mathbf{v} \times \mathbf{u}\|$$

The quantity $\|\mathbf{v} \times \mathbf{u}\| = \|\mathbf{u} \times \mathbf{v}\|$. And since P is inside the triangle, we know that to meet the requirements of a convex combination $s \geq 0$, so

$$s = \frac{\|\mathbf{v} \times \mathbf{w}\|}{\|\mathbf{u} \times \mathbf{v}\|}$$

A similar construction finds that

$$t = \frac{\|\mathbf{u} \times \mathbf{w}\|}{\|\mathbf{u} \times \mathbf{v}\|}$$

Note that this is equivalent to computing the areas a and b of the two subtriangles shown in Figure 1.39 and dividing by the total area of the triangle c, so

$$\begin{aligned} s &= b/c \\ t &= a/c \end{aligned}$$

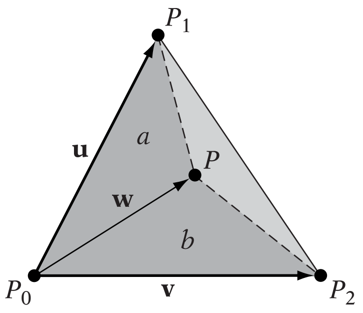


FIGURE 1.39 Computing barycentric coordinates for point in triangle.

where

$$
\begin{aligned}
a &= \frac{1}{2}\|\mathbf{u} \times \mathbf{w}\| \\
b &= \frac{1}{2}\|\mathbf{v} \times \mathbf{w}\| \\
c &= \frac{1}{2}\|\mathbf{u} \times \mathbf{v}\|
\end{aligned}
$$

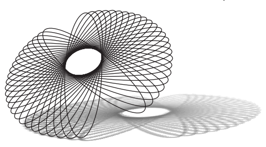
These simple examples are only a taste of how we can use triangles in mathematical calculations. More details on the use and implementation of triangles can be found throughout the text, particularly in Chapters 6 and 11.

1.7 Chapter Summary

In this chapter, we have covered some basic geometric entities: vectors and points. We have discussed linear and affine spaces, the relationships between them, and how we can use affine combinations of vectors and points to define other entities like lines and planes. We've also shown how we can use our knowledge of affine spaces and vector properties to compute some simple tests on triangles. These skills will prove useful to us throughout the remainder of the text.

For those who are interested in reading further, Anton and Rorres [3] is a standard reference for many first courses in linear algebra. Other texts with slightly different approaches are Axler [7] and Friedberg [37]. Information on points and affine spaces can be found in Schneider and Eberly [96], as well as in deRose [25].

Chapter 2



Linear Transformations and Matrices

2.1 Introduction

In the previous chapter we discussed vectors and points and some simple operations we can apply to them. Now we'll begin to expand our discussion to cover specific functions that we can apply to vectors and points; functions known as transformations. In this chapter we'll begin with a class of transformations that we can apply to vectors called linear transformations. These encompass nearly all of the common operations we might want to perform on vectors and points, so understanding what they are and how to apply them is important. We'll define these functions and how they are distinguished from other, more general transformations.

Properties of linear transformations allow us to use a structure called a matrix as a compact representation for transforming vectors. A matrix is a simple 2D array of values, but within it lies all the power of a linear transformation. Through simple operations we can use the matrix to apply linear transformations to vectors. We can also combine two transformation matrices to create a new one that has the same effect of the first two. Using matrices effectively lies at the heart of the pipeline for manipulating virtual objects and rendering them on the screen.

Matrices have other applications as well. Examining the structure of a matrix can tell us something about the transformation it represents; for example, whether it can be reversed, what that reverse transformation might be, or whether it distorts the data that it is given. Matrices can also be used to solve systems of linear equations, which is useful to know for certain algorithms in graphics and physical simulation. For all of these reasons, matrices are primary data structures in graphics application programmer interfaces (APIs).

2.2 Linear Transformations

Linear transformations are a very useful and important concept in linear algebra. As one of a class of functions known as transformations, they map vector spaces to vector spaces. This allows us to apply complex functions to, or *transform*, vectors. Linear transformations perform this mapping while also having the additional property of preserving linear combinations. We will see how this permits us to describe the linear transformation in terms of how it affects the basis vectors of a vector space. Later parts will show how this in turn allows us to represent linear transformations using matrices.

2.2.1 Definitions

Before we can begin to discuss transformations and linear transformations in particular, we need to define a few terms. A *relation* maps a set X of values (known as the *domain*) to another set Y of values (known as the *range*). A *function* is a relation where every value in the first set maps to one and only one value in the second set, for example $f(x) = \sin x$. An example of a relation that is not a function is $\pm\sqrt{x}$, because there are two possible results for a positive value of x, either positive or negative.

A function whose domain is an n-dimensional space and whose range is an m-dimensional space is known as a *transformation*. A transformation that maps from $\mathbb{R}^n$ to $\mathbb{R}^m$ is expressed as $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$. If the domain and the range of a transformation are equal (i.e., $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$), then the transformation is sometimes called an *operator*.

An example of a transformation is the function

$$f(x, y) = x^2 + 2y$$

which maps from $\mathbb{R}^2$ to $\mathbb{R}$. Another example is

$$f(x, y, z) = x^2 + 2y + \sqrt{z}$$

which maps from $\mathbb{R}^3$ to $\mathbb{R}$.

We can also map to a multidimensional space. For example, we could define a transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$ as follows:

$$\mathcal{T}(a, b) = (f(a, b), g(a, b)) \tag{2.1}$$

A *linear transformation* $\mathcal{T}$ is a mapping between two vector spaces V and W, where for all $\mathbf{v}$ in V and for all scalars a:

1. $\mathcal{T}(\mathbf{v}_0 + \mathbf{v}_1) = \mathcal{T}(\mathbf{v}_0) + \mathcal{T}(\mathbf{v}_1)$ for all $\mathbf{v}_0, \mathbf{v}_1$ in V
2. $\mathcal{T}(a\mathbf{v}) = aT(\mathbf{v})$ for all $\mathbf{v}$ in V

To determine whether a transformation is linear, it is sufficient to show that

$$\mathcal{T}(a\mathbf{x} + \mathbf{y}) = a\mathcal{T}(\mathbf{x}) + \mathcal{T}(\mathbf{y})$$

An example of a linear transformation is $\mathcal{T}(\mathbf{x}) = k\mathbf{x}$, where k is any fixed scalar. We can show this by

$$\begin{aligned} \mathcal{T}(a\mathbf{x} + \mathbf{y}) &= k(a\mathbf{x} + \mathbf{y}) \\ &= ak\mathbf{x} + k\mathbf{y} \\ &= a\mathcal{T}(\mathbf{x}) + \mathcal{T}(\mathbf{y}) \end{aligned}$$

On the other hand, the function $g(x) = x^2$ is not linear because, for $a = 2$, $x = 1$, and $y = 1$:

$$\begin{aligned} g(2(1) + 1) &= (2(1) + 1)^2 \\ &= 3^2 = 9 \\ &\neq 2(g(1)) + g(1) \\ &= 2(1^2) + 1^2 = 3 \end{aligned}$$

As we might expect, the only operations possible in a linear function are multiplication by a constant and addition.

2.2.2 Null Space and Range

We define the *null space* (or *kernel*) $N(\mathcal{T})$ of a linear transformation $\mathcal{T} : V \to W$ as the set of all vectors in V that map to $\mathbf{0}$, or

$$N(\mathcal{T}) = \{\mathbf{x} \mid \mathcal{T}(\mathbf{x}) = \mathbf{0}\}$$

The dimension of $N(\mathcal{T})$ is called the *nullity* of the transformation.

We define the *range* $R(\mathcal{T})$ of a linear transformation $\mathcal{T} : V \rightarrow W$ as the set of all vectors in W that are mapped to by at least one vector in V, or

$$R(\mathcal{T}) = \{\mathcal{T}(\mathbf{x}) | \mathbf{x} \in V\}$$

The dimension of $R(\mathcal{T})$ is called the *rank* of the transformation.

The null space and range have two important properties. First of all, they are both vector spaces, and in fact the null space is a subspace of V and the range is a subspace of W. Second,

$$nullity(\mathcal{T}) + rank(\mathcal{T}) = dim(\mathcal{T})$$

To get a better sense of this, let's look at an example. Suppose we have the linear transformation

$$\mathcal{T}(a, b) = (a + b, 0)$$

The resulting range space is of the form $(x, 0)$, so it can be spanned by the vector $(1, 0)$ and has dimension 1. The transformation will produce the vector $(0, 0)$ only when $a = -b$. So the null space has a basis of $(1, -1)$ and is also one-dimensional. As we expect, they add up to 2, the dimension of our original vector space (Figure 2.1).

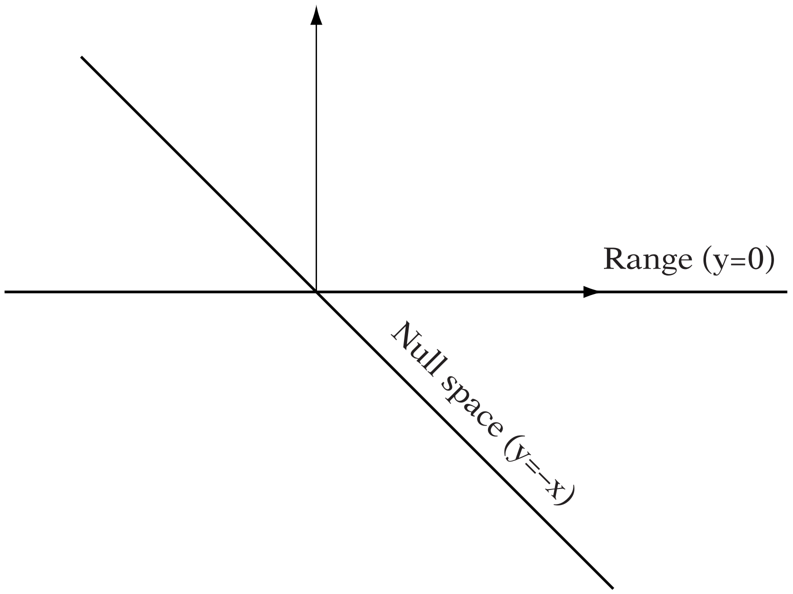


FIGURE 2.1 Range and null space for transformation T(a,b) = (a+b, 0).

2.2.3 Linear Transformations and Basis Vectors

Using standard function notation to represent linear transformations (as in equation 2.1) is not the most convenient nor compact format, particularly for transformations between higher-dimensional vector spaces. Fortunately, using the properties of vectors will allow us to define something more useful to us.

Recall that we can represent any vector $\mathbf{x}$ in an n-dimensional vector space V as

$$\mathbf{x} = x_0\mathbf{v}_0 + x_1\mathbf{v}_1 + \cdots + x_{n-1}\mathbf{v}_{n-1}$$

where $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}$ is a basis for V.

Now suppose we have a linear transformation $\mathcal{T} : V \to W$ that maps from V to an m-dimensional vector space W. If we apply our transformation to our arbitrary vector $\mathbf{x}$, then we have

$$\begin{aligned}\mathcal{T}(\mathbf{x}) &= \mathcal{T}(x_0\mathbf{v}_0 + x_1\mathbf{v}_1 + \cdots + x_{n-1}\mathbf{v}_{n-1})\\ &= x_0\mathcal{T}(\mathbf{v}_0) + x_1\mathcal{T}(\mathbf{v}_1) + \cdots + x_{n-1}\mathcal{T}(\mathbf{v}_{n-1}) \qquad (2.2)\end{aligned}$$

So if we know how our linear transformation affects our basis for V, then we can calculate the effect of the linear transformation for any arbitrary vector in V.

There is still an open question: What are the components of each $\mathcal{T}(\mathbf{v}_d)$ equal to? For a member $\mathbf{v}_d$ of V's basis, we can represent $\mathcal{T}(\mathbf{v}_d)$ in terms of the basis $\{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}\}$ for W, again as a linear combination:

$$\mathcal{T}(\mathbf{v}_j) = a_{0,j}\mathbf{w}_0 + a_{1,j}\mathbf{w}_1 + \cdots + a_{m-1,j}\mathbf{w}_{m-1}$$

If $\{\mathbf{w}_0, \ldots, \mathbf{w}_{m-1}\}$ is the standard basis for W, this simplifies to

$$\mathcal{T}(\mathbf{v}_j) = (a_{0,j}, a_{1,j}, \ldots, a_{m-1,j}) \qquad (2.3)$$

Combining equations 2.2 and 2.3 gives us

$$\begin{aligned}\mathcal{T}(\mathbf{x}) = x_0&(a_{0,0}, a_{1,0}, \ldots, a_{m-1,0})\\ &+ x_1(a_{0,1}, a_{1,1}, \ldots, a_{m-1,1})\\ &\cdots\\ &+ x_{n-1}(a_{0,n-1}, a_{1,n-1}, \ldots, a_{m-1,n-1}) \qquad (2.4)\end{aligned}$$

If we set $\mathbf{b} = \mathcal{T}(\mathbf{x})$, then for a given component of $\mathbf{b}$

$$b_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1} \tag{2.5}$$

Knowing this, we can precalculate and store the n transformed basis vectors $(a_{0,j}, a_{1,j}, \ldots, a_{m-1,j})$ and use this formula at any time to transform a general vector $\mathbf{x}$.

Let's look at an example. Taking a transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$, using the standard basis for both vector spaces:

$$\mathcal{T}(a, b) = (a + b, b)$$

If we look at how this affects our standard basis for $\mathbb{R}^2$, we get

$$\begin{aligned}\mathcal{T}(1, 0) &= (1 + 0, 0) = (1, 0)\\ \mathcal{T}(0, 1) &= (0 + 1, 1) = (1, 1)\end{aligned}$$

Transforming an arbitrary vector in $\mathbb{R}^2$, say $(2, 3)$, we get

$$\begin{aligned}\mathcal{T}(2, 3) &= 2\mathcal{T}(1, 0) + 3\mathcal{T}(0, 1)\\ &= 2(1, 0) + 3(1, 1)\\ &= (5, 3)\end{aligned}$$

which is what we expect.

It should be made clear that applying a linear transformation to a basis does not produce the basis for the new vector space. It only shows where the basis vectors end up in the new vector space—in our case in terms of the standard basis. In fact, a transformed basis may no longer be linearly independent. Take as another example

$$\mathcal{T}(a, b) = (a + b, 0)$$

Applying this to our standard basis for $\mathbb{R}^2$, we get

$$\begin{aligned}\mathcal{T}(1, 0) &= (1 + 0, 0) = (1, 0)\\ \mathcal{T}(0, 1) &= (0 + 1, 0) = (1, 0)\end{aligned}$$

The two resulting vectors are clearly linearly dependent.

These two examples illustrate one useful property. If the rank of a linear transformation $\mathcal{T}$ equals the number of elements in a transformed basis β, then we can say that β is linearly independent. In fact, the rank is equal to the number of linearly independent elements in β, and those linearly independent elements will span the range of $\mathcal{T}$.

In summary, knowing that we can represent a linear transformation in terms of how the basis vectors are transformed is a very powerful tool. As we will see, it is precisely this property of linear transformations that allows us to represent them concisely by using a matrix.

2.3 Matrices

2.3.1 Introduction to Matrices

A matrix is a rectangular, two-dimensional array of values. Throughout this book, most of the values we use will be real numbers, but they could be complex numbers or even vectors. Each individual value in a matrix is called an *element*. Examples of matrices are

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 35 & -15 \\ 2 & 52 & 1 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 2 & -1 \\ 0 & 2 \\ 6 & 3 \end{bmatrix}$$

A matrix is described as having m rows by n columns, or being an $m \times n$ matrix. A row is a horizontal group of elements from left to right, while a column is a vertical, top-to-bottom group. Matrix **A** in our example has 3 rows and 3 columns and is a 3×3 matrix, whereas matrix **C** is a 3×2 matrix. Rows are numbered 0 to $m-1$,[1] while columns are numbered 0 to $n-1$. An individual element of a matrix **A** is referenced as either $(\mathbf{A})_{i,j}$ or just $a_{i,j}$, where i is the row number and j is the column. Looking at matrix **B**, element b_{10} contains the value 2 and element b_{01} equals 35.

If an individual matrix has an equal number of rows and columns, that is if m equals n, then it is called a *square matrix*. Matrix **A** is square, whereas matrices **B** and **C** are not.

If all elements of a matrix are zero, then it is called a *zero matrix*. We will represent a matrix of this type as **0** and assume a matrix of the appropriate size for the operation we are performing.

If two matrices have an equal number of rows and columns, then they are said to be the same *size*. If they are the same size and their corresponding

1. As a reminder, mathematical convention starts with 1, but we're using 0 to be compatible with C++.

elements have the same values, then they are *equal*. Below, the two matrices are the same size, but they are not equal.

$$\begin{bmatrix} 0 & 1 \\ 3 & 2 \\ 0 & -3 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 2 & -3 \\ 1 & 3 \end{bmatrix}$$

The set of elements where row and column number are the same is called the *main diagonal*. In the next example the main diagonal is in bold.

$$\mathbf{U} = \begin{bmatrix} \mathbf{3} & -5 & 0 & 1 \\ 0 & \mathbf{2} & 6 & 0 \\ 0 & 0 & \mathbf{1} & -8 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix}$$

The *trace* of a matrix is the sum of the main diagonal elements. In this case the trace is $3 + 2 + 1 + 1 = 7$.

In matrix **U**, all elements below the diagonal are equal to 0. This is known as an *upper triangular* matrix. Note that elements above the diagonal don't necessarily have to be nonzero in order for the matrix to be upper triangular, nor does the matrix have to be square.

If elements above the diagonal are 0, then we have a *lower triangular* matrix

$$\mathbf{L} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ -6 & 1 & 0 & 1 \end{bmatrix}$$

Finally, if a square matrix has nondiagonal elements of zero, we call the matrix a *diagonal* matrix:

$$\mathbf{D} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It follows that any diagonal matrix is both an upper triangular and lower triangular matrix.

2.3.2 Simple Operations

Matrix Addition and Scalar Multiplication

We can add and scale matrices just as we can vectors. Adding two matrices together:

$$\mathbf{S} = \mathbf{A} + \mathbf{B}$$

is done componentwise like vectors, thus

$$s_{i,j} = a_{i,j} + b_{i,j}$$

Clearly, in order for this to work, **A**, **B**, and **S** must all be the same size (also known as *conformable for addition*). Subtraction works similarly but as with real numbers and vectors is not commutative.

To scale a matrix,

$$\mathbf{P} = s\mathbf{A}$$

each element is multiplied by the scalar, again like vectors:

$$p_{i,j} = s \cdot a_{i,j}$$

Matrix addition and scalar multiplication have their algebraic rules, which should seem quite familiar at this point:

1. $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$
2. $\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$
3. $\mathbf{A} + \mathbf{0} = \mathbf{A}$
4. $\mathbf{A} + (-\mathbf{A}) = \mathbf{0}$
5. $a(\mathbf{A} + \mathbf{B}) = a\mathbf{A} + a\mathbf{B}$
6. $a(b\mathbf{A}) = (ab)\mathbf{A}$
7. $(a + b)\mathbf{A} = a\mathbf{A} + b\mathbf{A}$
8. $1\mathbf{A} = \mathbf{A}$

As we can see, these rules match the requirements for a vector space, and so the set of matrices of a given size is also a vector space.

Transpose

The *transpose* of a matrix $\mathbf{A}$ (represented by $\mathbf{A}^T$) interchanges the rows and columns of $\mathbf{A}$. It does this by exchanging elements across the matrix's main diagonal, so $(\mathbf{A}^T)_{i,j} = (\mathbf{A})_{j,i}$. An example of this is

$$\begin{bmatrix} 2 & -1 \\ 0 & 2 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 6 \\ -1 & 2 & 3 \end{bmatrix}$$

As we can see, the matrix does not have to be square, so an $m \times n$ matrix becomes an $n \times m$ matrix. Also, the main diagonal doesn't change, or is invariant, since $(\mathbf{A}^T)_{i,i} = (\mathbf{A})_{i,i}$.

A matrix where $(\mathbf{A})_{i,j} = (\mathbf{A})_{j,i}$ (i.e., cross-diagonal entries are equal) is called a *symmetric matrix*. All diagonal matrices are symmetric. Another example of a symmetric matrix is

$$\begin{bmatrix} 3 & 1 & 2 & 3 \\ 1 & 2 & -5 & 0 \\ 2 & -5 & 1 & -9 \\ 3 & 0 & -9 & 1 \end{bmatrix}$$

The transpose of a symmetric matrix is the matrix again, since in this case $(\mathbf{A}^T)_{j,i} = (\mathbf{A})_{i,j} = (\mathbf{A})_{j,i}$.

A matrix where $(\mathbf{A})_{i,j} = -(\mathbf{A})_{j,i}$ (i.e., cross-diagonal entries are negated and the diagonal is 0) is called a *skew symmetric matrix*. An example of a skew symmetric matrix is

$$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & -5 \\ -2 & 5 & 0 \end{bmatrix}$$

The transpose of a skew symmetric matrix is the negation of the original matrix, since in this case $(\mathbf{A}^T)_{j,i} = (\mathbf{A})_{i,j} = -(\mathbf{A})_{j,i}$.

Some useful algebraic rules involving the transpose are

1. $(\mathbf{A}^T)^T = \mathbf{A}$
2. $(a\mathbf{A}^T) = a\mathbf{A}^T$
3. $(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$

where a is a scalar and $\mathbf{A}$ and $\mathbf{B}$ are conformable for addition.

2.3.3 Vector Representation

If a matrix has only one row or one column, then we have a row or column matrix, respectively:

$$\begin{bmatrix} .5 & .25 & 1 & -1 \end{bmatrix} \quad \begin{bmatrix} 5 \\ -3 \\ 6.9 \end{bmatrix}$$

These are often used to represent vectors. While there is no particular standard as to which one to use, in this text we will assume that vectors are represented as column matrices (also known as column vectors). First of all, most math texts use column vectors and we wish to remain compatible. In addition, we want to ensure that any matrix we may reference will be usable by our graphics pipeline. We'll be doing some derivations based on the OpenGL specification and its documentation uses column vectors. DirectX, by comparison, uses row vectors. Finally, the classical presentation of quaternions (another means for performing some linear transformations) uses a concatenation order consistent with the use of column matrices for vectors.

The choice to represent vectors as column matrices does have some effect on how we construct and multiply our matrices, which we will discuss in more detail in the following parts. In the cases where we do wish to indicate that a vector is represented as a row matrix, we'll display it with a transpose applied, like $\mathbf{b}^T$.

2.3.4 Block Matrices

A matrix can also be represented by submatrices, rather than by individual elements. This is also known as a block matrix. For example, the matrix

$$\begin{bmatrix} 2 & 3 & 0 \\ -3 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

can also be represented as

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ -3 & 2 \end{bmatrix}$$

and

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We will sometimes use this to represent a matrix as a set of row or column matrices. For example, if we have a matrix **A**

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix}$$

We can represent its rows as three vectors

$$\begin{aligned} \mathbf{a}_0^T &= \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \end{bmatrix} \\ \mathbf{a}_1^T &= \begin{bmatrix} a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix} \\ \mathbf{a}_2^T &= \begin{bmatrix} a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \end{aligned}$$

and represent **A** as

$$\begin{bmatrix} \mathbf{a}_0^T \\ \mathbf{a}_1^T \\ \mathbf{a}_2^T \end{bmatrix}$$

Similarly, we can represent a matrix **B** with its columns as three vectors

$$\begin{aligned} \mathbf{b}_0 &= \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \end{bmatrix} \\ \mathbf{b}_1 &= \begin{bmatrix} b_{0,1} \\ b_{1,1} \\ b_{2,1} \end{bmatrix} \\ \mathbf{b}_2 &= \begin{bmatrix} b_{0,2} \\ b_{1,2} \\ b_{2,2} \end{bmatrix} \end{aligned}$$

and subsequently **B** as

$$\begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 \end{bmatrix}$$

As mentioned earlier, the transpose notation tells us whether we're using row or column vectors.

2.3.5 Matrix Product

The primary operation we will apply to matrices is multiplication, also known as the matrix product. The product is important to us because it allows us to do two essential things. First, multiplying a matrix by a compatible vector will perform a linear transformation on the vector. Second, multiplying matrices together will create a single matrix that performs their combined linear transformations. We'll discuss how this is possible shortly, but first we must define how to perform matrix multiplication.

As with real numbers, the product **C** of two matrices **A** and **B** is represented as

$$\mathbf{C} = \mathbf{AB}$$

Computing the matrix product is not as simple as multiplying real numbers but is not that bad if you understand the process. To calculate a given element $c_{i,j}$ in the product, we take the dot product of row i from **A** with column j from **B**. We can express this symbolically as

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

As an example, we'll look at computing the first element of a 3×3 matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} b_{0,0} & \cdots & \cdots \\ b_{1,0} & \cdots & \cdots \\ b_{2,0} & \cdots & \cdots \end{bmatrix} = \begin{bmatrix} c_{0,0} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \vdots & \cdots & \ddots \end{bmatrix}$$

To compute the value of $c_{0,0}$, we take the dot product of row 1 from **A** and column 1 from **B**:

$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0}$$

Expanding this for a 2×2 matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} = \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} \end{bmatrix}$$

If we represent **A** as a collection of rows and **B** as a collection of columns, then

$$\begin{bmatrix} \mathbf{a}_0^T \\ \mathbf{a}_1^T \end{bmatrix} \begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{a}_0 \cdot \mathbf{b}_0 & \mathbf{a}_0 \cdot \mathbf{b}_1 \\ \mathbf{a}_1 \cdot \mathbf{b}_0 & \mathbf{a}_1 \cdot \mathbf{b}_1 \end{bmatrix}$$

We can also multiply by using block matrices:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}\begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix} = \begin{bmatrix} \mathbf{AE}+\mathbf{BG} & \mathbf{AF}+\mathbf{BH} \\ \mathbf{CE}+\mathbf{DG} & \mathbf{CF}+\mathbf{DH} \end{bmatrix}$$

Note that this is only allowable if the submatrices are conformable for addition and multiplication.

There is a restriction on which matrices can be multiplied together; in order to perform a dot product the two vectors have to have the same length. So to multiply together two matrices, the number of columns in the first (i.e., the width of each row) has to be the same as the number of rows in the second (i.e., the height of each column). Because of this restriction, the only matrices that can be multiplied by themselves are square.

In general, matrix multiplication is not commutative. As an example, if we multiply a row matrix by a column matrix, we perform a dot product:

$$\begin{bmatrix} 1 & 2 \end{bmatrix}\begin{bmatrix} 3 \\ 4 \end{bmatrix} = 1 \cdot 3 + 2 \cdot 4 = 11$$

Because of this, you may often see a dot product represented as

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b}$$

If we multiply them in the opposite order, we get a square matrix:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix}\begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix}$$

Even multiplication of square matrices is not necessarily commutative:

$$\begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 9 & 6 \\ 12 & 8 \end{bmatrix}$$
$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 7 & 14 \end{bmatrix}$$

Aside from the size restriction and not being commutative, the algebraic rules for matrix multiplication are very similar to those for real numbers:

1. $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$
2. $\mathbf{a}(\mathbf{BC}) = (\mathbf{aB})\mathbf{C}$
3. $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$

4. $(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$
5. $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$

where **A**, **B**, and **C** are matrices conformable for multiplication and a is a scalar. Note that matrix multiplication is still associative (rules 1 and 2) and distributive (rules 3 and 4).

2.3.6 Transforming Vectors

As previously indicated, matrices can be used to represent linear transformations on vectors. We do this by multiplying the matrix by the vector we wish to transform, or simply

$$\mathbf{b} = \mathbf{Ax}$$

Let's expand our terms and examine the components of the matrix and each vector:

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

This represents a transformation from an n-dimensional space V to an m-dimensional space W, so **x** has n components and the resulting vector **b** has m. In order for the multiplication to proceed, matrix **A** must be $m \times n$. As with general matrix multiplication, whenever we perform matrix–vector multiplication, the number of components in the multiplied vector must match the number of columns in the matrix, and the resulting vector will have a number of components equal to the number of rows.

To see how this operation performs a linear transformation, we'll use the fact that we only need to know where the basis of a vector space V is mapped to. Suppose that we know that our standard basis $\{\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-1}\}$ is transformed to $\{\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{n-1}\}$ in W, again using the standard basis. We will store, in order, each of these transformed basis vectors as the columns of **A**, or

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \cdots & \mathbf{a}_{n-1} \end{bmatrix}$$

Using our matrix multiplication definition to compute the product of **A** and a vector **x** in V, we see that the result for element i in **b** is

$$b_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

This is exactly the same as equation 2.5. So by setting up our matrix with the transformed basis vectors in each column, we can use matrix multiplication to perform linear transformations.

Column vectors aren't the only possibility. We can also premultiply by a vector by treating it as a row matrix:

$$\begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \cdots & x_{m-1} \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

or

$$\mathbf{c}^T = \mathbf{x}^T \mathbf{A}$$

In this case the rows of $\mathbf{A}$ are acting as our transformed basis vectors, and the number of components in $\mathbf{x}^T$ must match the number of rows in our matrix.

At this point we can define some additional properties for matrices. The *column space* of a matrix is the vector space spanned by the matrix's column vectors and is the range of the linear transformation performed by postmultiplying by a column vector. Correspondingly, the *row space* is the vector space spanned by the row vectors of the matrix and, as we'd expect, is the range of the linear transformation performed by premultiplying by a row vector. As it happens, the dimensions of the row space and column space are equal and that value is called the *rank* of the matrix. The matrix rank is equal to the rank of the associated linear transformation.

The column space and row space are not necessarily the same vector space. As an example, take the matrix

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

When postmultiplied by a column vector, it maps a vector (x, y, z) in $\mathbb{R}^3$ to a vector $(y, z, 0)$ on the xy-plane. Premultiplying by a row vector, on the other hand, maps (x, y, z) to $(0, x, y)$ on the yz-plane. They have the same dimension, and hence the same rank, but they are not the same vector space.

This makes a certain amount of sense. When we multiply by a row vector, we use the row vectors of the matrix as our transformed basis instead of the column vectors. To achieve the same result as the column vector multiplication, we need to change our matrix's column vectors to row vectors by taking the transpose:

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} y & z & 0 \end{bmatrix}$$

We can now see the purpose of the transpose: it exchanges a matrix's row space with its column space.

Like a linear transformation, a matrix also has a null space, which is all vectors $\mathbf{x}$ in V such that

$$\mathbf{Ax} = \mathbf{0}$$

In the preceding example, the null space N is all vectors with zero y and z components. As with linear transformations, $\dim(N)$+rank = $\dim(V)$.

2.3.7 Combining Linear Transformations

Suppose we have two transformations, $\mathcal{S} : U \to V$ and $\mathcal{T} : V \to W$, and we want to perform one after the other; namely, for a vector $\mathbf{x}$, we want the result $\mathcal{T}(\mathcal{S}(\mathbf{x}))$. If we know that we are going to transform a large collection of vectors by $\mathcal{S}$ and the resulting vectors by $\mathcal{T}$, it will be more efficient to find a single transformation that generates the same result so that we only have to transform the vectors once. This is known as the *composition* of $\mathcal{S}$ and $\mathcal{T}$ and is written as

$$(\mathcal{T} \circ \mathcal{S})(\mathbf{x}) = \mathcal{T}(\mathcal{S}(\mathbf{x}))$$

Composition (or alternatively, *concatenation*) of transformations is done via generalized matrix multiplication.

Suppose that matrix $\mathbf{A}$ is the corresponding transformation matrix for $\mathcal{S}$ and $\mathbf{B}$ is the corresponding matrix for $\mathcal{T}$. Recall that in order to set up $\mathbf{A}$ for vector transformation, we pretransform the standard basis vectors by $\mathcal{S}$ and store them as the columns of $\mathbf{A}$. Now we need to transform those vectors again, this time by $\mathcal{T}$. We could either do this explicitly or use the fact that multiplying by $\mathbf{B}$ will transform vectors (in V) by $\mathcal{T}$. So we just multiply each column of $\mathbf{A}$ by $\mathbf{B}$ and store the results, in order, as columns in a new matrix $\mathbf{C}$:

$$\mathbf{C} = \mathbf{BA}$$

If U has dimension n, V has dimension m, and W has dimension l, then $\mathbf{A}$ will be an $m \times n$ matrix and $\mathbf{B}$ will be an $l \times m$ matrix. Since the number of columns in **B** matches the number of rows in **A**, the matrix product can proceed, as we'd expect. The result **C** will be an $l \times n$ matrix and will apply the transformation of **A** followed by the transformation of **B** in a single matrix–vector multiplication.

This is the power of using matrices as a representation for linear transformations. By continually concatenating matrices, we can use the result to produce the effect of an entire series of transformations, in order, through a single matrix multiplication. Note that the order does matter. The preceding result **C** will perform the result of applying **A** followed by **B**. If we swap the terms (assuming they're still conformable under multiplication),

$$\mathbf{D} = \mathbf{A}\mathbf{B}$$

and matrix **D** will perform the result of applying **B** followed by **A**. This is almost certainly not the same transformation.

For the discussion thus far, we have assumed that the resulting matrix will be applied to a vector represented as a column matrix. It is good to be aware that the choice of whether to represent a vector as a row matrix or column matrix affects the order of multiplications when combining matrices. Suppose we multiply a column vector **u** by three matrices, where the intended transformation order is to apply $\mathbf{M}_0$, then $\mathbf{M}_1$, and finally $\mathbf{M}_2$:

$$\begin{aligned} \mathbf{v} &= \mathbf{M}_0\mathbf{u} \\ \mathbf{w} &= \mathbf{M}_1\mathbf{v} \\ \mathbf{x} &= \mathbf{M}_2\mathbf{w} \end{aligned} \tag{2.6}$$

If we take equation 2.6 and substitute $\mathbf{M}_1\mathbf{v}$ for **w** and then $\mathbf{M}_0\mathbf{u}$ for **v**, we get

$$\begin{aligned} \mathbf{x} &= \mathbf{M}_2\mathbf{M}_1\mathbf{v} \\ &= \mathbf{M}_2\mathbf{M}_1\mathbf{M}_0\mathbf{u} \\ &= \mathbf{M}_c\mathbf{u} \end{aligned}$$

Doing something similar for a row vector $\mathbf{a}^T$:

$$\begin{aligned} \mathbf{b}^T &= \mathbf{a}^T\mathbf{N}_0 \\ \mathbf{c}^T &= \mathbf{b}^T\mathbf{N}_1 \\ \mathbf{d}^T &= \mathbf{c}^T\mathbf{N}_2 \end{aligned}$$

and substituting:

$$\begin{aligned}\mathbf{d}^T &= \mathbf{b}^T\mathbf{N}_1\mathbf{N}_2\\ &= \mathbf{a}^T\mathbf{N}_0\mathbf{N}_1\mathbf{N}_2\\ &= \mathbf{a}^T\mathbf{N}_r\end{aligned}$$

The order difference is quite clear. When using row vectors and concatenating, matrix order follows the left to right progress used in English text. Column vectors work right to left instead, which may not be as intuitive. We will just need to be careful about our matrix order and transpose any matrices that assume we're using row vectors.

There are two other ways to modify transformation matrices that aren't used as often. Instead of concatenating two transformations, we may want to create a new one by adding two together: $\mathcal{Q}(\mathbf{x}) = \mathcal{S}(\mathbf{x})+\mathcal{T}(\mathbf{x})$. This is easily done by adding the corresponding matrices together, so the matrix that performs $\mathcal{Q}$ is $\mathbf{C} = \mathbf{A}+\mathbf{B}$. Another means we might use for generating a new transformation from an existing one is to scale it: $\mathcal{R}(\mathbf{x}) = s \cdot \mathcal{T}(\mathbf{x})$. The corresponding matrix is created by scaling the original matrix: $\mathbf{D} = s\mathbf{A}$.

2.3.8 Identity Matrix

We know that when we multiply a scalar or vector by 1, the result is the scalar or vector again:

$$1 \cdot x = x$$

Similarly, in matrix multiplication there is a special matrix known as the identity matrix, represented by the letter **I**. Thus,

$$\mathbf{A} \cdot \mathbf{I} = \mathbf{I} \cdot \mathbf{A} = \mathbf{A}$$

The identity matrix maps the basis vectors of the domain to the same vectors in the range; it performs a linear transformation that has no effect on the source vector: the identity transformation.

A particular identity matrix is a diagonal square matrix, where the diagonal is all 1s:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0\\ 0 & 1 & & 0\\ \vdots & & \ddots & \vdots\\ 0 & 0 & \cdots & 1\end{bmatrix}$$

If a particular $n \times n$ identity matrix is needed, it is sometimes referred to as $\mathbf{I}_n$. Take as an example $\mathbf{I}_3$:

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rather than referring to it in this way, we'll just use the term $\mathbf{I}$ to represent a general identity matrix and assume it is the correct size in order to allow an operation to proceed.

2.3.9 Performing Vector Operations with Matrices

Recall that if we multiply a row vector by a column vector, it performs a dot product:

$$\mathbf{w}^T \mathbf{v} = w_x v_x + w_y v_y + w_z v_z = \mathbf{v} \cdot \mathbf{w}$$

And multiplying them in the opposite order produces a square matrix:

$$\mathbf{T} = \mathbf{v}\,\mathbf{w}^T = \begin{bmatrix} v_x w_x & v_x w_y & v_x w_z \\ v_y w_x & v_y w_y & v_y w_z \\ v_z w_x & v_z w_y & v_z w_z \end{bmatrix}$$

This square matrix $\mathbf{T}$ is known as the *tensor product* $\mathbf{v} \otimes \mathbf{w}$. We can use it to rewrite vector expressions of the form $(\mathbf{u} \cdot \mathbf{v})\mathbf{w}$ as

$$(\mathbf{u} \cdot \mathbf{v})\mathbf{w} = (\mathbf{w} \otimes \mathbf{v})\mathbf{u}$$

In particular, we can rewrite a projection by a unit vector as

$$(\mathbf{u} \cdot \hat{\mathbf{v}})\hat{\mathbf{v}} = (\hat{\mathbf{v}} \otimes \hat{\mathbf{v}})\mathbf{u}$$

This will prove useful to us in the next chapter.

We can also perform our other vector product, the cross product, through a matrix multiplication. If we have two vectors $\mathbf{v}$ and $\mathbf{w}$ and we want to compute $\mathbf{v} \times \mathbf{w}$, we can replace $\mathbf{v}$ with a particular skew symmetric matrix, represented as $\tilde{\mathbf{v}}$:

$$\tilde{\mathbf{v}} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

Multiplying by **w** gives

$$\begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} v_y w_z - w_y v_z \\ v_z w_x - w_z v_x \\ v_x w_y - w_x v_y \end{bmatrix}$$

which is the formula for the cross product. This will also prove useful to us in subsequent chapters.

2.3.10 Implementation

Source Code
Library
IvMath
Filename
IvMatrix33
IvMatrix44

One might expect that the most natural data format for, say, a 3 × 3 matrix would be

```
class IvMatrix33
{
   float mData[3][3];
};
```

However, the memory layout of such a matrix is not ideal for our purposes. In C or C++, two-dimensional arrays are stored in what is called *row major order*, meaning that the matrix is stored in memory in a row by row order. If we use a one-dimensional array as our member variable instead:

```
class IvMatrix33
{
   float mV[9];
};
```

the index order for a 3 × 3 matrix is

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

The indexing operator for a row major matrix (we have to use `operator()` because `operator[]` only works for a single index) is

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
```

```
{
  return mV[col + 3*row];
}
```

Why won't this work? Well, in Direct3D matrices are expected to be used with row vectors. And even in OpenGL, despite the fact that the documentation is written using column vectors, the internal representation premultiplies the vectors; that is, it expects row vectors as well. Accordingly, since we're using column vectors, we will need to transpose our matrices before we pass them in as arguments to the graphics API. Doing this for every single matrix takes time and is a bit of nuisance to remember. Missing that one transpose can make debugging your algorithm a longer process than it needs to be.

The solution is to pretranspose the matrix in the storage representation. This is a format known as *column major order* and stores a matrix column by column instead of row by row. Writing out our indices in column major order gives us

$$\begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

Notice that the indices are the transpose of row major order. The indexing operator becomes

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
{
  return mV[row + 3*col];
}
```

Alternatively, if we want to use two-dimensional arrays:

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
{
  return mV[col][row];
}
```

Using column major format and column vectors, matrix–vector multiplication becomes

```
IvVector3
IvMatrix33::operator*( const IvVector3& vector ) const
```

```
{
    IvVector3 result;

    result.x = mV[0]*vector.x + mV[3]*vector.y + mV[6]*vector.z;
    result.y = mV[1]*vector.x + mV[4]*vector.y + mV[7]*vector.z;
    result.z = mV[2]*vector.x + mV[5]*vector.y + mV[8]*vector.z;

    return result;
}
```

and matrix–matrix multiplication is

```
IvMatrix33
IvMatrix33::operator*( const IvMatrix33& other ) const
{
    IvMatrix33 result;

    result.mV[0] = mV[0]*other.mV[0] + mV[3]*other.mV[1] + mV[6]*other.mV[2];
    result.mV[1] = mV[1]*other.mV[0] + mV[4]*other.mV[1] + mV[7]*other.mV[2];
    result.mV[2] = mV[2]*other.mV[0] + mV[5]*other.mV[1] + mV[8]*other.mV[2];

    result.mV[3] = mV[0]*other.mV[3] + mV[3]*other.mV[4] + mV[6]*other.mV[5];
    result.mV[4] = mV[1]*other.mV[3] + mV[4]*other.mV[4] + mV[7]*other.mV[5];
    result.mV[5] = mV[2]*other.mV[3] + mV[5]*other.mV[4] + mV[8]*other.mV[5];

    result.mV[6] = mV[0]*other.mV[6] + mV[3]*other.mV[7] + mV[6]*other.mV[8];
    result.mV[7] = mV[1]*other.mV[6] + mV[4]*other.mV[7] + mV[7]*other.mV[8];
    result.mV[8] = mV[2]*other.mV[6] + mV[5]*other.mV[7] + mV[8]*other.mV[8];

    return result;
}
```

Matrix addition is just

```
IvMatrix33
IvMatrix33::operator+( const IvMatrix33& other ) const
{
    IvMatrix33 result;

    for (int i = 0; i < 9; ++i)
```

```
        {
            result.mV[i] = mV[i]+other.mV[i];
        }

        return result;
    }
```

Scalar multiplication of matrices is similar.

It is common practice to refer to a matrix intended to be used with row vectors (i.e., its transformed basis vectors are stored as rows) as row major order and, similarly, to a matrix intended to be used with column vectors as column major order. This is incorrect terminology. Row and column major order refer only to the storage format; namely, where an element $a_{i,j}$ will lie in the one-dimensional representation of the matrix. Whether your matrix library intends for vectors to be pre- or postmultiplied should be independent of the underlying storage.

2.4 Systems of Linear Equations

2.4.1 Definition

Other than performing linear transformations, another purpose of matrices is to act as a mechanism for solving systems of linear equations. A general system of m linear equations with n unknowns is represented as

$$\begin{aligned}
b_0 &= a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1}\\
b_1 &= a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1}\\
\vdots\ &\ \ \vdots\\
b_{m-1} &= a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1}
\end{aligned} \tag{2.7}$$

The problem we are trying to solve is, Given $a_{0,0}, \ldots, a_{m-1,n-1}$ and $b_0, \ldots, b_{m-1}$, what are the values of $x_0, \ldots, x_{n-1}$? For a given linear system, the set of all possible solutions is called the *solution set*.

As an example, the system of equations

$$\begin{aligned}
x_0 + 2x_1 &= 1\\
3x_0 - x_1 &= 2
\end{aligned}$$

has the solution set $\{x_0 = 5/7, x_1 = 1/7\}$.

There may not be a single solution to the linear system. For example, the plane equation

$$ax + by + cz = -d$$

has an infinite number of solutions: the solution set for this example is all the points on the particular plane.

Alternatively, it may not be possible to find any solution to the linear system. Suppose that we have the linear system

$$\begin{aligned} x_0 + x_1 &= 1 \\ x_0 + x_1 &= 2 \end{aligned}$$

There are clearly no solutions for x and y. The solution set is the empty set.

Let's reexamine equation 2.7. If we think of $(x_0, \ldots, x_{n-1})$ as elements of an n-dimensional vector **x** and $(b_0, \ldots, b_{m-1})$ as elements of an m-dimensional vector **b**, then this starts to look a lot like matrix multiplication. We can rewrite this as

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix}$$

Or our old friend

$$\mathbf{Ax} = \mathbf{b}$$

The coefficients of the equation become the elements of matrix **A**, and matrix multiplication encapsulates our entire linear system. Now the problem becomes one of the form: Given **A** and **b**, what is **x**?

2.4.2 Solving Linear Systems

One case is very easy to solve. Suppose **A** looks like

$$\begin{bmatrix} 1 & a_{0,1} & \cdots & a_{0,n-1} \\ 0 & 1 & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

This is equivalent to the linear system

$$\begin{aligned}
b_0 &= x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} \\
b_1 &= x_1 + \cdots + a_{1,n-1}x_{n-1} \\
\vdots &\quad \vdots \\
b_{m-1} &= x_{n-1}
\end{aligned}$$

We see that we immediately have the solution to one unknown via $x_{n-1} = b_{m-1}$. We can substitute this value into the previous $m - 1$ equations and possibly solve for another x_i. If so, we can substitute that x_i into the remaining unsolved equations and so on up the chain. If there is a single solution for the system of equations, we will find it; otherwise, we will solve as many terms as possible and derive a solution set for the remainder.

This matrix is said to be in *row echelon form*. The formal definition for row echelon form is

1. If a row is entirely zeros, it will be below any nonzero rows of the matrix; in other words, all zero rows will be at the bottom of the matrix.
2. The first nonzero element of a row (if any) will be 1 (called a *leading 1*).
3. Each leading 1 will be to the right of a leading 1 in any preceding row.

If the following additional condition is met, we say that the matrix is in *reduced row echelon* form.

4. Each column with a leading 1 will be zero in the other rows.

The process we've described gives us a clue about how to proceed in solving general systems of linear equations. Suppose we can multiply both sides of our equation by a series of matrices so that the left-hand side becomes a matrix in row echelon form. Then we can use this in combination with the right-hand side to give us the solution for our system of equations.

However, we need to use matrices that preserve the properties of the linear system; the solution set for both systems of equations must remain equal. This restricts us to those matrices that perform one of three transformations called *elementary row operations*. These are

1. Multiply a row by a nonzero scalar.
2. Add a nonzero multiple of one row to another.
3. Swap two rows.

These three types of transformations maintain the solution set of the linear system while allowing us to reduce it to a simpler problem. The matrices that perform elementary row operations are called *elementary matrices*.

Some simple examples of elementary matrices include one which multiplies row 2 by a scalar a:

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

one which adds k times row 2 to row 1:

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

and one that swaps rows 2 and 3:

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

2.4.3 Gaussian Elimination

Source Code
Library
IvMath
Filename
IvGaussianElim

In practice we don't solve linear systems through matrix multiplication. Instead, it is more efficient to iteratively perform the operations directly on **A** and **b**. The most basic method for solving linear systems is known as *Gaussian elimination*, after Karl Friedrich Gauss, a prolific German mathematician of the eighteenth and nineteenth centuries. It involves concatenating the matrix **A** and vector **b** into a form called an *augmented matrix* and then performing a series of elementary row operations on the augmented matrix, in a particular order. This will either give us a solution to the system of linear equations or tell us that computing a single solution is not possible: either there is no solution or an infinite number of solutions.

To create the augmented matrix, we take the original matrix **A** and combine it with our constant vector **b**, for example,

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 3 \\ 4 & 5 & 6 & 2 \\ 7 & 8 & 9 & 1 \end{array}\right]$$

The vertical line within the matrix indicates the separation between **A** and **b**. To this augmented matrix, we will directly apply one or more of our row operations.

The process begins by looking at the first element in the first row. The first step is called a *pivoting* step. At the very least we need to ensure that we have a nonzero entry in the diagonal position, so if necessary we will swap this row with one of the lower rows with a nonzero entry in the same column. The element that we're swapping into place is called the *pivot* element, and swapping two rows to move the pivot element into place is known as *partial pivoting*. For better numerical precision, we usually go one step further and swap with the row that contains the element of largest absolute value. If no pivot element can be found, then there is no single solution and we abort.

Now let's say that the current pivot element value is k. We scale the entry row by $1/k$ to set the diagonal entry to 1. Finally, we set the column elements below the diagonal entry to zero by adding appropriate multiples of the current row. Then we move on to the next row and look at its diagonal entry. At the end of this process, our matrix will be in row echelon form.

Let's take a look at an example. Suppose we have the following system of linear equations:

$$\begin{array}{rrcrl} x & -3y & + & z & = 5 \\ 2x & -y & + & 2z & = 5 \\ 3x & +6y & + & 9z & = 3 \end{array}$$

The equivalent augmented matrix is

$$\left[\begin{array}{ccc|c} 1 & -3 & 1 & 5 \\ 2 & -1 & 2 & 5 \\ 3 & 6 & 9 & 3 \end{array}\right]$$

If we look at column 0, the maximal entry is 3, in row 2. So we begin by swapping row 2 with row 0:

$$\left[\begin{array}{ccc|c} 3 & 6 & 9 & 3 \\ 2 & -1 & 2 & 5 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

We scale the new row 0 by 1/3 to set the pivot element to 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 2 & -1 & 2 & 5 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

Now we start clearing the lower entries. The first entry in row 1 is 2, so we scale row 0 by -2 and add it to row 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -5 & -4 & 3 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

We do the same for row 2, scaling by −1 and adding:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -5 & -4 & 3 \\ 0 & -5 & -2 & 4 \end{array}\right]$$

We are done with row 0 and move on to row 1. Row 1, column 1 is the maximal entry in the column, so we don't need to swap rows. However, it isn't 1, so we need to scale row 1 by −1/5:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & -5 & -2 & 4 \end{array}\right]$$

We now need to clear element 1 of row 2 by scaling row 1 by 5 and adding:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & 0 & 2 & 1 \end{array}\right]$$

Finally we scale the bottom row by 1/2 to set the pivot element in the row to 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & 0 & 1 & 1/2 \end{array}\right]$$

This matrix is now in row-echelon form. We have two possibilities at this point. We could clear the upper triangle of the matrix in a fashion similar to how we cleared the lower triangle, but by working up from the bottom and adding multiples of rows. The solution **x** to the linear system would end up in the right-hand column. This is known as *Gauss-Jordan elimination*.

But let's look at the linear system we have now:

$$\begin{aligned} x + 2y + 3z &= 1 \\ y + 4/5z &= -3/5 \\ z &= 1/2 \end{aligned}$$

As expected, we already have a known quantity: z. If we plug z into the second equation, we can solve for y:

$$\begin{aligned} y &= -3/5 - 4/5z && (2.8) \\ &= -3/5 - 4/5(1/2) && (2.9) \\ &= -1 && (2.10) \end{aligned}$$

Once y is known, we can solve for x:

$$x = 1 - 2y - 3z \tag{2.11}$$
$$= 1 - 2(-1) - 3(1/2) \tag{2.12}$$
$$= 3/2 \tag{2.13}$$

So our final solution for $\mathbf{x}$ is $(3/2, -1, 1/2)$.

This process of substituting known quantities into our equations is called *back substitution*.

A summary of Gaussian elimination with back substitution follows:

```
for p = 1 to n do
    // find the element with largest absolute value in col p

    // if max is zero, stop!

    // if max element not in row p, swap rows

    // set pivot element to 1
    multiply row p by 1/A[p][p]

    // clear lower column entries
    for r = p+1 to n do
        subtract row p times A[r,p] from current row,
            so that element in pivot column becomes 0

// do backwards substitution
for row = n-1 to 1
    for col = row+1 to n
        // subtract out known quantities
        b[row] = b[row] - A[row][col]*b[col]
```

The pseudocode shows what may happen when we encounter a linear system with no single solution. If we can't swap a nonzero entry in the pivot location, then there is a column that is all zeros. This is only possible if the rank of the matrix (i.e., the number of linearly independent column vectors) is less than the number of unknowns. In this case there is no solution to the linear system and we abort.

In general, we can state that if the rank of the coefficient matrix $\mathbf{A}$ equals the rank of the augmented matrix $\mathbf{A}|\mathbf{b}$, then there will be at least one solution to the linear system. If the two ranks are unequal, then there are no solutions. There is a single solution only if the rank of $\mathbf{A}$ is equal to the minimum of the number of rows or columns of $\mathbf{A}$.

2.5 Matrix Inverse

This may seem like a lot of trouble to go to solve a simple equation like $\mathbf{b} = \mathbf{A}\mathbf{x}$. If this were scalar math, we could simply divide both sides of the equation by $\mathbf{A}$ to get

$$\mathbf{x} = \mathbf{b}/\mathbf{A}$$

Unfortunately, matrices don't have a division operation. However, we can use an equivalent concept: the inverse.

2.5.1 Definition

In scalar multiplication, the inverse is defined as the reciprocal:

$$x \cdot \frac{1}{x} = 1$$

or

$$x \cdot x^{-1} = 1$$

Correspondingly, for a given matrix $\mathbf{A}$, we can define its inverse $\mathbf{A}^{-1}$ as a matrix such that

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$$

and

$$\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I}$$

There are a few things that fall out from this definition. First of all, in order for the first multiplication to occur, the number of rows in the inverse must be the same as the number of columns in the original matrix. For the second to occur, the converse is true. So the matrix and its inverse must be square and the same size. Since not all matrices are square, it's clear that not every matrix has an inverse.

Second, the inverse of the inverse returns the original matrix. Given

$$\mathbf{A}^{-1} \cdot (\mathbf{A}^{-1})^{-1} = \mathbf{I}$$

and

$$\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I}$$

then

$$(\mathbf{A}^{-1})^{-1} = \mathbf{A}$$

Even if a matrix is square, there isn't always an inverse. An extreme example is the zero matrix. Any matrix multiplied by this gives the zero matrix, so there is no matrix multiplication that will produce the identity. Another set of examples is matrices that have a zero row or column vector. Multiplying by such a row or column will return a dot product of zero, so you'll end up with a zero row or column vector in the product as well—again, not the identity matrix. In general, if the null space of the matrix is nonzero, then the matrix is non-invertible; that is, the matrix is only invertible if the rank of the matrix is equal to the number of rows and columns.

Given these identities, we can now solve for our preceding linear system. Recall that the equation was

$$\mathbf{Ax} = \mathbf{b}$$

If we multiply both sides by $\mathbf{A}^{-1}$, then

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{Ix} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

Therefore, if we could find the inverse of $\mathbf{A}$, we could use it to solve for $\mathbf{x}$. This is not usually a good idea, computationally speaking. It's usually cheaper to solve for $\mathbf{x}$ directly, rather than generating the inverse and then performing the matrix multiplication. The latter can also lead to increased numerical error. However, sometimes finding the inverse is a necessary evil.

The left-hand side of the above derivation shows us that we can think of the inverse $\mathbf{A}^{-1}$ as undoing the effect of $\mathbf{A}$. If we start with $\mathbf{Ax}$ and premultiply by $\mathbf{A}^{-1}$, we get back $\mathbf{x}$, our original vector.

We can find the inverse of a matrix using Gaussian elimination to solve for it column by column. Suppose we call the first column of $\mathbf{A}^{-1}$ $\mathbf{x}_0$. We can represent this as

$$\mathbf{x}_0 = \mathbf{A}^{-1}\mathbf{e}_0$$

where, as we recall, $\mathbf{e}_0 = (1, 0, \ldots, 0)$. Multiplying both sides by $\mathbf{A}$ gives

$$\mathbf{Ax}_0 = \mathbf{e}_0$$

Finding the solution to this linear system gives us the first column of $\mathbf{A}^{-1}$. We can do the same for the other columns, but using $\mathbf{e}_1$, $\mathbf{e}_2$, and so on. Instead of solving these one at a time, though, it is more efficient to create an augmented matrix with **A** and $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$ as columns on the right — or just **I**. For example,

$$\left[\begin{array}{ccc|ccc} 2 & 0 & 4 & 1 & 0 & 0 \\ 0 & 3 & -9 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

If we use Gauss-Jordan elimination to turn the left-hand side of the augmented matrix into the identity matrix, then we will end up with the inverse (if any) on the right-hand side. So from here we perform our elementary row operations as before. The maximal entry is already in the pivot point, so we scale the first row by 1/2:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1/2 & 0 & 0 \\ 0 & 3 & -9 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

The nonpivot entries in the first column are zero, so we move to the second column. Scaling the second row by 1/3 to set the pivot point to 1 gives us

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1/2 & 0 & 0 \\ 0 & 1 & -3 & 0 & 1/3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Again, our nonpivot entries in the second column are 0, so we move to the third column. Our pivot entry is 1, so we don't need to scale. We add −2 times the last row to the first row to clear that entry, then 3 times the last row to the second row to clear that entry, and get

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1/2 & 0 & -2 \\ 0 & 1 & 0 & 0 & 1/3 & 3 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

The inverse of our original matrix is now on the right-hand side of the augmented matrix.

2.5.2 Simple Inverses

Gaussian elimination, while useful, is unnecessary for computing the inverse of many of the matrices we will be using. The majority of matrices that we will encounter in games and 3D applications have simple inverses, and knowing the form of the matrix can make computing the inverse trivial.

One case is that of an *orthogonal* matrix, where the component row or column vectors are orthonormal. Recall that this means that the vectors are of unit length and perpendicular. If a matrix **A** is orthogonal, its inverse is the transpose:

$$\mathbf{A}^{-1} = \mathbf{A}^T$$

One example of an orthogonal matrix is

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

Another simple case is a diagonal matrix with nonzero elements in the diagonal. The inverse of such a matrix is also diagonal, where the new diagonal elements are the reciprocal of the original diagonal elements, as shown by the following:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}^{-1} = \begin{bmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1/c \end{bmatrix}$$

The third case is a modified identity matrix, where the diagonal is all ones but one column or row is nonzero. One such 3×3 matrix is

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

For a matrix of this form, we simply negate the non-zero elements to invert it. Using the previous example:

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, we can combine this knowledge to take advantage of an algebraic property of matrices. If we have two square matrices **A** and **B**, both of which are invertible, then

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

So if we know that our current matrix is the product of any of the cases we've just discussed, we can easily compute its inverse using the preceding formula. This will prove to be useful in subsequent chapters.

2.6 The Determinant

2.6.1 Definition

The determinant is a scalar quantity created by evaluating the elements of a square matrix. In real vector spaces, it acts as a general measure of how vectors transformed by the matrix change in size. For example, if we take the columns of a 2×2 matrix (i.e., the transformed basis vectors) and use them as the sides of a parallelogram (Figure 2.2), then the absolute value of the determinant is equal to the area of a parallelogram. For a 3×3 matrix, the absolute value of the determinant is equal to the volume of a parallelpiped described by the three transformed basis vectors (Figure 2.3).

The sign of the determinant depends on whether or not we have switched our ordered basis vectors from being relatively right-handed to being left-handed. In Figure 2.2, the shortest angle from $\mathbf{a}_0$ to $\mathbf{a}_1$ is clockwise, so they are left-handed. The determinant, therefore, is negative.

We represent the determinant in one of two ways, either $\det(\mathbf{A})$ or $|\mathbf{A}|$. The second is often used when showing the elements of a matrix:

$$\det(\mathbf{A}) = \begin{vmatrix} 1 & -3 & 1 \\ 2 & -1 & 2 \\ 3 & 6 & 9 \end{vmatrix}$$

The diagrams showing area of a parallelogram and volume of a parallelpiped should look familiar from our discussion of cross product and triple scalar product. In fact, the cross product is sometimes represented as

$$\mathbf{v} \times \mathbf{w} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$

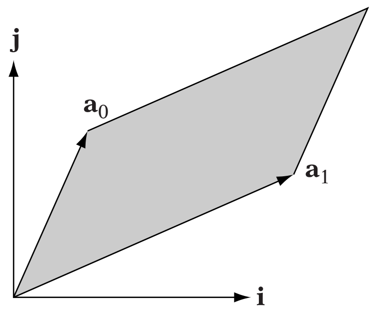


Figure 2.2 Determinant of 2×2 matrix as area of parallelogram bounded by transformed basis vectors $\mathbf{a}_0$ and $\mathbf{a}_1$.

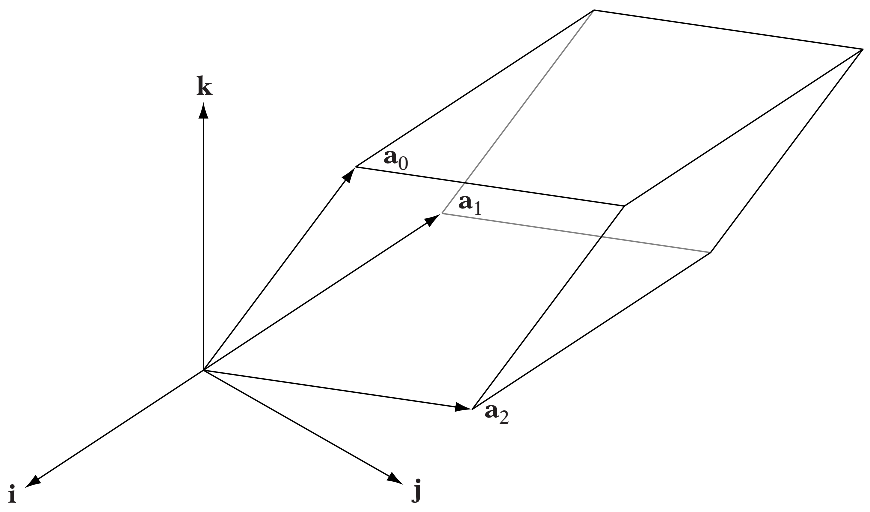


Figure 2.3 Determinant of 3 × 3 matrix as volume of parallelopiped bounded by transformed basis vectors $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2$.

while the triple product is represented as

$$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$

Since $\det(\mathbf{A}^T) = \det(\mathbf{A})$, this representation is equivalent.

2.6.2 Computing the Determinant

There are a few ways of representing the determinant computation for a specific matrix **A**. A standard recursive definition, choosing any row i, is

$$\det(\mathbf{A}) = \sum_{j=1}^{n} a_{i,j}(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

Alternatively, we can expand by column j instead:

$$\det(\mathbf{A}) = \sum_{i=1}^{n} a_{i,j}(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

In both cases, $\tilde{\mathbf{A}}_{i,j}$ is the submatrix formed by removing the ith row and jth column from $\mathbf{A}$. The base case is the determinant of a matrix with a single element, which is the element itself.

The term $\det(\tilde{\mathbf{A}}_{i,j})$ is also referred to as the *minor of entry* $a_{i,j}$, and the term $(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$ is called the *cofactor of entry* $a_{i,j}$.

The first formula tells us: for a given row i, we multiply each row entry $a_{i,j}$ by the determinant of the submatrix formed by removing row i and column j and either add or subtract it to the total depending on its position in the matrix. The second does the same but moves along column j instead of row i.

Let's compute an example determinant, expanding by row 0:

$$\det\left(\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix}\right) = ?$$

The first element of row 0 is 1, and the submatrix with row 0 and column 0 removed is

$$\begin{bmatrix} 4 & -3 \\ 6 & -5 \end{bmatrix}$$

The second element is also 1. However, we negate it since we are considering row 0 and column 1: $0 + 1 = 1$, which is odd. The submatrix is $\mathbf{A}$ with row 0 and column 1 removed:

$$\begin{bmatrix} 2 & -3 \\ 3 & -5 \end{bmatrix}$$

The third element of the row is 2, with the submatrix

$$\begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix}$$

We don't negate since we are considering row 0 and column 2: $0 + 2 = 2$, which is even.

So the determinant is

$$\begin{aligned} \det(\mathbf{A}) &= 1 \cdot \begin{vmatrix} 4 & -3 \\ 6 & -5 \end{vmatrix} - 1 \cdot \begin{vmatrix} 2 & -3 \\ 3 & -5 \end{vmatrix} + 2 \cdot \begin{vmatrix} 2 & 4 \\ 3 & 6 \end{vmatrix} \\ &= -1 \end{aligned}$$

In general, the determinant of a 2×2 matrix is

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = a \cdot \det([d]) - b \cdot \det([c]) = ad - bc$$

And the determinant of a 3×3 matrix is

$$\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) = a \cdot \det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b \cdot \det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c \cdot \det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right)$$

or

$$a(ei - fh) - b(di - fg) + c(dh - eg)$$

There are some additional properties of the determinant that will be useful to us. If we have two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$, the following hold:

1. $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$
2. $\det(\mathbf{A}^{-1}) = \dfrac{1}{\det(\mathbf{A})}$

We can look at the value of the determinant to tell us some features of our matrix. First of all, as we have mentioned, any matrix that transforms our basis vectors from right-handed to left-handed will have a negative determinant. If the matrix is also orthogonal, we call a matrix of this type a *reflection*. We will learn more about reflection matrices in the next chapter.

Then there are matrices that have a determinant of 1. The matrices we will encounter most often with this property are orthogonal matrices, where the handedness of the resulting basis stays the same (i.e., a right-handed basis is transformed to a right-handed basis). Figure 2.4 provides an example. Our transformed basis vectors are $(-\sqrt{2}/2, \sqrt{2}/2)$ and $(\sqrt{2}/2, \sqrt{2}/2)$. They remain orthonormal, so their area is just the product of the lengths of the two vectors, or 1×1 or 1. This type of matrix is called a *rotation*. As with reflections, we'll see more of rotations in the next chapter.

Finally, if the determinant is 0, then we know that the matrix has no inverse. The obvious case is if the matrix has a row or column of all 0s. Look again at our formula for the determinant. Suppose row i is all 0s. Multiplying all the submatrices against this row and summing together will clearly give us 0 as a result. The same is true for a zero column. The other and related possibility is that we have a linearly dependent row or column vector. In both cases the rank of the matrix is less than n—the size of the matrix— and therefore the matrix does not have an inverse. So if the determinant of a matrix is 0, we know the matrix is not invertible.

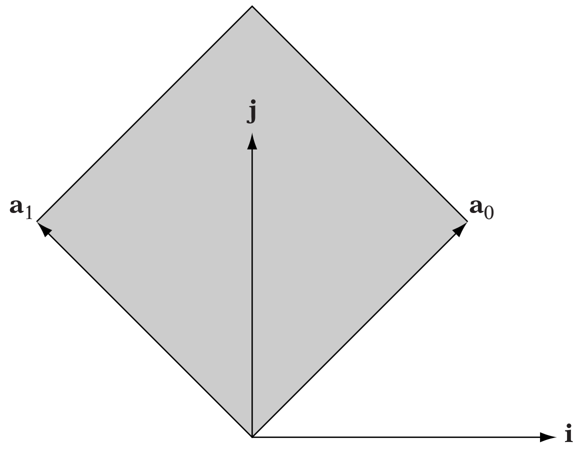


Figure 2.4 Determinant of example 2 × 2 orthogonal matrix.

2.6.3 Determinants and Elementary Row Operations

Source Code
Library
IvMath
Filename
IvGaussianElim

For 2 × 2 and 3 × 3 matrices, computing the determinant in this manner is a simple process. However, for larger and larger matrices, our recursive definition becomes unwieldy, and for large enough n will take an unreasonable amount of time to compute. In addition, computing the determinant in this manner can lead to floating point precision problems. Fortunately, there is another way.

Suppose we have an upper triangular matrix **U**. The first part of the determinant sum is $u_{0,0}\tilde{\mathbf{U}}_{0,0}$. The other terms, however, are 0, because the first column with the first row removed is all 0s. So the determinant is just

$$\det(\mathbf{U}) = u_{0,0}\tilde{\mathbf{U}}_{0,0}$$

If we expand the recursion, we find that the determinant is the product of all the diagonal elements, or

$$\det(\mathbf{U}) = u_{0,0}u_{1,1}\dots u_{nn}$$

As we did when solving linear systems, we can use Gaussian elimination to change our matrix into row echelon form, which is an upper triangular matrix. However, this assumes that elementary row operations have no effect on the determinant, which is not the case. Let's look at a few examples.

Suppose we have the matrix

$$\begin{bmatrix} 2 & -4 \\ -1 & 1 \end{bmatrix}$$

The determinant of this matrix is −2. If we multiply the first row by 1/2, we get

$$\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix}$$

which has a determinant of −1. Multiplying a row by a scalar k multiplies the determinant by k as well.

Now suppose we add two times the first row to the second one. We get

$$\begin{bmatrix} 1 & -2 \\ 1 & -3 \end{bmatrix}$$

which also has a determinant of −1. Adding a multiple of one row to another has no effect on the determinant.

Finally we can swap row 1 with row 2:

$$\begin{bmatrix} 1 & -3 \\ 1 & -2 \end{bmatrix}$$

which has a determinant of 1. Swapping two rows or two columns changes the sign of the determinant.

The effect of elementary row operations on the determinant can be summarized as follows:

Multiply row by k:	Multiplies determinant by k
Add multiple of one row to another:	No effect
Swap rows:	Changes sign of determinant

So our approach for calculating the determinant for a general matrix is this: as we perform Gaussian elimination, we keep a running product p of any multiplies we do to create leading 1s and negate p for every row swap. If we find a zero column when we look for a pivot element, we know the determinant is 0 and return such.

Let's suppose our final product is p. This represents what we've multiplied the determinant of our original matrix by to get the determinant of the final matrix $\mathbf{A}'$, or

$$p \cdot \det(\mathbf{A}) = \det(\mathbf{A}')$$

so

$$\det(\mathbf{A}) = \frac{1}{p} \cdot \det(\mathbf{A}')$$

We know that the determinant of $\mathbf{A}'$ is 1, since the diagonal of the row echelon matrix is all 1s. So our final determinant is just $1/p$. However, this is just the product of the multiplies we do to create leading 1s, and -1 for every row swap, or

$$p = \frac{1}{p_{0,0}} \frac{1}{p_{1,1}} \cdots \frac{1}{p_{n,n}} (-1)^k$$

where k is the number of row swaps. Then

$$1/p = p_{0,0} p_{1,1} \ldots p_{n,n} (-1)^k$$

So all we need to do is multiply our running product by each pivot element and negate for each row swap. At the end of our Gaussian elimination process, our running product will be the determinant we seek.

2.6.4 Adjoint Matrix and Inverse

Source Code
Library
IvMath
Filename
IvMatrix33

Recall that the cofactor of an entry $a_{i,j}$ is

$$C_{i,j} = (-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

For an $n \times n$ matrix, we can construct a corresponding matrix where we replace each element with its corresponding cofactor, or

$$\begin{bmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,n-1} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix}$$

This is called the matrix of cofactors from $\mathbf{A}$, and its transpose is the *adjoint matrix* $\mathrm{A}^{\mathrm{adj}}$.

Gabriel Cramer, a Swiss mathematician, showed that the inverse of a matrix can be computed from the adjoint by

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \mathrm{A}^{\mathrm{adj}}$$

Many graphics engines use *Cramer's method* to compute the inverse, and for 3×3 and 4×4 matrices it's not a bad choice; for matrices of this size Cramer's method is actually faster than Gaussian elimination. Because of this, we have chosen to implement `IvMatrix33::Inverse()` using an efficient form of Cramer's method.

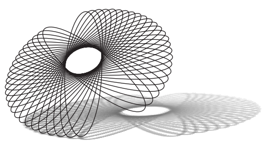
However, whether you're using Gaussian elimination or Cramer's method, you're probably doing more work than is necessary for the matrices we will encounter. Most will be in one of the formats described in Section 2.5.2 or a multiple of these matrix types. Using the process described in that section, you can compute the inverse by decomposing the matrix into a set of these types, inverting the simple matrices, and multiplying in reverse order to compute the matrix. This is often faster than either Gaussian elimination or Cramer's method and can be more tolerant of floating point errors because you can find near-exact solutions for the simple matrices.

2.7 Chapter Summary

In this chapter, we've discussed the general properties of linear transformations and how they are represented and performed by matrices. Matrices can also be used to compute solutions to linear systems of equations by using either Gaussian elimination or similar methods. We covered some basic matrix properties, the concepts of matrix identity and inverse (and various methods for calculating the latter), and the meaning and calculation of the determinant. This lays the foundation for what we'll be discussing in the next chapter: using matrix transformations to manipulate models in a three-dimensional world.

For those who are interested in reading further, Anton and Rorres [3] is a standard reference for many first courses in linear algebra. Other texts with slightly different approaches include Axler [7] and Friedberg [37]. More information on Gaussian elimination and its extensions such as LU decomposition can be found in Anton and Rorres [3], as well as in the *Numerical Recipes* series [92]. Finally, Blinn has an excellent article in his collection *Notation, Notation, Notation* [13] on the geometry underlying 2×2 matrix operations.

Chapter 3 Affine Transformations



3.1 Introduction

Now that we've chosen a mathematically sound basis for representing geometry in our game and discussed some aspects of matrix arithmetic, we need to combine them into an efficient method for placing and moving virtual objects or models. There are a few reasons we seek this efficiency. Suppose we wish to build a core level in our game space, say the office of a computer company. We could build all of our geometry in place and hard-code all of the locations. However, if we have a number of objects that are duplicated throughout the space — computers, desks, and chairs for example — it would be more memory-efficient to create one master copy of the geometry for each type of object. Then, for each instance of a particular object, we can specify just a position and orientation and let the rendering and simulation engine handle the placement.

Another, more obvious reason is that objects in games generally move so that setting them at a fixed location is not practical. We will need to have some means to specify, for a model as a whole, its position and orientation in space.

There are a few characteristics we desire in our method. We want it to be fast and work well with our existing data and math library. We want to be able to concatenate a series of operations so we can perform them with a single operation, just as we did with linear transformations. Since our objects consist of collections of points, we need our method to work on points in an affine space, but we'll still need to transform vectors as well. The specific method we will use is called an *affine transformation*.

3.2 Affine Transformations

3.2.1 Definition

In the last chapter, we discussed linear transformations, which map from one vector space to another. We can apply such transformations to vectors using matrix operations. There is a nearly equivalent set of transformations that map between affine spaces, which we can apply to points and vectors in an affine space. These are known as *affine transformations* and they too can be applied using matrix operations, albeit in a slightly different form.

Recall that linear transformations preserve the linear operations of vector addition and scalar multiplication. In other words, linear transformations map from one vector space to another and preserve linear combinations. Thus, for a given linear transformation $\mathcal{S}$:

$$\mathcal{S}(a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1}) = a_1\mathcal{S}(\mathbf{v}_0) + a_1\mathcal{S}(\mathbf{v}_1) + \cdots + a_{n-1}\mathcal{S}(\mathbf{v}_{n-1})$$

Correspondingly, an affine transformation $\mathcal{T}$ maps between two affine spaces A and B and preserves affine combinations. For scalars $a_0, \ldots, a_{n-1}$ and points $P_0, \ldots, P_{n-1}$ in A:

$$\mathcal{T}(a_0P_0 + \cdots + a_{n-1}P_{n-1}) = a_0\mathcal{T}(P_0) + \cdots + a_{n-1}\mathcal{T}(P_{n-1})$$

where $a_0 + \cdots + a_{n-1} = 1$.

As with our test for linear transformations, to determine whether a given transformation $\mathcal{T}$ is an affine transformation, it is sufficient to test a single affine combination:

$$\mathcal{T}(a_0P_0 + a_1P_1) = a_0\mathcal{T}(P_0) + a_1\mathcal{T}(P_1)$$

where $a_0 + a_1 = 1$.

Affine transformations are particularly useful to us because they preserve certain properties of geometry. First, they maintain collinearity, so points on a line will remain collinear and points on a plane will remain coplanar when transformed.

If we transform a line:

$$\begin{aligned} L(t) &= (1-t)P_0 + tP_1 \\ \mathcal{T}(L(t)) &= \mathcal{T}((1-t)P_0 + tP_1) \\ &= (1-t)\mathcal{T}(P_0) + t\mathcal{T}(P_1) \end{aligned}$$

The result is clearly still a line (assuming $\mathcal{T}(P_0)$ and $\mathcal{T}(P_1)$ aren't coincident). Similarly, if we transform a plane:

$$\begin{aligned} P(t) &= (1-s-t)P_0 + sP_1 + tP_2 \\ \mathcal{T}(P(t)) &= \mathcal{T}((1-s-t)P_0 + sP_1 + tP_2) \\ &= (1-s-t)\mathcal{T}(P_0) + s\mathcal{T}(P_1) + t\mathcal{T}(P_2) \end{aligned}$$

The result is clearly a plane (assuming $\mathcal{T}(P_0)$, $\mathcal{T}(P_1)$, and $\mathcal{T}(P_2)$ aren't collinear).

The second property of affine transformations is that they preserve relative proportions. The point that lies at t distance between P_0 and P_1 on the original line will map to the point that lies at t distance between $\mathcal{T}(P_0)$ and $\mathcal{T}(P_1)$ on the transformed line.

Note that while ratios of distances remain constant, angles and exact distances don't necessarily stay the same. The specific subset of affine transformations that preserve these features are called *rigid transformations*; those that don't are called *deformations*. It should be no surprise that we find rigid transformations useful. When transforming our models, in most cases we don't want them distorted unrecognizably. A bottle should maintain its size and shape—it should look like a bottle no matter where we place it in space. However, the deformations have their use as well. On occasion we may want to make an object larger or smaller or reflect it across a plane, as in a mirror.

To apply an affine transformation to a vector in an affine space, we can apply it to the difference of two points that equal the vector, or

$$\mathcal{T}(\mathbf{v}) = \mathcal{T}(P - Q) = \mathcal{T}(P) - \mathcal{T}(Q)$$

As we will see, an affine transformation that is applied to a vector performs a linear transformation.

3.2.2 Representation

Suppose we have an affine transformation that maps from affine spaces A and B, where the frame for A has basis vectors $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$ and origin O_A, and the frame for B has basis vectors $(\mathbf{w}_0, \ldots, \mathbf{w}_{m-1})$ and origin O_B. If we apply an affine transformation to a point $P = (x_0, \ldots, x_{n-1})$ in A, this gives

$$\begin{aligned}\mathcal{T}(P) &= \mathcal{T}(x_0\mathbf{v}_0 + \cdots + x_{n-1}\mathbf{v}_{n-1} + O_A)\\ &= x_0\mathcal{T}(\mathbf{v}_0) + \cdots + x_{n-1}\mathcal{T}(\mathbf{v}_{n-1}) + \mathcal{T}(O_A)\end{aligned}$$

As we did with linear transformations, we can express a given $\mathcal{T}(\mathbf{v})$ in terms of B's frame:

$$\mathcal{T}(\mathbf{v}_j) = a_{0,j}\mathbf{w}_0 + a_{1,j}\mathbf{w}_1 + \cdots + a_{m-1,j}\mathbf{w}_{m-1}$$

Similarly, we can express $\mathcal{T}(O_A)$ in terms of B's frame:

$$\mathcal{T}(O_A) = y_0\mathbf{w}_0 + y_1\mathbf{w}_1 + \cdots + y_{m-1}\mathbf{w}_{m-1} + O_B$$

Again, as we did with linear transformations, we can rewrite this as a matrix product. However, unlike linear transformations, we write a mapping from an n-dimensional affine space to an m-dimensional affine space as an $(m+1) \times (n+1)$ matrix:

$$\begin{bmatrix} a_{0,0}\mathbf{w}_0 & a_{0,1}\mathbf{w}_0 & \cdots & a_{0,n-1}\mathbf{w}_0 & y_0\mathbf{w}_0 \\ a_{1,0}\mathbf{w}_1 & a_{1,1}\mathbf{w}_1 & \cdots & a_{1,n-1}\mathbf{w}_1 & y_1\mathbf{w}_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0}\mathbf{w}_{m-1} & a_{m-1,1}\mathbf{w}_{m-1} & \cdots & a_{m-1,n-1}\mathbf{w}_{m-1} & y_{m-1}\mathbf{w}_{m-1} \\ 0 & 0 & \cdots & 0 & O_B \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ 1 \end{bmatrix}$$

The $n+1$ columns represent the n transformed basis vectors plus the transformed origin. We need $m+1$ rows since the frame of B has m basis vectors plus the origin O_B. As we can see, in order to allow the multiplication to proceed, we'll represent our point with a trailing "1" component.

We can pull out the frame terms to get

$$\begin{bmatrix}\mathbf{w}_0 & \mathbf{w}_1 & \cdots & \mathbf{w}_{m-1} & O_B\end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} & y_0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} & y_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} & y_{m-1} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ 1 \end{bmatrix}$$

So, similar to linear transformations, if we know how the affine transformation affects the frame for A, we can copy the transformed frame *in terms of the frame for B* into the columns of a matrix and use matrix multiplication to apply the affine transformation to an arbitrary point. We can represent this process of transformation using block matrices:

$$\mathcal{T}(P) = \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{Ax} + \mathbf{y} \\ 1 \end{bmatrix} \tag{3.1}$$

For the purposes of computation, the vector $\mathbf{0}^T$, the 1 in the lower right-hand corner of the matrix, and the trailing 1s in the points are unnecessary. They take up memory and using the full matrix takes additional instructions to multiply by constant values. Because of this, an affine transformation matrix is sometimes represented in a form where these constant terms are implied. This form is often either an $m \times (n + 1)$ matrix or, simpler still, a matrix multiplication plus a vector add:

$$\mathbf{Ax} + \mathbf{y}$$

where $\mathbf{x}$ consists of the point coordinates $(x_0, \ldots, x_{n-1})$ without the trailing 1. The matrix $\mathbf{A}$ is an $m \times n$ matrix, and we need at least $n+1$ columns in a matrix if we're going to multiply it by an n-dimensional point, so the multiplication $\mathbf{A}P$ is not considered mathematically legal in this case.

If we subtract two points in an affine space, we get a vector:

$$\begin{aligned} \mathbf{v} &= P_0 - P_1 \\ &= \begin{bmatrix} \mathbf{x}_0 \\ 1 \end{bmatrix} - \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_0 - \mathbf{x}_1 \\ 0 \end{bmatrix} \end{aligned}$$

As we can see, a vector is represented in an affine space with a trailing 0. As previously noted in Chapter 1, this provides justification for some math libraries to use the trailing 1 on points and trailing 0 on vectors. If we multiply a vector using this representation by our $(m + 1) \times (n + 1)$ matrix, expanding terms:

$$\begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} & y_0 \\ a_{1,0} & \cdots & a_{1,n-1} & y_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} & y_{m-1} \\ 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \\ 0 \end{bmatrix} = \begin{bmatrix} a_{0,0}v_0 + \cdots + a_{0,n-1}v_{n-1} \\ a_{1,0}v_0 + \cdots + a_{1,n-1}v_{n-1} \\ \vdots \\ a_{m-1,0}v_0 + \cdots + a_{m-1,n-1}v_{n-1} \\ 0 \end{bmatrix}$$

we see that the vector is affected by the upper left $m \times n$ matrix $\mathbf{A}$, but not the vector $\mathbf{y}$. This has the same effect on the first n elements of $\mathbf{v}$ as multiplying an n-dimensional vector by $\mathbf{A}$, which is a linear transformation. So this representation allows us to use affine transformation matrices to apply linear transformations on vectors in an affine space.

Suppose we wish to concatenate two affine transformations $\mathcal{S}$ and $\mathcal{T}$, where the matrix representing $\mathcal{S}$ is

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

and the matrix representing $\mathcal{T}$ is

$$\begin{bmatrix} \mathbf{B} & \mathbf{z} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

As with linear transformations, to find the matrix that represents the composition of $\mathcal{S}$ and $\mathcal{T}$, we multiply the matrices together. This gives

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{B} & \mathbf{z} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{AB} & \mathbf{Az}+\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.2}$$

Finding the inverse for an affine transformation is equally as straightforward; again, we can use a process similar to the one we used with linear transformation matrices. Starting with

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{0}^T & 1 \end{bmatrix}$$

we multiply by both sides to remove the **y** component from the left-most matrix:

$$\begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

We then multiply by both sides to change the left-most matrix to the identity:

$$\begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.3}$$

thereby giving us the inverse on the right-hand side.

When we're working in $\mathbb{R}^3$, $\mathbf{A}$ will be a 3×3 matrix and $\mathbf{y}$ will be a 3-vector; hence the full affine matrix will be a 4×4 matrix. Most graphics libraries expect transformations to be in the 4×4 matrix form, so if we do use the more compact forms in our math library to save memory, we will still have to expand them before rendering our objects. Because of this, we will use the 4×4 form for our following discussions, with the understanding that in our ultimate implementation we may choose one of the other forms for efficiency's sake.

3.3 Standard Affine Transformations

Now that we've defined affine transformations in general, we can discuss some specific affine transformations that will prove useful when manipulating objects in our game. We'll cover these in terms of transformations from $\mathbb{R}^3$ to $\mathbb{R}^3$, since they will be the most common uses. However, we can apply similar principles to find transformations from $\mathbb{R}^2$ to $\mathbb{R}^2$ or even $\mathbb{R}^4$ to $\mathbb{R}^4$ if we desire.

Since affine space A and B are the same in this case, to simplify things we'll use the same frame for each one: the standard Cartesian frame of $(\mathbf{i}, \mathbf{j}, \mathbf{k}, O)$.

3.3.1 Translation

The most basic affine transformation is *translation*. For a single point, it's the same as adding a vector $\mathbf{t}$ to it, and when applied to an entire set of points it has the effect of moving them rigidly through space (Figure 3.1). Since all the points are shifted equally in space, the size and shape of the object will not change, so this is a rigid transformation.

We can determine the matrix for a translation by computing the transformation for each of the frame elements. For the origin O, this is

$$\begin{aligned}\mathcal{T}(O) &= \mathbf{t} + O \\ &= t_x\mathbf{i} + t_y\mathbf{j} + t_z\mathbf{k} + O\end{aligned}$$

For a given basis vector, we can find two points P and Q that define the vector and compute the transformation of their difference. For example, for $\mathbf{i}$:

$$\begin{aligned}\mathcal{T}(\mathbf{i}) &= \mathcal{T}(P - Q) \\ &= \mathcal{T}(P) - \mathcal{T}(Q) \\ &= (\mathbf{t} + P) - (\mathbf{t} + Q) \\ &= P - Q \\ &= \mathbf{i}\end{aligned}$$

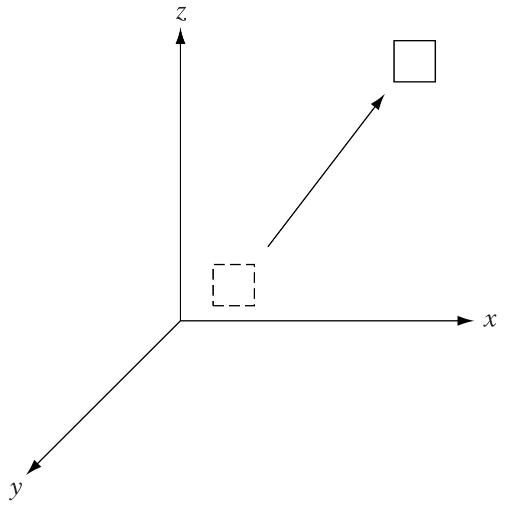


FIGURE 3.1 Translation.

The same holds true for **j** and **k**, so translation has no effect on the basis vectors in our frame. We end up with a 4 × 4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Or, in block form:

$$\mathbf{T_t} = \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Translation only affects points. To see why, suppose we have a vector **v**, which equals the displacement between two points P and Q, that is, $\mathbf{v} = P - Q$. If we translate $P - Q$, we get

$$\begin{aligned} \text{trans}(P - Q) &= (P + \mathbf{t}) - (Q + \mathbf{t}) \\ &= (P - Q) + (\mathbf{t} - \mathbf{t}) \\ &= \mathbf{v} \end{aligned}$$

This fits with our geometric notion that points have position and hence can be translated in space, while vectors do not and cannot.

We can use equation 3.3 to compute the inverse translation transformation:

$$\mathbf{T}_{\mathbf{t}}^{-1} = \begin{bmatrix} \mathbf{I}^{-1} & -\mathbf{I}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.4}$$

$$= \begin{bmatrix} \mathbf{I} & -\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.5}$$

$$= \mathbf{T}_{-\mathbf{t}} \tag{3.6}$$

So the inverse of a given translation negates the original translation vector to displace the point back to its original position.

3.3.2 Rotation

The other common rigid transformation is *rotation*. If we consider the rotation of a vector, we are rigidly changing its direction around an axis without changing its length. In $\mathbb{R}^2$, this is the same as replacing a vector with the one that's θ degrees counterclockwise (Figure 3.2).

In $\mathbb{R}^3$, we usually talk about an *axis of rotation*. In his rotation theorem, Euler showed that when applying a rotation in three-dimensional space, there is a linear set of points (i.e., a line) which does not change. This is called the axis of rotation, and the amount we rotate around this axis is the *angle of rotation*. A helpful mnemonic is the right-hand rule: if you point your right thumb in the direction of the axis vector, the curl of your fingers represents the direction of positive rotation (Figure 3.3).

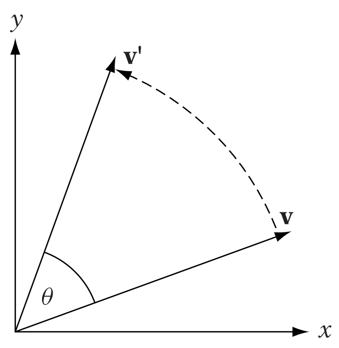


Figure 3.2 Rotation of vector in R^2.

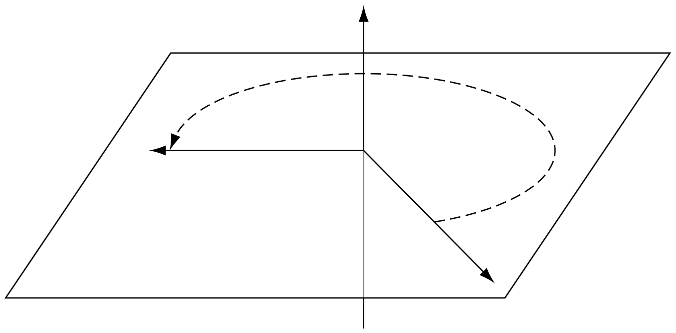


FIGURE 3.3 Axis and plane of rotation.

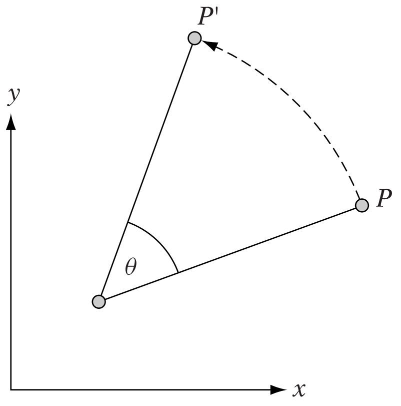


FIGURE 3.4 Rotation of point in R^2.

For a given point, we rotate it by moving it along a planar arc a constant distance from another point, known as the center of rotation (Figure 3.4). This center of rotation is commonly defined as the origin of the current frame (we'll refer to this as a *pure rotation*) but can be any arbitrary point. We can think of this as defining a vector **v** from the center of rotation to the point to be rotated, rotating **v**, and then adding the result to the center of rotation to compute the new position of the point. For now we'll only cover pure rotations; applying general affine transformations about an arbitrary center will be discussed later.

To keep things simple, we'll begin with rotations around one of the three frame axes, with a center of rotation equal to the origin. The following system of equations rotates a vector or point counterclockwise (assuming the axis is pointing at us) around **k**, or the *z*-axis (Figure 3.5c):

$$
\begin{aligned}
x' &= x\cos\theta - y\sin\theta \\
y' &= x\sin\theta + y\cos\theta \\
z' &= z
\end{aligned}
\tag{3.7}
$$

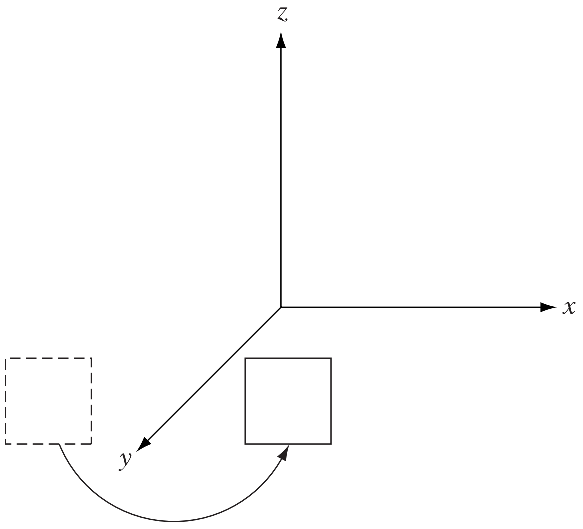


FIGURE 3.5a *x*-axis rotation.

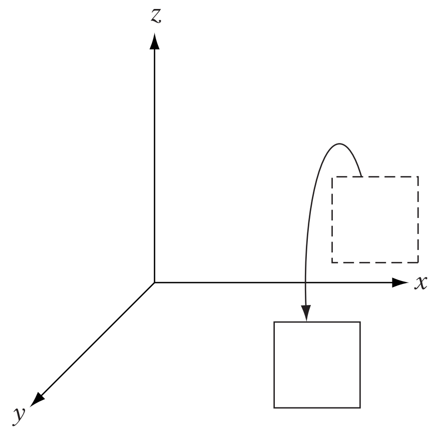


FIGURE 3.5b *y*-axis rotation.

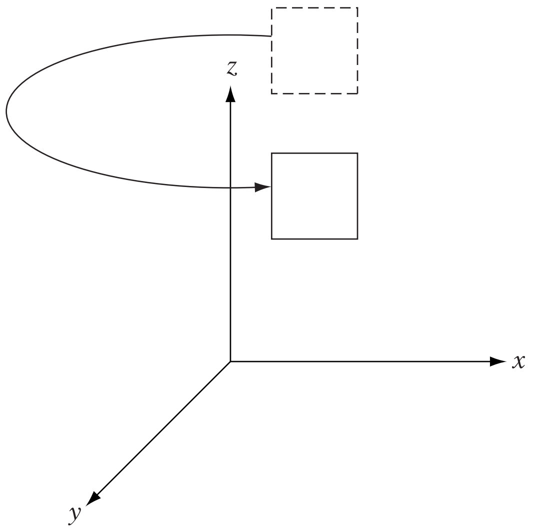


FIGURE 3.5C *z*-axis rotation.

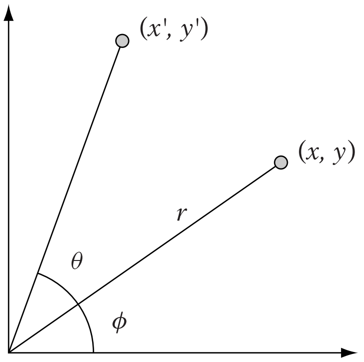


FIGURE 3.6 Rotation in *xy*-plane.

Figure 3.6 shows why this works. Since we're rotating around the z-axis, no z values will change, so we will consider only how the rotation affects the xy values of the points. The starting position of the point is (x, y), and we want to rotate that θ degrees counterclockwise. Handling this in Cartesian coordinates can be problematic, but this is one case where polar coordinates are useful.

Recall that a point P in polar coordinates has representation (r, ϕ), where r is the distance from the origin and ϕ[1] is the counterclockwise angle from

1. We're using ϕ for polar coordinates in this case to distinguish it from the rotation angle θ.

the x-axis. We can think of this as rotating an r length radius lying along the x-axis by ϕ degrees. If we rotate this a further θ degrees, the end of the radius will be at $(r, \phi+\theta)$ (in polar coordinates). Converting to Cartesian coordinates, the final point will lie at

$$\begin{aligned} x' &= r\cos(\phi+\theta) \\ y' &= r\sin(\phi+\theta) \end{aligned}$$

Using trigonometric identities, this becomes

$$\begin{aligned} x' &= r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ y' &= r\cos\phi\sin\theta + r\sin\phi\cos\theta \end{aligned}$$

But $r\cos\phi = x$, and $r\sin\phi = y$, so we can substitute and get

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}$$

We can derive similar equations for rotation around the x-axis (Figure 3.5a):

$$\begin{aligned} x' &= x \\ y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta \end{aligned}$$

and rotation around the y-axis (Figure 3.5b):

$$\begin{aligned} x' &= z\sin\theta + x\cos\theta \\ y' &= y \\ z' &= z\cos\theta - x\sin\theta \end{aligned}$$

To create the corresponding transformation, we need to determine how the frame elements are transformed. The frame's origin will not change since it's our center of rotation, so $\mathbf{y} = \mathbf{0}$. So our primary concern will be the contents of the 3×3 matrix $\mathbf{A}$.

For this matrix, we need to compute where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ will go. For example, for rotations around the z-axis we can transform $\mathbf{i}$ to get

$$\begin{aligned} x' &= (1)\cos\theta - (0)\sin\theta = \cos\theta \\ y' &= (1)\sin\theta + (0)\cos\theta = \sin\theta \\ z' &= 0 \end{aligned}$$

Transforming **j** and **k** similarly and copying the results into the columns of a 3×3 matrix gives

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Similar matrices can be created for rotation around the x-axis:

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

and around the y-axis:

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

One thing to note about these matrices is that their determinants are equal to 1, and they are all orthogonal. For example, look at the component 3-vectors of the z-axis rotation matrix. We have $(\cos\theta, \sin\theta, 0)$, $(-\sin\theta, \cos\theta, 0)$, and $(0, 0, 1)$. The first two lie on the xy-plane and so are perpendicular to the third, and they are perpendicular to each other. All three are unit length and so form an orthonormal basis.

The product of two orthogonal matrices is also an orthogonal matrix, thus the product of a series of pure rotation matrices is also a rotation matrix. For example, by concatenating matrices which rotate around the z-axis, then the y-axis, and then the x-axis, we can create one form of a generalized rotation matrix:

$$\mathbf{R}_x\mathbf{R}_y\mathbf{R}_z = \begin{bmatrix} CyCz & -CySz & Sy \\ SxSyCz + CxSz & -SxSySz + CxCz & -SxCy \\ -CxSyCz + SxSz & CxSySz + SxCz & CxCy \end{bmatrix} \tag{3.8}$$

where

$$\begin{aligned} Cx &= \cos\theta_x & Sx &= \sin\theta_x \\ Cy &= \cos\theta_y & Sy &= \sin\theta_y \\ Cz &= \cos\theta_z & Sz &= \sin\theta_z \end{aligned}$$

Recall that the inverse of an orthogonal matrix is its transpose. Because pure rotation matrices are orthogonal, the inverse of any rotation matrix is also its transpose. Therefore, the inverse of the z-axis rotation, centered on the origin, is

$$\mathbf{R}_z^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This follows if we think of the inverse transformation as "undoing" the original transformation. If you substitute $-\theta$ for θ in the original matrix and replace $\cos(-\theta)$ with $\cos\theta$ and $\sin(-\theta)$ with $-\sin\theta$, then we have:

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which, as we can see, results in the immediately preceding inverse matrix.

Now that we have looked at rotations around the coordinate axes, we will consider rotations about an arbitrary axis. The formula for a rotation of a vector $\mathbf{v}$ by an angle θ around a general axis $\hat{\mathbf{r}}$ is derived as follows. We begin by breaking $\mathbf{v}$ into two parts: the part parallel with $\hat{\mathbf{r}}$ and the part perpendicular to it, which lies on the plane of rotation (Figure 3.7a). Recall from Chapter 1 that the parallel part $\mathbf{v}_{\parallel}$ is the projection of $\mathbf{v}$ onto $\hat{\mathbf{r}}$, or

$$\mathbf{v}_{\parallel} = (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \tag{3.9}$$

The perpendicular part is what remains of $\mathbf{v}$ after we subtract the parallel part, or

$$\mathbf{v}_{\perp} = \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \tag{3.10}$$

To properly compute the effect of rotation, we need to create a two-dimensional basis on the plane of rotation (Figure 3.7b). We'll use $\mathbf{v}_{\perp}$ as our first basis vector, and we'll need a vector $\mathbf{w}$ perpendicular to it for our second basis vector. We can take the cross product with $\hat{\mathbf{r}}$ for this:

$$\mathbf{w} = \hat{\mathbf{r}} \times \mathbf{v}_{\perp} = \hat{\mathbf{r}} \times \mathbf{v} \tag{3.11}$$

In the standard basis for $\mathbb{R}^2$, if we rotate the vector $\mathbf{i} = (1, 0)$ by θ, we get the vector $(\cos\theta, \sin\theta)$. Equivalently,

$$R\mathbf{i} = (\cos\theta)\mathbf{i} + (\sin\theta)\mathbf{j}$$

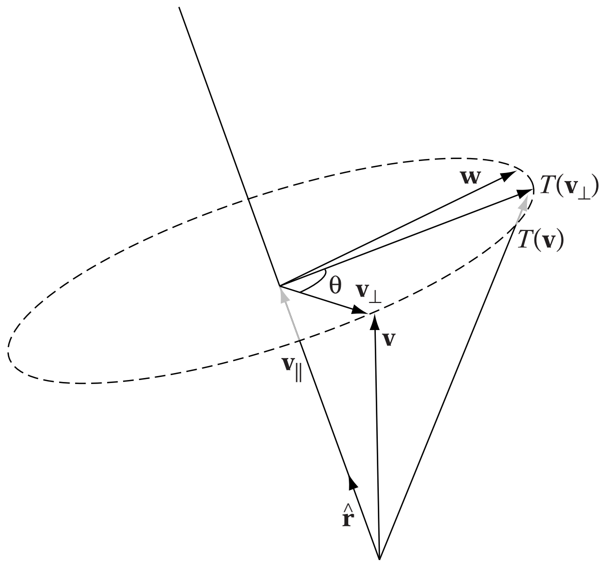


Figure 3.7a General rotation, showing axis of rotation and rotation plane.

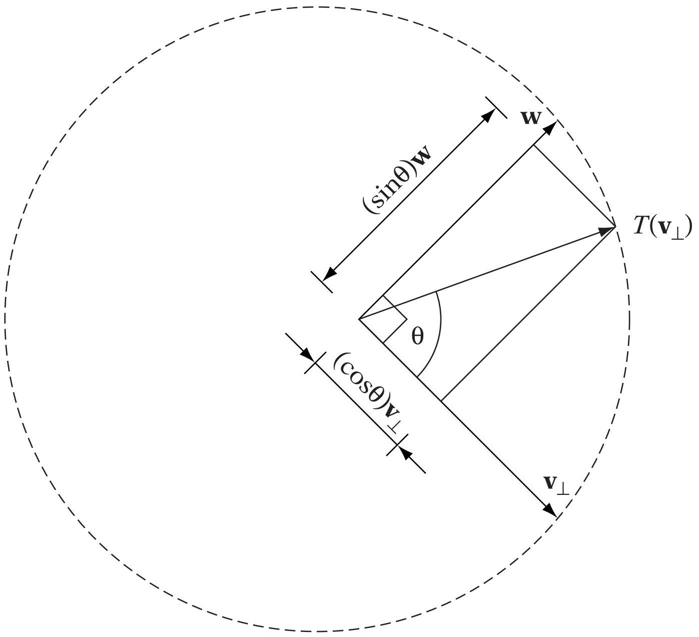


Figure 3.7b General rotation, showing vectors on rotation plane.

If we use $\mathbf{v}_\perp$ and $\mathbf{w}$ as the 2D basis for the rotation plane, we can find the rotation of $\mathbf{v}_\perp$ by θ in a similar manner:

$$R\mathbf{v}_\perp = (\cos\theta)\mathbf{v}_\perp + (\sin\theta)\mathbf{w} \tag{3.12}$$

The parallel part of $\mathbf{v}$ doesn't change with the rotation, so the final result of rotating $\mathbf{v}$ around $\hat{\mathbf{r}}$ by θ is

$$\begin{aligned}
R\mathbf{v} &= R\mathbf{v}_\| + R\mathbf{v}_\perp \\
&= R\mathbf{v}_\| + (\cos\theta)\mathbf{v}_\perp + (\sin\theta)\mathbf{w} \\
&= (\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} + \cos\theta[\mathbf{v} - (\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}}] + \sin\theta(\hat{\mathbf{r}}\times\mathbf{v}) \\
&= \cos\theta\mathbf{v} + [1-\cos\theta](\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}}\times\mathbf{v})
\end{aligned} \tag{3.13}$$

This is one form of what is known as the *Rodrigues formula*.

The projection $(\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}}$ can be replaced by the tensor product $(\hat{\mathbf{r}}\otimes\hat{\mathbf{r}})\mathbf{v}$. Similarly, the cross product $\hat{\mathbf{r}}\times\mathbf{v}$ can be replaced by a multiplication by a skew symmetric matrix $\tilde{\mathbf{r}}\mathbf{v}$. This gives

$$\begin{aligned}
R\mathbf{v} &= \cos\theta\mathbf{v} + (1-\cos\theta)(\hat{\mathbf{r}}\otimes\hat{\mathbf{r}})\mathbf{v} + \sin\theta\tilde{\mathbf{r}}\mathbf{v} \\
&= [\cos\theta\mathbf{I} + (1-\cos\theta)(\hat{\mathbf{r}}\otimes\hat{\mathbf{r}}) + \sin\theta\tilde{\mathbf{r}}]\mathbf{v}
\end{aligned}$$

Expanding the terms, we end up with a matrix:

$$\mathbf{R}_{\hat{\mathbf{r}}\theta} = \begin{bmatrix} tx^2+c & txy-sz & txz+sy \\ txy+sz & ty^2+c & tyz-sx \\ txz-sy & tyz+sx & tz^2+c \end{bmatrix}$$

where

$$\begin{aligned}
\hat{\mathbf{r}} &= (x, y, z) \\
c &= \cos\theta \\
s &= \sin\theta \\
t &= 1-\cos\theta
\end{aligned}$$

As we can see, there is a wide variety of choices for the 3×3 matrix $\mathbf{A}$, depending on what sort of rotation we wish to perform. The full affine matrix for rotation around the origin is

$$\begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where **R** is one of the rotation matrices just given. For example, the affine matrix for rotation around the x-axis is

$$\begin{bmatrix} \mathbf{R}_x & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is also an orthogonal matrix and its inverse is the transpose, as before.

Finally, when discussing rotations one has to be careful to distinguish rotation from orientation, which is to rotation as position is to translation. If we consider the representation of a point in an affine space:

$$P = \mathbf{v} + O$$

then we can think of the origin as a reference position and the vector **v** as a translation which relates our position to the reference. We can represent our position as just the components of the translation. Similarly, we can define a reference orientation Ω_0, and any orientation Ω is related to it by a rotation, or

$$\Omega = \mathbf{R}_0\Omega_0$$

Just as we might use the components of the vector **v** to represent our position, we can use the rotation $\mathbf{R}_0$ to represent our orientation. To change our orientation, we apply an additional rotation just as we might add a translation vector to change our position:

$$\Omega' = \mathbf{R}_1\Omega$$

In this case our final orientation, using the rotation component, is

$$\mathbf{R}_1\mathbf{R}_0$$

Remember that the order of concatenation matters, because matrix multiplication—particularly for rotation matrices—is not a commutative operation.

3.3.3 Scaling

The remaining affine transformations that we will cover are *deformations*, since they don't preserve exact lengths or angles. The first is *scaling*, which can be thought of as corresponding to our other basic vector operation, scalar

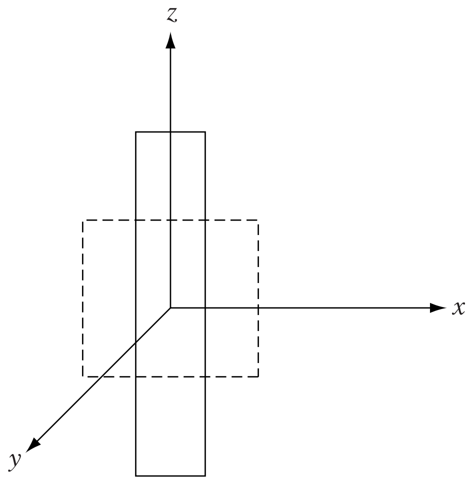


Figure 3.8 Nonuniform scaling.

multiplication; however, it is not quite the same. Scalar multiplication of a vector has only one multiplicative factor and changes a vector's length equally in all directions. We can also multiply a vector by a negative scalar. In comparison, scaling as it is commonly used in computer graphics applies a possibly different but positive factor to each basis vector in our frame.[2] If all the factors are equal, then it is called uniform scaling and is—for vectors in the affine space—equivalent to scalar multiplication by a single positive scalar. Otherwise, it is called nonuniform scaling. Full nonuniform scaling can be applied differently in each axis direction, so we can scale by 2 in z to make an object twice as tall, but 1/2 in x and y to make it half as wide.

A point doesn't have a length per se, so instead we change its relative distance from another point C_s, known as the center of scaling. We can consider this as scaling the vector from the center of scaling to our point P. For a set of points, this will end up scaling their distance relative to each other, but still maintaining the same relative shape (Figure 3.8).

For now we'll consider only scaling around the origin, so $C_s = O$ and $\mathbf{y} = \mathbf{0}$. For the upper 3×3 matrix $\mathbf{A}$, we again need to determine how the frame basis vectors change, which is defined as

$$\mathcal{T}(\mathbf{i}) = a\mathbf{i}$$
$$\mathcal{T}(\mathbf{j}) = b\mathbf{j}$$
$$\mathcal{T}(\mathbf{k}) = c\mathbf{k}$$

2. We'll consider negative factors when we discuss reflections in the following section.

where $a, b, c > 0$ and are the scale factors in the x, y, z directions, respectively. Writing these transformed basis vectors as the columns of **A**, we get an affine matrix of

$$\mathbf{S}_{abc} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is a diagonal matrix, with the positive scale factors lying along the diagonal, so the inverse is

$$\mathbf{S}_{abc}^{-1} = \mathbf{S}_{\frac{1}{a}\frac{1}{b}\frac{1}{c}} = \begin{bmatrix} 1/a & 0 & 0 & 0 \\ 0 & 1/b & 0 & 0 \\ 0 & 0 & 1/c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.3.4 Reflection

The reflection transformation symmetrically maps an object across a plane or through a point. One possible reflection is (Figure 3.9a):

$$\begin{aligned} x' &= -x \\ y' &= y \\ z' &= z \end{aligned}$$

This reflects across the yz plane and gives an effect like a standard mirror (mirrors don't swap left to right, they swap front to back). If we want to reflect across the xz-plane instead, we would use (Figure 3.9b)

$$\begin{aligned} x' &= x \\ y' &= -y \\ z' &= z \end{aligned}$$

As one might expect, we can create a planar reflection that reflects across a general plane, defined by a normal $\hat{\mathbf{n}}$ and a point on the plane P_0. For now we'll consider only planes that pass through the origin. If we have a vector $\mathbf{v}$ in our affine space, we can break it into two parts: the part coincident to the plane $\mathbf{v}_\perp$, which will remain unchanged, and the part orthogonal to it $\mathbf{v}_\parallel$, which will be reflected to the other side of the plane to become $-\mathbf{v}_\parallel$. The transformed vector will be the sum of $\mathbf{v}_\perp$ and the reflected $-\mathbf{v}_\parallel$ (Figure 3.10).

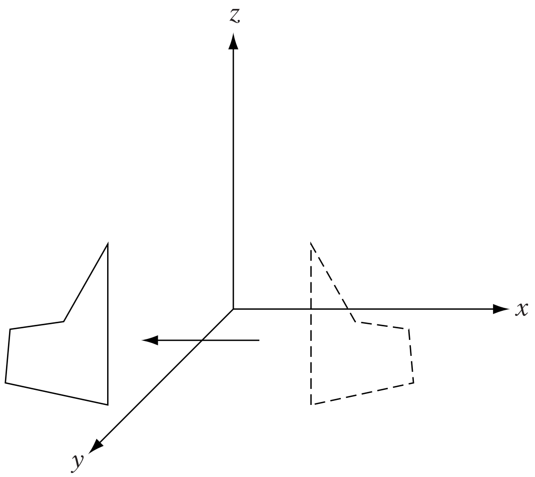


Figure 3.9a *yz* reflection.

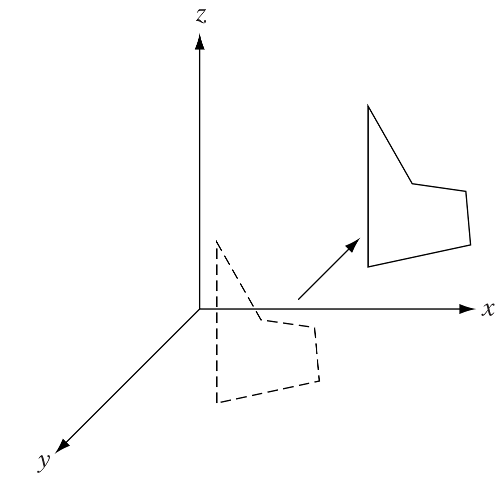


Figure 3.9b *xz* reflection.

To compute $\mathbf{v}_\parallel$, we merely have to take the projection of $\mathbf{v}$ against the plane normal $\hat{\mathbf{n}}$, or

$$\mathbf{v}_\parallel = (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{3.14}$$

Subtracting this from $\mathbf{v}$, we can compute $\mathbf{v}_\perp$:

$$\mathbf{v}_\perp = \mathbf{v} - \mathbf{v}_\parallel \tag{3.15}$$

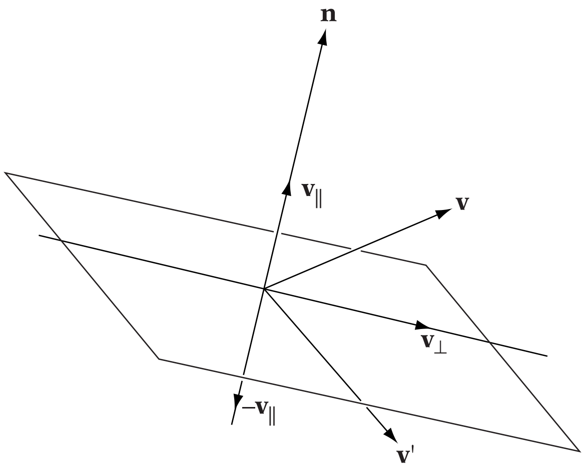


Figure 3.10 General reflection.

We know that the transformed vector will be $\mathbf{v}_{\perp} - \mathbf{v}_{\|}$. Substituting equations 3.15 and 3.14 into this gives us

$$\begin{aligned}\mathcal{T}(\mathbf{v}) &= \mathbf{v}_{\perp} - \mathbf{v}_{\|} \\ &= \mathbf{v} - 2\mathbf{v}_{\|} \\ &= \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}\end{aligned}$$

From Chapter 2, we know that we can perform the projection of $\mathbf{v}$ on $\hat{\mathbf{n}}$ by multiplying by the tensor product matrix $\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}$, so this becomes

$$\begin{aligned}\mathcal{T}(\mathbf{v}) &= \mathbf{v} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})\mathbf{v} \\ &= [\mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})]\mathbf{v}\end{aligned}$$

Thus, the linear transformation part $\mathbf{A}$ of our affine transformation is $[\mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})]$. Writing this as a block matrix:

$$\mathbf{F}_{\mathbf{n}} = \begin{bmatrix} \mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}) & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

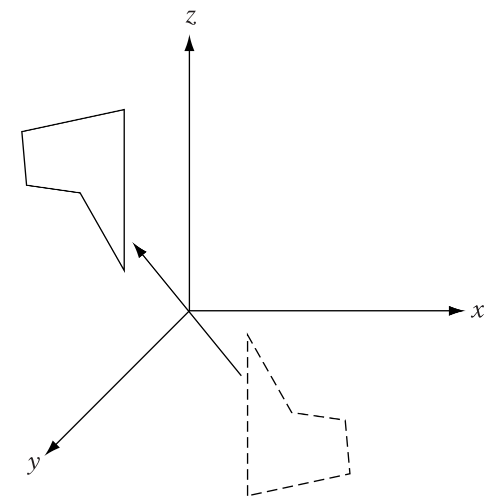


FIGURE 3.11 Point reflection.

While in the real world we usually see planar reflections, in our virtual world we can also compute a reflection through a point. The following performs a reflection through the origin (Figure 3.11):

$$x' = -x$$
$$y' = -y$$
$$z' = -z$$

The corresponding block matrix is

$$\mathbf{F}_O = \begin{bmatrix} -\mathbf{I} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Reflections are a symmetric operation—that is, the reflection of a reflection returns the original point or vector. Because of this, the inverse of a reflection matrix is the matrix itself.

As an aside, we would (incorrectly) expect that if we can reflect through a plane and a point, we can reflect through a line. The following system:

$$x' = -x$$
$$y' = -y$$
$$z' = z$$

appears to reflect through the z axis, giving a "funhouse mirror" effect, where right and left are swapped (if y is left, it becomes $-y$ in the "reflection" and so ends up on the right side). However, if we examine the transformation closely, we see that while it does perform the desired effect, this is actually a rotation of 180 degrees around the z-axis. While both pure rotations and pure reflections through the origin are orthogonal matrices, we can distinguish between them by noting that reflection matrices have a determinant of -1, while rotation matrices have a determinant of 1.

3.3.5 Shear

The final affine transformation that we will cover is shear. Because it affects the angles of objects it is not used all that often, but it comes up particularly when discussing oblique projections. An axis-aligned shear provides a shift in one or two axes proportional to the component in a third axis. Transforming a square to a rhombus or a cube to a rhomboid solid is a shear transformation (Figure 3.12).

There are a number of ways of specifying shear ([79], [96]). In our case we will define a shear plane, with normal $\hat{\mathbf{n}}$, that does not change due to the transformation. We define an orthogonal shear vector $\mathbf{s}$, which indicates how planes parallel to the shear plane will be transformed. Points on the plane 1 unit of distance from the shear plane, in the direction of the plane normal, will be displaced by $\mathbf{s}$. Points on the plane 2 units from the shear plane will

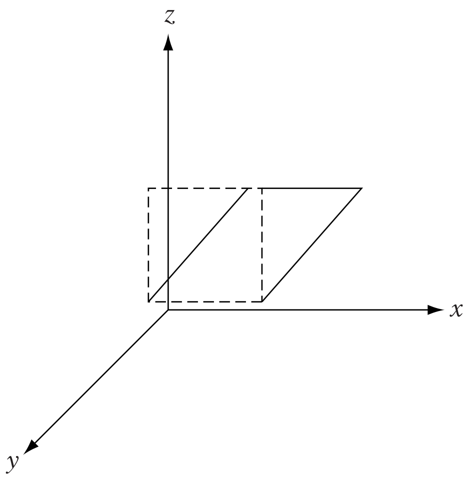


Figure 3.12 z-shear on square.

be displaced by 2**s**, and so on. In general, if we take a point P and define it as $P_0 + \mathbf{v}$, where P_0 is a point on the shear plane, then P will be displaced by $(\hat{\mathbf{n}} \cdot \mathbf{v})\mathbf{s}$.

The simplest case is when we apply shear perpendicular to one of the main coordinate axes. For example, if we take the yz-plane as our shear plane, our normal is **i** and the shear plane passes through the origin O. We know from this that O will not change with the transformation, so our vector **y** is **0**. As before, to find **A** we need to figure out how the transformation affects our basis vectors. If we define **j** as $P_1 - O$, then

$$\mathcal{T}(\mathbf{j}) = \mathcal{T}(P_1) - \mathcal{T}(O)$$

But P_1 and O lie on the shear plane, so

$$\begin{aligned}\mathcal{T}(\mathbf{j}) &= P_1 - O \\ &= \mathbf{j}\end{aligned}$$

The same is true for the basis vector **k**. For **i**, we can define it as $P_0 - O$. We know that P_0 is distance 1 from the shear plane, so it will become $P_0 + \mathbf{s}$, so

$$\begin{aligned}\mathcal{T}(\mathbf{i}) &= \mathcal{T}(P_0) - \mathcal{T}(O) \\ &= P_0 + \mathbf{s} - O \\ &= \mathbf{i} + \mathbf{s}\end{aligned}$$

The vector **s** in this case is orthogonal to **i**, therefore it is of the form $(0, a, b)$, so our transformed basis vector will be $(1, a, b)$. Our final matrix **A** is

$$\mathbf{H}_x = \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$$

We can go through a similar process to get shear by the y-axis:

$$\mathbf{H}_y = \begin{bmatrix} 1 & c & 0 \\ 0 & 1 & 0 \\ 0 & d & 1 \end{bmatrix}$$

and shear by the z-axis:

$$\mathbf{H}_z = \begin{bmatrix} 1 & 0 & e \\ 0 & 1 & f \\ 0 & 0 & 1 \end{bmatrix}$$

For shearing by a general plane through the origin, we already have the formula for the displacement: $(\hat{\mathbf{n}} \cdot \mathbf{v})\mathbf{s}$. We can rewrite this as a tensor product to get $(\hat{\mathbf{n}} \otimes \mathbf{s})\mathbf{v}$. Because this is merely the displacement, we need to include the original point, and thus our origin-centered general shear matrix is simply $\mathbf{I} + \hat{\mathbf{n}} \otimes \mathbf{s}$. Our final shear matrix is

$$\mathbf{H}_{\hat{\mathbf{n}},\mathbf{s}} = \begin{bmatrix} \mathbf{I} + \mathbf{s} \otimes \hat{\mathbf{n}} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

The inverse shear transformation is shear in the opposite direction, so the corresponding matrix is

$$\mathbf{H}^{-1}_{\hat{\mathbf{n}},\mathbf{s}} = \begin{bmatrix} \mathbf{I} - \mathbf{s} \otimes \hat{\mathbf{n}} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{H}_{\hat{\mathbf{n}},-\mathbf{s}}$$

3.3.6 Applying an Affine Transformation Around an Arbitrary Point

Up to this point, we have been assuming that our affine transformations are applied around the origin of the frame. For example, when discussing rotation we treated the origin as our center of rotation. Similarly, our shear planes were assumed to pass through the origin. This doesn't necessarily have to be the case.

Let's look at a particular example—the rotation of a point around an arbitrary center of rotation C—and determine how this transformation affects the origin of our frame. If we look at Figure 3.13, we see the situation. We have a point C and our origin O. We want to rotate the difference vector

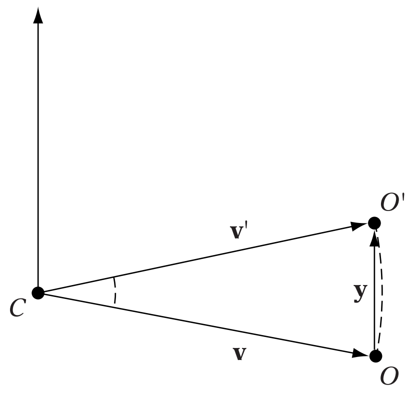


Figure 3.13 Rotation of origin around arbitrary center.

$\mathbf{v} = O - C$ between the two points by matrix $\mathbf{R}$ and determine where the resulting point $\mathcal{T}(O)$, or $C + \mathcal{T}(\mathbf{v})$, will be. From that we can compute the difference vector $\mathbf{y} = \mathcal{T}(O) - O$. From Figure 3.13, we can see that $\mathbf{y} = \mathcal{T}(\mathbf{v}) - \mathbf{v}$, so we can reduce this as follows:

$$\begin{aligned}\mathbf{y} &= \mathcal{T}(\mathbf{v}) - \mathbf{v} \\ &= \mathbf{Rv} - \mathbf{v} \\ &= (\mathbf{R} - \mathbf{I})\mathbf{v}\end{aligned}$$

It's usually more convenient to write this in terms of the vector dual to C, which is $\mathbf{x} = C - O = -\mathbf{v}$, so this becomes

$$\begin{aligned}\mathbf{y} &= -(\mathbf{R} - \mathbf{I})\mathbf{x} \\ &= (\mathbf{I} - \mathbf{R})\mathbf{x}\end{aligned}$$

We can achieve the same result by translating our center C to the frame origin by $-\mathbf{x}$, performing our origin-centered rotation, and then translating back by $\mathbf{x}$:

$$\begin{aligned}\mathbf{M}_c &= \begin{bmatrix} \mathbf{I} & \mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & -\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R} & \mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & -\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R} & (\mathbf{I}-\mathbf{R})\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\end{aligned}$$

Notice that the upper left-hand block $\mathbf{R}$ is not affected by this process.

The same construction can be used for all affine transformations that use a center of transformation: rotation, scale, reflection, and shear. The exception is translation, since such an operation has no effect: $P - \mathbf{x} + \mathbf{t} + \mathbf{x} = P + \mathbf{t}$. But for the others, using a point $C = (\mathbf{x}, 1)$ as our arbitrary center of transformation gives

$$\mathbf{M}_c = \begin{bmatrix} \mathbf{A} & (\mathbf{I} - \mathbf{A})\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{A}$ is the upper 3×3 matrix of an origin-centered transformation. The corresponding inverse is

$$\mathbf{M}_c^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & (\mathbf{I} - \mathbf{A}^{-1})\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

3.3.7 Transforming Plane Normals

As we saw in the previous section, if we want to transform a line or plane represented in parametric form, we transform the points in the affine combination. For example,

$$\mathcal{T}(P(t)) = (1 - s - t)\mathcal{T}(P_0) + s\mathcal{T}(P_1) + t\mathcal{T}(P_2)$$

But suppose we have a plane represented using the generalized plane equation. One way of considering this is as a plane normal (a, b, c) and a point on the plane P_0. We could transform these and try to use the resulting vector and point to build the new plane. However, if we apply an affine transform to the plane normal (a, b, c) directly, we may end up performing a deformation. Since angles aren't preserved under deformations, the resulting "normal" may no longer be orthogonal to the points in the plane.

The correct approach is as follows. We can represent the generalized plane equation as the product of a row matrix and column matrix, or

$$\begin{aligned} ax + by + cz + d &= \begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ &= \mathbf{n}^T P \end{aligned}$$

Now P is clearly a point, and $\mathbf{n}$ is the vector of coefficients for the plane. For points that lie on the plane:

$$\mathbf{n}^T P = 0$$

If we transform all the points on the plane by some matrix $\mathbf{M}$, then to maintain the relationship between $\mathbf{n}^T$ and P, we'll have to transform $\mathbf{n}$ by some unknown matrix $\mathbf{Q}$, or

$$(\mathbf{Qn})^T(\mathbf{M}P) = 0$$

This can be rewritten as

$$\mathbf{n}^T\mathbf{Q}^T\mathbf{M}P = 0$$

One possible solution for this is if

$$\mathbf{I} = \mathbf{Q}^T\mathbf{M}$$

Solving for **Q** gives

$$\mathbf{Q} = \left(\mathbf{M}^{-1}\right)^T$$

So the transformed plane coefficients become

$$\mathbf{n}' = \left(\mathbf{M}^{-1}\right)^T \mathbf{n}$$

The same approach will work if we're transforming the plane normal and point as described earlier. We transform the point P_0 by **M** and the normal by $(\mathbf{M}^{-1})^T$.

In many cases the inverse matrix $\mathbf{M}^{-1}$ may not exist. So if we're just transforming a normal vector (a, b, c), we can use a different method. Instead of $\mathbf{M}^{-1}$, we use the adjoint matrix from Cramer's rule. Normally we couldn't proceed at this point: if the inverse doesn't exist, we end up dividing by a zero determinant. However, even when the inverse exists, the division by the determinant is a scale factor. So we can ignore it in all cases and just use the adjoint matrix directly, because we're going to normalize the resulting vector anyway.

3.4 Using Affine Transformations

3.4.1 Manipulation of Game Objects

The primary use of affine transformations is for the manipulation of objects in our game world. Suppose, from our earlier hypothetical, we have an office environment that is acting as our game space. The artists could build the basic level — the walls, the floor, the ceilings, and so forth — as a single set of triangles with coordinates defined to place them exactly where we might want them in the world. However, suppose we have a single desk model that we want to duplicate and place in various locations in the level. The artist could build a new version of the desk for each location in the core level geometry, but that would involve unnecessarily duplicating all the memory needed for the model. Instead, we could have one version, or *master*, of the desk model and then set a series of transformations that indicate where in the level each copy, or *instance*, of the desk should be placed [106].

Before we can begin to discuss how we specify these transformations and what they might mean, we need to define the two different coordinate frames we are working in: the local coordinate frame and the world coordinate frame.

Local and World Coordinate Frames

When artists create an object or we create an object directly in a program, the coordinates of the points that make up that object are defined in that particular object's *local frame*. This is also commonly known as *local space*, or alternatively as *model space* or *object space*.

The orientation of the basis vectors in the local frame is usually set so that the engineers know which part of the object is the front, which is the top, and which is the side. This allows us to orient the object correctly relative to the rest of the world and to translate it in the correct direction if we want to move it forward. The convention that we will be using in this book is one where the x-axis points along the forward direction of the object, the y-axis points towards the left of the object, and the z-axis points out the top of the object (Figure 3.14). Another common convention is to use the y-axis for up, the z-axis for forward, and the x-axis for either out to the left or to the right, depending on whether we want to work in a right-handed or left-handed frame.

Typically, the origin of the frame is placed in a position convenient for the game, either at the center of the object or at the bottom of the object. The first is useful when we want to rotate objects around their centers, the second for placement on the ground.

When constructing our world, we define a specific coordinate frame, or *world frame*, also known as *world space*. The world frame acts as a common reference among all the objects, much as the origin acts as a common reference among points. Ultimately, in order to render, simulate, or otherwise interact with objects, we will need to transform their local coordinates into the world frame.

When an artist builds the level geometry, the coordinates are usually set in the world frame. Orientation of the level relative to our world frame is set

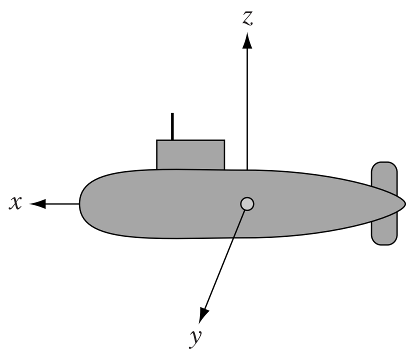


FIGURE 3.14 Local object frame.

by convention. Knowing which direction is “up” is important in a 3D game; in our case we’ll be using the z-axis, but the y-axis is also commonly used. Aligning the level to the other two axes (in our case, x and y) is arbitrary, but if our level is either gridlike or box-shaped, it is usually convenient to orient the grid lines or box sides to these remaining axes.

Positioning the level relative to the origin of the frame is also arbitrary but is usually set so that the origin lies in the center of a box defining our maximum play area. This helps avoid precision problems, since floating point precision is centered around 0 (see Chapter 4). For example, we might have a 300 meter by 300 meter play area, so that in the xy directions the origin will lie directly in the center. While we can set things so that the origin is centered in z as well, we may want to adjust that depending on our application. If our game mainly takes place on a flat play area, such as in an arena fighting game, we might set the floor so that it lies at the origin; this will make it simple to place objects and characters exactly at floor level. In a submarine game, we might place sea level at the origin; negative z lies under the waterline and positive z above.

Placing Objects

If we were to use the objects’ local coordinates directly in the world frame, they would end up interpenetrating and centered around the world origin. To avoid that situation, we apply affine transformations to each object to place them at their own specific position and orientation in the world. For each object, this is known as their particular *local-to-world transformation*. We often display the relative position and orientation of a particular object in the world by drawing its frame relative to the world frame (Figure 3.15). The local-to-world transformation, or world transformation for short, describes this relative relationship: the column vectors of the local-to-world matrix $\mathbf{A}$ describe where the local frame’s basis vectors will lie relative to the world space basis, and the vector $\mathbf{y}$ describes where the local frame’s origin lies relative to the world origin.

The most commonly used affine transformations for object placement are translation, rotation, and scaling. Translation and rotation are convenient for two reasons. First, they correspond naturally to two of the characteristics we want to control in our objects, position and orientation. Second, they are rigid transformations, meaning they don’t affect the size or shape of our object, which is generally the desired effect. Scaling is a deformation but is commonly useful to change the size of objects. For example, if two artists build two objects but fail to agree on a relative measure of size, you might end up with a table bigger than a room, if placed directly in the level. Rather than have the artist redo the model, we can use scaling to make it appear smaller. Scaling is also useful in fantastical games to either shrink a character to fit in a small space or grow a character to be more imposing. However, for most games you can actually get away without using scaling at all.

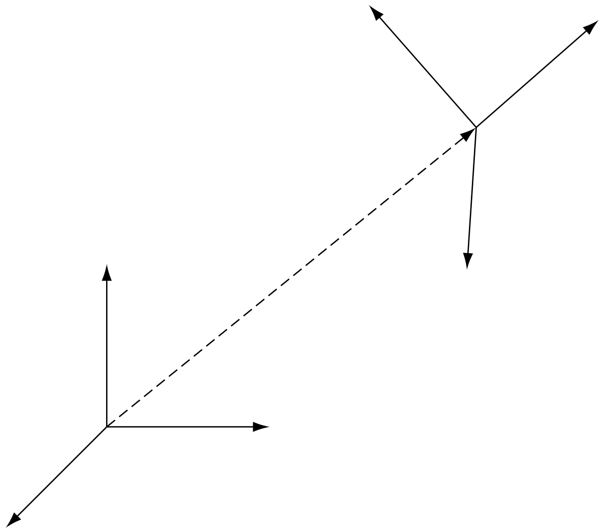


FIGURE 3.15 Local to world transformation.

Source Code
Demo
Interaction

To create the final world transformation, we'll be concatenating a sequence of these translation, rotation, and scaling transformations together. However, remember that concatenation of transformations is not commutative. So the order in which we apply our transformations affects the final result, sometimes in surprising ways. One basic example is transforming the point (0,0,0). A pure rotation around the origin has no effect on (0,0,0), so rotating by 90 degrees around z and then translating by (t_x, t_y, t_z) will just act as a translation, and we end up with (t_x, t_y, t_z). Translating the point first will transform it to (t_x, t_y, t_z), so in this case a subsequent rotation of 90 degrees around z will have an effect, with the final result of $(-t_y, t_x, t_z)$. As another example, look at Figure 3.16a, which shows a rotation and translation. Figure 3.16b shows the equivalent translation and rotation.

Scaling and rotation are also noncommutative. If we first scale (1,0,0) by (s_x, s_y, s_z), we get the point $(s_x, 0, 0)$. Rotating this by 90 degrees around z, we end up with $(0, s_x, 0)$. Reversing the transformation order, if we rotate (1,0,0) by 90 degrees around z, we get the point (0, 1, 0). Scaling this by (s_x, s_y, s_z), we get the point $(0, s_y, 0)$. Note that in the second case we rotated our object so that our original x-axis lies along the y-axis and then applied our scale, giving us the unexpected result. Figures 3.17a and 3.17b show another example of this applied to an object.

The final combination is scaling and translation. Again, this is not commutative. Remember that pure scaling is applied from the origin of the frame. If we translate an object from the origin and then scale, there will be

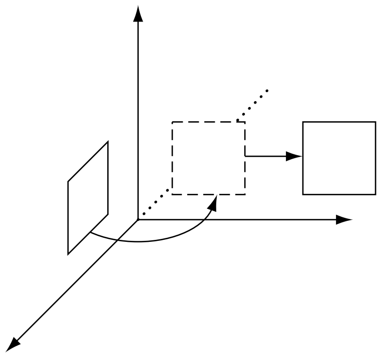


Figure 3.16a Rotation, then translation.

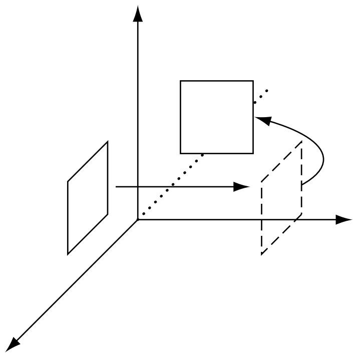


Figure 3.16b Translation, then rotation.

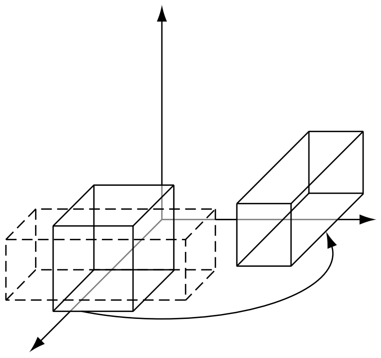


Figure 3.17a Scale, then rotation.

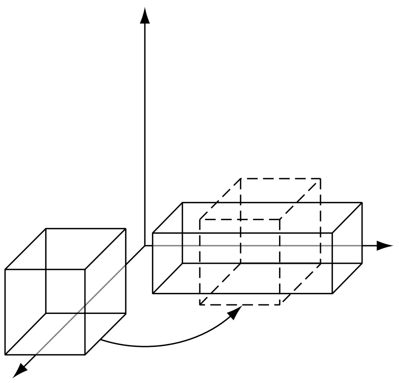


FIGURE 3.17b Rotation, then scale.

additional scaling done to the translation of the object. So for example, if we scale (1, 1, 1) by (s_x, s_y, s_z) and then translate by (t_x, t_y, t_z), we end up with $(t_x + s_x, t_y + s_y, t_z + s_z)$. If instead we translate first, we get $(t_x + 1, t_y + 1, t_z + 1)$, and then scaling gives us $(s_x t_x + s_x, s_y t_y + s_y, s_z t_z + s_z)$. Another example can be seen in Figures 3.18a and 3.18b.

Generally, the desired order we wish to use for these transforms is to scale first, then rotate, then translate. Scaling first gives us the scaling along the axes we expect. We can then rotate around the origin of the frame, and then translate it into place. This gives us the following multiplication order:

$$\mathbf{M} = \mathbf{TRS}$$

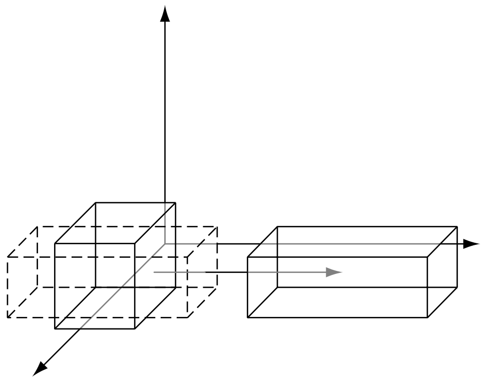


FIGURE 3.18a Scale, then translate.

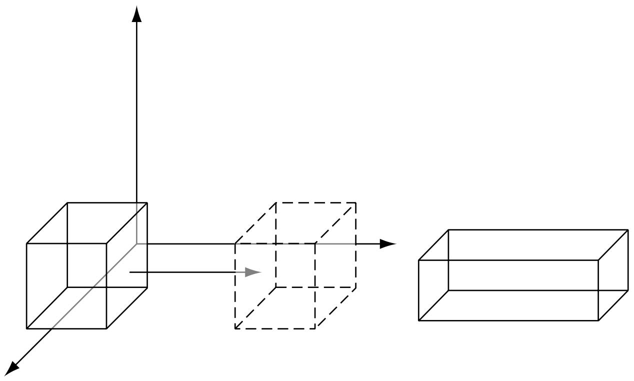


Figure 3.18b Translate, then scale.

3.4.2 Matrix Decomposition

It is sometimes useful to break an affine transformation matrix into its component basic affine transformations. This is called *matrix decomposition*. We performed one such decomposition when we pulled the translation information out of the matrix, effectively representing our transformation as the product of two matrices:

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Suppose we continue the process and break down **A** into the product of more basic affine transformations. For example, if we're using only scaling, rotation, and translation, it would be ideal if we could break **A** into the product of a scaling and rotation matrix. If we know for a fact that **A** is the product of only a scaling and rotation matrix, in the order **RS**, we can multiply it out to get

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x r_{11} & s_y r_{12} & s_z r_{13} & 0 \\ s_x r_{21} & s_y r_{22} & s_z r_{23} & 0 \\ s_x r_{31} & s_y r_{32} & s_z r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case the lengths of the first three column vectors will give our three scale factors s_x, s_y, and s_z. To get the rotation matrix, all we need to do is normalize those three vectors.

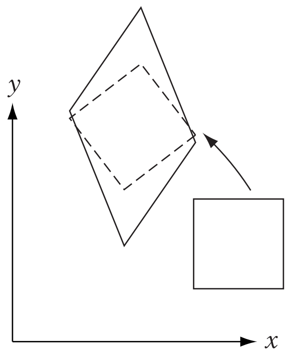


FIGURE 3.19 Effect of rotation, then scale.

Unfortunately, it isn't always that simple. As we'll see in Part 3.5, often we'll be concatenating a series of **TRS** transformations to get something like

$$\mathbf{M} = \mathbf{T}_n\mathbf{R}_n\mathbf{S}_n \cdots \mathbf{T}_1\mathbf{R}_1\mathbf{S}_1\mathbf{T}_0\mathbf{R}_0\mathbf{S}_0$$

In this case, even ignoring the translations, it is impossible to decompose **M** into the form **RS**. As a quick example, suppose that all these transformations with the exception of $\mathbf{S}_1$ and $\mathbf{R}_0$ are the identity transformation. This simplifies to

$$\mathbf{M} = \mathbf{S}_1\mathbf{R}_0$$

Now suppose $\mathbf{S}_1$ scales by 2 along y and by 1 along x and z, and $\mathbf{R}_0$ rotates by 60 degrees around z. Figure 3.19 shows how this affects a square on the xy plane. The sides of the transformed square are no longer perpendicular. Somehow, we have ended up applying a shear within our transformation, and clearly we cannot represent this by a simple concatenation **RS**.

One solution is to decompose the matrix using a technique known as *singular value decomposition*, or simply SVD. Assuming no translation, the matrix **M** can be represented by three matrices **L**, **D**, and **R**, where **L** and **R** are orthogonal matrices, **D** is a diagonal matrix with nonnegative entries, and

$$\mathbf{M} = \mathbf{LDR}$$

An alternative formulation to this is *polar decomposition*, which breaks the nontranslational part of the matrix into two pieces, an orthogonal matrix **Q** and a stretch matrix **S**, where

$$\mathbf{S} = \mathbf{U}^T\mathbf{KU}$$

Matrix **U** in this case is another orthogonal matrix, and **K** is a diagonal matrix. The stretch matrix combines the scale-plus-shear effect we saw in our example: it rotates the frame to an orientation, scales along the axes, and then rotates back. Using this, a general affine matrix can be broken into four transformations:

$$\mathbf{M} = \mathbf{TRNS}$$

where **T** is a translation matrix, **Q** has been separated into a rotation matrix **R** and a reflection matrix $\mathbf{N} = \pm\mathbf{I}$, and **S** is the preceding stretch matrix.

Performing either SVD or polar decomposition is out of the purview of this text. As we'll see, there are ways to avoid matrix decomposition at the cost of some conversion before we send our models down the graphics pipeline. However, at times we may get a matrix of unknown structure from a library module that we don't control. For example, we could be using a commercial physics engine or writing a plug-in for a 3D modeling package such as Max or Maya. Most of the time a function is provided that will decompose such matrices for us, but this isn't always the case. For those times and for those who are interested in pursuing this topic, more information on decompositions can be found in [45], [46], and [103].

3.4.3 Avoiding Matrix Decomposition

Source Code
Demo
Centered

In the preceding section, we made no assumptions about the values for our scaling factors. Now let's assume that they are equal; that is, each scaling matrix performs a uniform scale. Looking at just the rotation and scaling transformations, we have

$$\mathbf{M} = \mathbf{R}_n\mathbf{S}_n \cdots \mathbf{R}_1\mathbf{S}_1\mathbf{R}_0\mathbf{S}_0$$

Since each scaling transformation is uniformly scaling, we can simplify this to

$$\mathbf{M} = \mathbf{R}_n\sigma_n \cdots \mathbf{R}_1\sigma_1\mathbf{R}_0\sigma_0$$

Using matrix algebra, we can shuffle terms to get

$$\begin{aligned}\mathbf{M} &= \mathbf{R}_n \cdots \mathbf{R}_1\mathbf{R}_0\sigma_n \cdots \sigma_1\sigma_0\\ &= \mathbf{R}\sigma\\ &= \mathbf{RS}\end{aligned}$$

where **R** is a rotation matrix and **S** is a uniform scaling matrix. So if we use uniform scaling, we can in fact decompose our matrix into a rotation and scaling matrix, as we just did.

Source Code
Demo
Separate

However, even in this case the decomposition takes three square roots and nine scaling operations to perform. This leads to an alternate approach to handling transformations. Instead of storing transformations for our objects as a single 4×4 or even 3×4 matrix, we will break out the individual parts: a scale factor s, a 3×3 rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{t}$. To apply this transformation to a point P, we use

$$\mathcal{T}(P) = \begin{bmatrix} s\mathbf{R}\mathbf{x} + \mathbf{t} \\ 1 \end{bmatrix}$$

Note the similarity to equation 3.1. We've replaced $\mathbf{A}$ with $s\mathbf{R}$ and $\mathbf{y}$ with $\mathbf{t}$. In practice we ignore the trailing 1.

Concatenating transformations in matrix format is as simple as performing a multiplication. Concatenating in our alternate format is a little less straightforward but is not difficult and actually takes fewer operations on a standard floating point processor:

$$\begin{aligned} s' &= s_1 s_0 \\ \mathbf{R}' &= \mathbf{R}_1\mathbf{R}_0 \\ \mathbf{t}' &= \mathbf{t}_1 + s_1\mathbf{R}_1\mathbf{t}_0 \end{aligned} \tag{3.16}$$

Computing the new scale and rotation makes a certain amount of sense, but it may not be clear why we don't add the two translations together to get the new translation. If we multiply the two transforms in matrix format, we have the following order:

$$\mathbf{M} = \mathbf{T}_1\mathbf{R}_1\mathbf{S}_1\mathbf{T}_0\mathbf{R}_0\mathbf{S}_0$$

But since $\mathbf{T}_0$ is applied after $\mathbf{R}_0$ and $\mathbf{S}_0$, they have no effect on it. So if we want to find how the translation changes, we drop them:

$$\mathbf{M}' = \mathbf{T}_1\mathbf{R}_1\mathbf{S}_1\mathbf{T}_0$$

Multiplying this out in block format gives us

$$\begin{aligned} \mathbf{M}' &= \begin{bmatrix} \mathbf{I} & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} s_1\mathbf{I} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{t}_0 \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} s_1\mathbf{I} & s_1\mathbf{t}_0 \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} s_1\mathbf{R}_1 & s_1\mathbf{R}_1\mathbf{t}_0 + \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \end{aligned}$$

We can see that the right-hand column vector **y** is equal to equation 3.16. So to get the final translation we need to apply the second scale and rotation before adding the second translation. Another way of thinking of this is that we need to scale and rotate the first translation vector into the frame of the second translation vector before they can be combined together.

There are a few advantages to this alternate format. First of all, it's clear what each part does — the scale and rotation aren't combined into a single 3×3 matrix. Because of this, it's also easier to change individual elements. We can update rotation, or scale through a simple multiplication, or even just set them directly. Surprisingly, on a serial processor concatenation is also cheaper. It takes 48 multiplications and 32 adds to do a traditional matrix multiplication, but only 40 multiplications and 27 adds to perform our alternate concatenation. This advantage disappears when using vector processor operations, however. In that case, it's much easier to parallelize the matrix multiplication (16 operations on some systems), and the cost of scaling and rotating the translation vector becomes more of an issue.

Even with serial processors our alternate format does have one main disadvantage, which is that we need to create a 4×4 matrix to be sent to the graphics API. Based on our previous explorations of the transformation matrix, we can create a matrix from our alternate format quite quickly; scale the three columns of the rotation matrix; and then copy it and the translation vector into our 4×4:

$$\begin{bmatrix} sr_{0,0} & sr_{0,1} & sr_{0,2} & t_x \\ sr_{1,0} & sr_{1,1} & sr_{1,2} & t_y \\ sr_{2,0} & sr_{2,1} & sr_{2,2} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which representation is better? It depends on your application. If all you wish to do is an initial scale and then apply sequences of rotations and translations, the 4×4 matrix format works fine and will be faster on a vector processor. If, on the other hand, you wish to make changes to scale as well, using the alternate format should at least be considered. And, as we'll see, if we wish to use a rotation representation other than a matrix, the alternate formation is almost certainly the way to go.

3.5 Object Hierarchies and Scene Graphs

3.5.1 Object Hierarchies

Source Code
Demo
Tank

In describing object transformations, we have considered them as transforming from the object's local frame (or local space) to a world frame (or word space). However, it is possible to define an object's transformation as

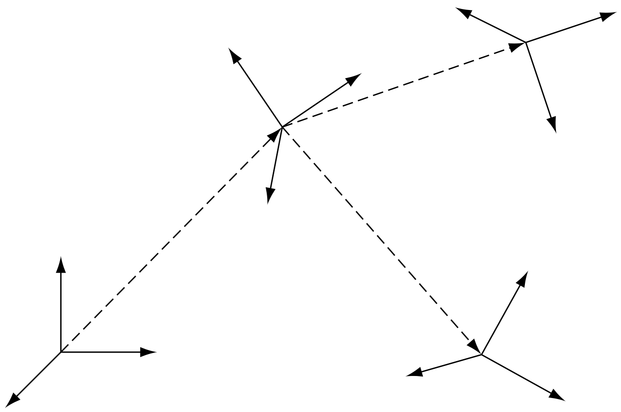


FIGURE 3.20 Hierarchy of frames.

being relative to another object's space instead. We could carry this out for a number of steps, thereby creating a hierarchy of objects, with world space as the root and each object's space as a node in a tree (Figure 3.20).

For example, suppose we wish to attach an arm to a body. The body is built with its origin relative to its center. The arm has its origin at the shoulder joint location because that will be our center of rotation. If we were to place them in the world using the same transformation, the arm would end up inside the body instead of at the shoulder. We want to find the transformation that modifies the arm's world transformation so that it matches the movement of the body and still remains at the shoulder. The way to do this is to define a transformation for the arm relative to the body's local space. If we combine this with the transformation for the body, this should place the arm in the correct place in world space relative to the body, no matter its position and orientation.

So the idea is to transform the arm to body space (Figure 3.21a) and then continue the transform into world space (Figure 3.21b). In this case, for each stage of transformation we perform the order as scale, rotate, and then translate. In matrix format the world transformation for the arm would be

$$\mathbf{W} = \mathbf{T}_{body}\mathbf{R}_{body}\mathbf{S}_{body}\mathbf{T}_{arm}\mathbf{R}_{arm}\mathbf{S}_{arm}$$

As we've indicated, the body and arm are treated as two separate objects, each with its own transformations, placed in a hierarchy. The body transformation is relative to world space, and the arm transformation is relative to the body's space. When rendering, for example, we begin by drawing the body with its world transformation and then drawing the arm with the concatenation of the body's transformation and the arm's transformation. By doing this,

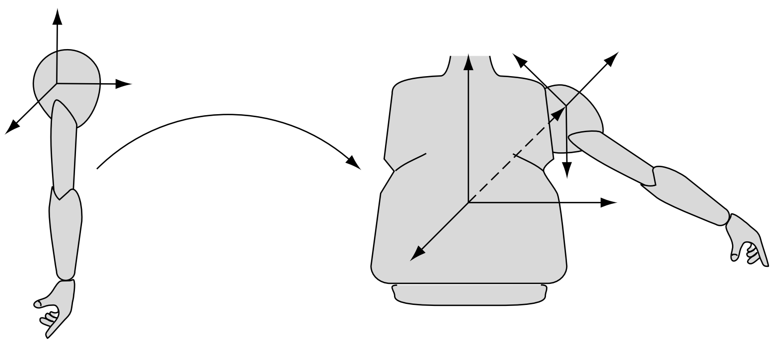


FIGURE 3.21a Mapping arm to body's local space.

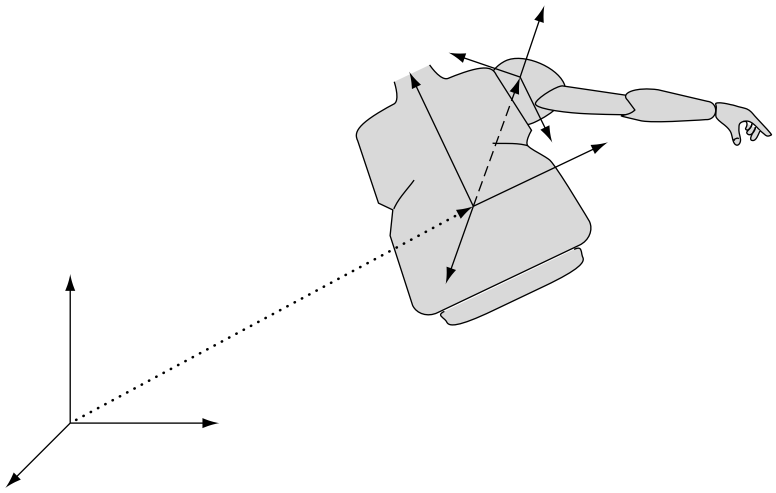


FIGURE 3.21b Mapping body and arm to world space.

we can change them independently—rotating the arm around the shoulder, for example, without affecting the body at all. Similar techniques can be used to create deeper hierarchies; for example, a turret that rotates on top of a tank chassis, with a gun barrel that elevates up and down relative to the turret.

One way of coding this is to create separate objects, each of which handles all the work of grabbing the transformation from the parent objects and combining to get the final display transform. The problem with this approach is that it generates a lot of duplicated code. Using the tank example, the code necessary for handling the hierarchy for the turret is going to be almost identical to that for the barrel. It would be much better to design a data structure that handles the generalized case of a hierarchy of frames and use that to manage our hierarchical objects. The *scene graph* is one such data structure, which we will describe in the next section.

3.5.2 Scene Graphs

Source Code
Demo
SceneGraph

The *scene graph* is meant to be used for managing hierarchical scenes, such as a collection of rooms and the objects contained within each room. While a generalized scene graph can be quite powerful, for now we will focus only on the basic structures needed for controlling hierarchical models efficiently. Although the scene graph is not necessarily a tree, for the purposes of this discussion we will be using it as such. Each object in the scene graph will have at most one parent, with one object (called the *root of the scene graph*) having no parent.

The implementation that we will present is only one of many possibilities. The more we want our scene graph to do, the more complex the implementation needs to be, but for simple purposes the following will serve. It consists of three classes: `IvSpatial`, `IvNode`, and `IvGeometry`.

`IvSpatial` is the base class. It contains two copies of the transformations as member variables. The first is a transformation relative to the parent; the transformation of a propeller relative to the submarine body, for example. We'll call this the local transformation. The second is the full transformation from the object's local space into world space, which we'll use to render and interact with the subobject—we'll call this the world transformation. This is generated by an `UpdateWorldTransform()` virtual method, which multiplies the local transformation by the parent's world transformation. Finally, we define a method called `Render()`, which uses the world transformation to render each level of the hierarchy. An abbreviated class definition looks like the following:

```
class IvSpatial
{
public:
  IvSpatial();
  virtual ~IvSpatial();

  virtual void UpdateWorldTransform();
  virtual void Render() = 0;

protected:
  IvSpatial* mParent;

  float      mLocalScale;
  IvMatrix33 mLocalRotate;
  IvVector3  mLocalTranslate;

  float      mWorldScale;
  IvMatrix33 mWorldRotate;
  IvVector3  mWorldTranslate;
};
```

where we define `UpdateWorldTransform()` as

```
void IvSpatial::UpdateWorldTransform()
{
  if (mParent)
  {
    mWorldScale = mParent->mWorldScale*mLocalScale;
    mWorldRotate = mParent->mWorldRotate*mLocalRotate;
    mWorldTranslate = mParent->mWorldTranslate
       +mParent->mWorldScale*mParent->mWorldRotate*mLocalTranslate;
  }
  else
  {
    mWorldScale = mLocalScale;
    mWorldRotate = mLocalRotate;
    mWorldTranslate = mLocalTranslate;
  }
}
```

The method `Render()` has no data to work with in this case, so it will remain undefined.

While `IvSpatial` provides a framework for managing transformations, we will never actually allocate an instance of it as an object in our scene graph. Instead, we will use one of the following subclasses.

The subclass of `IvSpatial` which acts as the root and intermediary nodes of the hierarchy is called `IvNode`. It contains a list or array of pointers to `IvSpatial` objects which are the children of the node, as well as a method for adding children to the node. The `UpdateWorldTransform()` method overrides the default method and calls `UpdateWorldTransform()` for all the child `IvSpatials` in addition to the current node:

```
class IvNode : public IvSpatial
{
public:
  IvNode();
```

```
    virtual ~IvNode();

    virtual void UpdateWorldTransform();
    virtual void Render();

  protected:
    unsigned int mNumChildren;
    IvSpatial** mChildren;
  };
```

The methods `UpdateWorldTransform()` and `Render()` become

```
  void IvNode::UpdateWorldTransform()
  {
    IvSpatial::UpdateWorldTransform();
    unsigned int i;
    for (i = 0; i < mNumChildren; ++i )
    {
      mChildren[i]->UpdateWorldTransform();
    }
  }

  void IvNode::Render()
  {
    unsigned int i;
    for (i = 0; i < mNumChildren; ++i )
    {
      mChildren[i]->Render();
    }
  }
```

The other subclass of `IvSpatial` is called `IvGeometry`. These are the leaf nodes of the scene graph, and contain the geometric data for each subobject. One way to use `IvGeometry` is to subclass it and hard-code our geometry information, but most of the time it will contain a pointer to model data. In both cases, the world transformations are updated using the base `IvSpatial` method, called by the parent `IvNode`, so we don't implement it. However, we will need to implement a `Render()` call, which builds the 4×4 matrix that is set as our world transform:

```
  class IvGeometry : public IvSpatial
  {
  public:
```

```
  IvGeometry();
  virtual ~IvGeometry();

virtual void Render();
};

void IvGeometry::Render()
{
    // build 4x4 matrix
    IvMatrix44 transform( mWorldRotate );
    transform(0,0) *= mWorldScale;
    transform(1,0) *= mWorldScale;
    transform(2,0) *= mWorldScale;
    transform(0,1) *= mWorldScale;
    transform(1,1) *= mWorldScale;
    transform(2,1) *= mWorldScale;
    transform(0,2) *= mWorldScale;
    transform(1,2) *= mWorldScale;
    transform(2,2) *= mWorldScale;
    transform(0,3) = mWorldTranslate.x;
    transform(1,3) = mWorldTranslate.y;
    transform(2,3) = mWorldTranslate.z;

    // set transform
    ::SetWorldMatrix( transform );

    // render geometry
}
```

Using the scene graph is a two-step process. In step 1, we call `UpdateWorldTransform()` at the root level, which updates transforms via a recursive traversal from the top of the tree down to the leaf nodes. At each level, we store the updated world transforms. These transforms may now be used by the game engine for other purposes. Step 2 occurs once we're ready to render the object, when we do another recursive tree traversal by calling `Render()` on it. `UpdateWorldTransform()` does *not* have to be called on the root of the scene graph in every rendered frame. Generally, it is called once on the root object, directly following the creation of the scene graph. Thereafter, it only needs to be called at or above any and all `IvNodes` whose local transforms have changed since the last call to `UpdateWorldTransform()`. This is often a small subset of the scene graph. In other words, it is often sufficient and much faster to call `UpdateWorldTransform()` several times on disjoint subsections (subtrees) of the scene graph that have changed than it is to make the single call to `UpdateWorldTransform()` at the root of the scene.

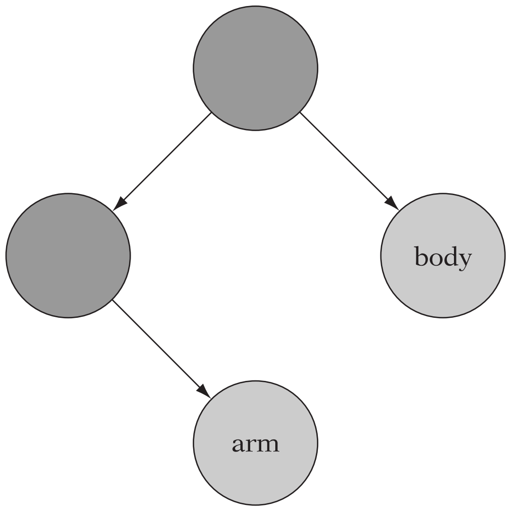


FIGURE 3.22 Scene graph of body–arm example.

Figure 3.22 shows our body–arm example stored as a scene graph. Note that the body is not a root node — it is a geometry leaf node that hangs directly off of the root node. This leads to some duplication of transformation information, but that is the price we pay for maintaining transformations in the base class.

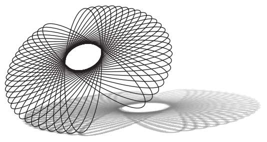
One might wonder why we have two recursive calls — one for generating the new transformations and one for rendering — or, for that matter, why we bother storing the transforms at all. We could just have one recursive call that generates the world transformation at each level and then passes the result down as a function argument. At the leaf level, we would create the transformation data and then render the data directly. However, there is usually a culling step where we try to avoid rendering models that are not currently visible on the screen. As we will see, it is convenient to keep the transformation data around for this and other purposes. Scene graphs are a very flexible and modular technique, and can be mixed with other data structures and rendering systems. For example, scene graphs are sometimes used to compute hierarchical transforms without using a hierarchical `Render()` function to draw the scene graph. In such cases, the scene graph is used only to manipulate the local transforms of objects and update the world transforms of visible geometry. Another method (such as a flat list of all of the leaf `IvGeometry` objects) is used to render the scene.

3.6 Chapter Summary

In this chapter we've discussed the general properties of affine transformations, how they map between affine spaces, and how they can be represented and performed by matrices at one dimension higher than the affine spaces involved. We've covered the basic affine transformations as used in interactive applications and how to combine three of them — scaling, rotation, and translation — to manipulate our objects within our world. While it may be desirable to separate a given affine transformation back into scaling, rotation, and translation components, we have seen that it is not always possible when using nonuniform scaling. Separating components in this manner may not be efficient, so we have presented an alternative affine transformation representation with the three components separated. Finally, we have discussed how to construct transformations relative to other objects, which allows us to create jointed, hierarchical structures.

For those interested in reading further, information on affine algebra can be found in Schneider and Eberly [96], as well as in deRose [25]. The standard affine transformations are described in most graphics textbooks, such as Möller and Haines [79] and Foley and van Dam [36]. Further details on hierarchical transformation management and scene graph construction and usage can be found in Eberly [27].

Chapter 4
Real-World Computer Number Representation



4.1 Introduction

In this chapter we'll discuss what is perhaps the most fundamental basis upon which 3D graphics pipelines are built — computer representation of numbers. While 3D programmers often use integers, unsigned integers, and floating-point numbers successfully without any understanding of how they are implemented, this can lead to subtle bugs and performance problems eventually. Most basic undergraduate computer architecture books [104] present the basics of integral data types (e.g., `int` and `unsigned int`, `short`, etc. in C/C++) but give only brief introductions to floating-point and other nonintegral number representations. Since the mathematics of 3D graphics are generally real-valued (wintess the predominance of $\mathbb{R}$, $\mathbb{R}^2$, and $\mathbb{R}^3$ in the preceding chapters), it is important for anyone in the field to understand the features, limitations, and idiosyncracies of the computer representation of these nonintegral types.

This chapter will begin by reviewing some of the issues surrounding the representation of whole numbers and integers on a computer. This review mainly serves to introduce concepts that will carry over to a discussion of real numbers. The chapter will discuss two major computer representations of the real numbers, fixed point and floating point, along with their bitwise formats, basic operations, features, and limitations. By design, we will transition from general mathematical discussions of number representation toward implementation-related topics of specific relevance to 3D graphics programmers. Much of the chapter will be spent on the ubiquitous IEEE floating point numbers, especially discussions of floating point limitations that often cause issues in 3D pipelines. It will also present a brief case study of floating-point–related performance issues in a real application.

4.2 Representing Integral Types on a Computer

4.2.1 Finiteness of Representation

The sets of whole numbers ($[0, 1, 2 \ldots]$, known as $\mathbb{W}$), integers ($[\ldots -2, -1, 0, 1, 2 \ldots]$, known as $\mathbb{Z}$), and real numbers (e.g., 1.5, 1/3, $\sqrt{2}$, known as $\mathbb{R}$) share one trait in common — they each have infinitely many elements. Computers, on the other hand, by their very physical nature can only represent a finite number of different values. As a result computers cannot represent *any* of the aforementioned number sets exactly and completely. We will have to settle for some finite subset of each. The sizes of these finite sets are determined by the number of distinct values that can be represented by the given number of storage systems.

Modern computers store their numbers as binary codings: finite, fixed-length strings of bits. (For a basic discussion of binary number representation, we refer the reader to a basic computer architecture text, such as Stallings [104].) As such, any N-bit computer number representation can only represent 2^N distinct values. For our purposes, we will generally assume 32-bit "words," which can represent about 4 billion different values. Each of the distinct values in a given representation can represent at most one element of the number set exactly. While 4 billion may seem an enormous number of possible values, we shall see that it becomes painfully finite when used to cover the set of real numbers.

4.2.2 Range

The range of a number representation system is described by two values: the minimum representable value and the maximum representable value, often written as the interval [*minimum, maximum*]. Values outside of this interval are assumed to be unrepresentable (although, as we shall see, there are some cases in floating point where values outside of the proper "range" of a representation can still be represented indirectly). We shall review the common computer representations of integers and whole numbers here, mainly to discuss the properties of these numbers and to give examples of range.

The whole numbers ($\mathbb{W}$) have an inherent, finite minimum: 0. In order to represent the whole numbers with a finite computer representation, we use the fact that for any finite whole number W_{max}, there will be a finite number of elements (possibly zero) $w \in \mathbb{W}$ such that $w \leq W_{max}$. In fact, the size of such a set is $(W_{max} + 1)$. Based on this observation, we can represent a useful K-element subset of the whole numbers by simply selecting a maximum representable value of $W_{max} = K - 1$. All whole numbers less than K can be represented exactly by such a system.

The type `unsigned int` is the most commonly used C/C++ representation of $\mathbb{W}$. For smaller numbers, `unsigned short` and `unsigned char` are also used. We will discuss only `unsigned int` in this section; the analysis of the other representations is analogous. On a 32-bit computer, the representation of `unsigned int` is simply an unsigned 32-bit binary number, capable of representing 2^{32} distinct values. As a result, all whole numbers in the range $[0, 2^{32} - 1]$ can be represented by `unsigned int`. For a basic discussion of the binary representation of unsigned numbers, see [104]. Note that the C++ specification [31] does not require `unsigned int` to be a 32-bit type; however, for the course of this discussion, we will assume the common case, which is a 32-bit `unsigned int`.

The integers have no inherent minimum or maximum. In order to represent the integers on a computer, we must select a finite *pair* of values Z_{min} and Z_{max}. There will be a finite number of elements (possibly zero) $i \in \mathbb{Z}$, such that $Z_{min} \leq i \leq Z_{max}$. In fact, the size of such a set is $max(0, Z_{max} - Z_{min} + 1)$. Unlike the whole numbers, there are infinitely many K-element sets, one for each chosen minimum value ($[Z_{min}, Z_{min} + K - 1]$). Historically, most computer representations of integers select Z_{min} such that the K-element set is as evenly distributed around 0 as possible:

$$Z_{min} \approx -Z_{max}, or\, Z_{min} \approx -\frac{K}{2}$$

This is done to ensure that for as many elements as possible:

$$i \in [Z_{min}, Z_{max}] \implies -i \in [Z_{min}, Z_{max}]$$

The type `int` is the most common C/C++ representation of $\mathbb{Z}$ (as before, we will not discuss the similar, smaller `short` and `char`). On a 32-bit computer the representation of `int` is simply a signed 32-bit binary representation using so-called '2's complement' to represent both positive and negative numbers (see [88] for a review of 2's complement). Being a 32-bit representation, it is capable of representing 2^{32} distinct values. As this is an even number of elements, there is no way to represent 2^{32} distinct integers *and* fulfill the requirement to have the range of the representation exactly center about 0. The standard 2's complement format of `int` represents all integers in the range $[-(2^{31}), 2^{31} - 1]$, meaning that while $-(2^{31})$ can be represented, its negative $-(-(2^{31})) = 2^{31}$ is out of range, since $2^{31} > 2^{31} - 1$.

Overflow

Overflow is a term used to describe what occurs when a computation generates a result with a value outside the range of the representation in use. As we shall see, different representation systems have different ways of dealing with this situation, but in all cases the result cannot be represented exactly, and in most cases the represented result is *very* different from the correct result. In general, the best method with any number system is to avoid overflow entirely. Such a strategy requires that the programmer understand the exact range of the representation(s) that they are using. The following sections will discuss the range of the number representations, along with the likely results of overflow. In many cases, standardization has set the overflow behavior of a given representation across platforms.

As mentioned, the range of the common type `unsigned int` on a 32-bit computer is $[0, 2^{32} - 1]$. Application code must take care to ensure that the results of all operations involving `unsigned int` values are in range. Positive overflow of 32-bit unsigned integer values is relatively uncommon in correct code. For example, if a counter is incremented once per frame on a game that is running at 100 frames per second as a 32-bit `unsigned int`, this counter will overflow only after

$$\frac{2^{32} frames}{100\frac{frames}{sec} \times 60\frac{secs}{min} \times 60\frac{mins}{hour} \times 24\frac{hours}{day} \times 365\frac{days}{yr}} \approx 1.4 \text{ years!}$$

Negative overflow of `unsigned int` values is rather easy to generate, especially in code with simple logic errors. Commonly, such code will subtract a larger number from a smaller one, leading to a negative result (which cannot be represented as a whole number). The C/C++ standard requires that `unsigned int` operations always return a result that is equal to the correct value *modulo* 2^{32}.

For most common applications, this result is not particularly useful as it leads to the following examples:

$$(2^{32} - 1) + 1 \rightarrow 0$$
$$0 - 1 \rightarrow (2^{32} - 1)$$

However, it does make sense when considered as a part of the larger picture. The result of any unsigned integer operation is the least significant 32 bits of the correct result. Using assembly language (where the overflow flag, or "carry bit," is accessible), it is possible to chain together 32-bit addition operations to add 64-bit (or larger) numbers. The carry bit "carries" into the low-order bit of the next 32-bit operation.

For most applications, the best way to handle negative overflow of `unsigned int` is to avoid the situation by ensuring that the result of the subtraction will be nonnegative prior to computation and reworking code that can generate negative overflows.

Range and Type Conversion

Mathematically, whole numbers are a proper subset of the set of integers. However, on a computer our representations of integers (`int`) and whole numbers (`unsigned int`) have the same size (generally 32 bits), so the set of `int`s and `unsigned int`s each have the same number of elements. This leads to the (sometimes problematic) fact that on a computer, `unsigned int` $\not\subseteq$ `int` and `int` $\not\subseteq$ `unsigned int`. Each set contains values that cannot be represented by the other set. Programmers must be very careful when converting between `int` and `unsigned int` to avoid problems.

Given that the range of `int` on a 32-bit machine is $[-(2^{31}), 2^{31} - 1]$ and the range of `unsigned int` is $[0, 2^{32} - 1]$, the safe range for conversion is thus

$$[-(2^{31}), 2^{31} - 1] \cap [0, 2^{32} - 1] = [0, 2^{31} - 1]$$

Applications should check `int` values to make sure they are not negative and `unsigned int` values to ensure that they will not overflow $2^{31} - 1$ prior to converting (casting) them. Most C/C++ compilers will generate a warning (at some warning levels) unless a signed/unsigned cast is made explicit.

4.3 Representing Real Numbers

Real numbers are, to most developers, the heart and soul of a 3D graphics system. Most of the rest of the text is based upon real numbers and spaces such as $\mathbb{R}^2$ and $\mathbb{R}^3$. They are the most flexible of the number systems we have described in this chapter and, not surprisingly, the most complicated and problematic to represent on a computer. We will present two different methods that are used to represent real numbers on computers today and will include numerous sections describing common issues that arise from the use of these representations in real-world applications.

All of the issues relating to storage of integers and whole numbers discussed thus far will continue to be issues with real number representation. However, real number representations add additional complexities that will result in implementation trade-offs, subtle errors, and difficult to trace performance issues that can easily confuse the programmer.

4.3.1 Approximations

While computer representations of whole numbers (`unsigned int`) and integers (`int`) are limited to a finite subset of their pure counterparts, in each case the finite set is contiguous; that is, if i and $i+2$ are both representable, then $i+1$ is also representable. Inside the range defined by the minimum and maximum representable integer values, all integers can be represented exactly. This stems from the earlier observation that any finitely bounded range of integers contains a finite number of elements.

When dealing with real numbers, however, this is no longer true. A subset of real numbers can have infinitely many elements even when bounded by finite minimal and maximal values. As a result, no matter how tightly we bound the range of real numbers (other than the trivial case of $R_{min} = R_{max}$) that we choose to represent, we will be unable to represent that subset of the real numbers exactly. Issues of both range *and* precision will thus be constant companions over the course of our discussion of real number representation. In order to adequately understand the representations of real numbers, we need to understand the concept of precision and error.

4.3.2 Precision and Error

For any number representation system, we imagine a generic function $Rep(A)$, which returns the value in that system that is closest to the value A. In a perfect representation system, $Rep(A) = A$ for all values of A. When representing real numbers, however, even limiting range to finite extremes will not allow us to represent all numbers in the bounded range exactly. $Rep(A)$ will be a many-to-one mapping, with infinitely many real numbers A mapping to each distinct value returned by $Rep(A)$. For each such distinct $Rep(A)$, almost all values A that map to it will not be represented exactly. In other words, for almost all real values A, $Rep(A) \neq A$. The obvious result in such cases is that $(Rep(A) - A) \neq 0$. The representation in such a case is an approximation of the actual value.

Making use of $(Rep(A) - A)$, we can define several derived values that form metrics of the error induced by representing A in the representation system. These two kinds of error metrics are called *absolute error* and *relative error*.

The simplest way to represent error is "absolute error," which is defined as

$$AbsError = |Rep(A) - A|$$

This is simply the "number line" distance between the actual value and its representation. While this value does correctly signify the difference between the actual and representative values, it does not quantify another important factor in representation error—the scale at which the error affects computation.

To better understand this, imagine a system of measurement that is accurate to within a kilometer. Such a system might be considered suitably accurate for measuring the 149,597,871 km between the earth and the sun. However, it would likely be woefully inaccurate at measuring the size of an apple (0.00011 km), which would be rounded to 0 km! Intuitively, this is obvious, but in both cases the absolute error of representation is less than 1 km. Clearly, absolute error is not sufficient in this case.

Relative error takes the scale of the value being approximated into account. It does so by dividing the absolute error by the actual value being represented. Relative error is defined as

$$RelError = \left| \frac{Rep(A) - A}{A} \right|$$

As such, relative error is dimensionless; even if the values being approximated have units (such as kilometers), the relative error has no units. Due to the division, relative error cannot be computed for a value that approximates zero. It is a measure of the ratio of the error to the magnitude of the value being approximated. Revisiting our previous example, the relative errors in each case would be (approximately)

$$RelError_{Sun} = \left| \frac{1\,\text{km}}{149,597,871\,\text{km}} \right| \approx 7 \times 10^{-9}$$

$$RelError_{Apple} = \left| \frac{0.00011\,\text{km}}{0.00011\,\text{km}} \right| = 1.0$$

Clearly, relative error is a much more useful error metric in this case. The earth–sun distance error is tiny (compared to the distance being measured), while the size of the apple was estimated so poorly that the error had the same magnitude as the actual value. In the former case a relatively "exact" representation was found, while in the latter case the representation is all but useless.

4.4 Fixed Point

4.4.1 Introduction

Much is made of the performance of hardware floating point units (FPUs) in modern desktop processors and full-sized game consoles. The basic use of floating point numbers is familiar to even the novice programmer. However, floating point is not the only way that real numbers are approximated on computers. In fact, for decades another representation, fixed point, was far more popular owing to its high performance and accuracy when used correctly, even on low-powered computers.

Over the past decade, fixed point numbers have become somewhat of a "lost art" to all but the most hardcore, experienced 3D programmers. For example, 3D PC games written in the late 1990s and beyond tended to use floating point heavily (if not exclusively). However, the popularity of powerful handheld computers and cellular telephones has brought fixed-point arithmetic back to the forefront of 3D game development. With the constant pressure on hardware manufacturers to make smaller, lower-cost embedded chips, it is likely that there will be a need for fixed point code in handheld 3D games for some time to come. This trend alone has caused discussion of fixed point number representation and computation among 3D game programmers to be extremely relevant once again.

4.4.2 Basic Representation

Fixed point numbers are a method of representing a subset of the real numbers on a computer. Fixed point numbers are based upon the computer representation of integers. In fact, as we shall see, integers can be thought of as a special case of fixed point. Like the computer representations of integers upon which they are built, fixed point numbers are finite. As such, they cannot represent the entire set of real numbers. However, the range and precision limitations of fixed point numbers are very simple, making them easy to describe and analyze.

Fixed point numbers are based on a very simple observation with respect to computer representation of integers. In the standard binary representation, each bit represents twice the value of the bit to its right, with the least significant bit representing 1. The following diagram shows these powers of two for a standard 8-bit unsigned value:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Just as a decimal number can have a decimal point, which represents the break between integral and fractional values, a binary value can have a binary point, or more generally a radix point (a decimal number is referred to as radix 10, a binary number as radix 2). In the previous number layout, we can imagine the radix point being to the right of the last digit. However, it does not have to be placed there. For example, let us revisit the previous case, this time placing the radix point in the middle of the number (between the fourth and fifth bits). The diagram would then look like this:

2^3	2^2	2^1	$2^0 \, . \, 2^{-1}$	2^{-2}	2^{-3}	2^{-4}
8	4	2	$1 \, . \, \frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Now, the least significant bit represents 1/16. The basic idea behind fixed point is one of scaling. A fixed point value is related to an integer with the same bit pattern by an implicit scaling factor. This scaling factor is fixed for a given fixed point format and is the value of the least significant bit in the representation. In the case of the preceding format, the scaling factor is 1/16.

The standard nomenclature for a fixed point format is "M-dot-N," where M is the number of integral bits (to the left of the radix point) and N is the number of fractional bits (to the right of the radix point). For example, the 8-bit format in our example would be referred to as "4-dot-4." As a further example, regular 32-bit integers would be referred to as "32-dot-0" because they have no fractional bits. More generally, the scaling factor for an M-dot-N format is simply 2^{-N}. Note that, as expected, the scaling factor for a 32-dot-0 format (integers) is $2^0 = 1$. No matter what the format, the radix point is "fixed" (or locked) at N bits from the least significant bit; thus the name "fixed point."

4.4.3 Range and Precision

Computing the range and precision for a given fixed point format is very easy and can be computed solely by knowing the "M-dot-N" format name. This simple analysis is made possible by the previously stated fact about the relationship between fixed point numbers and integers with the same bitwise representation. For any fixed point number viewed directly as an integer, we compute the fixed point number's value in M-dot-N format by multiplying the integer by a scaling factor equal to 2^{-N}.

To compute the range of an M-dot-N format, we recall the earlier discussion regarding 2's complement integers. We know that a 2's complement integer with B bits has range $[-(2^{B-1}), 2^{B-1} - 1]$. Since the total number of bits in a fixed-point representation is $M + N$, this leads to a fixed-point range of

$$\left[-(2^{M+N-1}) \times 2^{-N}, (2^{M+N-1} - 1) \times 2^{-N}\right]$$
$$\left[\frac{-(2^{M+N-1})}{2^N}, \frac{(2^{M+N-1} - 1)}{2^N}\right]$$
$$\left[-(2^{M-1}), \left(2^{M-1} - \frac{1}{2^N}\right)\right]$$

The precision of the representation can be computed just as simply. When dealing with integers, the spacing between each integer and its nearest neighbor is simply 1.0. Multiplying by the fixed point format's scaling factor, we find that the difference between any M-dot-N number and its closest neighbor is

$$1.0 \times 2^{-N} = \frac{1}{2^N}$$

Given this fixed distance between any dot-N fixed point value and its closest representable neighbor, we know that any real number A within the valid range for the preceding M-dot-N format is, at worst, different from its representation by half the distance between the values directly above and below A. So, A can be represented with an absolute error of at most

$$AbsError_A = |Rep(A) - A| \leq \frac{1}{2^N} \times \frac{1}{2} = \frac{1}{2^{N+1}}$$

This absolute error bound is constant across the range of the format. On the other hand, the relative error bound is

$$RelError_A = \left|\frac{Rep(A) - A}{A}\right| \leq \frac{AbsError_A}{|A|} = \frac{1}{|A| \times 2^{N+1}}$$

which rises sharply as A tends toward zero. In other words, with a fixed-point system the relative error falls as magnitudes increase. This leads to the basic guideline that an application must determine how small its smallest values can become and set the fractional precision based on this quantum.

Converting between Real and Fixed-Point

Converting a real number R to an M-dot-N fixed-point number F can be accomplished via the following method:

$$F = round(R \times 2^N)$$

where *round* is the function used to round a real number to the nearest integer. Basically, this method scales the real number to the correct scaling value for the fixed point format, and then rounds away any precision beyond what can be represented in the given format. For example, to convert the value 4.5 to our 4-dot-4 format, we do the following:

$$\begin{aligned} F &= round(4.5 \times 2^4) \\ &= round(4.5 \times 16) \\ &= round(72) \\ &= 72 \end{aligned}$$

$$72 = \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 1 & 0 & 0\,.\,1 & 0 & 0 & 0 \\ \hline\end{array}$$

which represents 4.5 exactly. Note that in this case, the *round* operation had no effect. If the *round* operation had changed the value, then this would indicate that the M-dot-N formula could not represent the given value exactly. An example of such a case is 3.7 represented in 4-dot-4 format:

$$\begin{aligned} F &= round(3.7 \times 2^4) \\ &= round(59.2) \\ &= 59 \end{aligned}$$

$$59 = \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 0 & 1 & 1\,.\,1 & 0 & 1 & 1 \\ \hline\end{array}$$

which represents the rational value 3.6875. The absolute error of representation in this case is $3.7 - 3.6875 = 0.0125$. This is much less than the maximum possible absolute error in 4-dot-4 format, which is $1/2^5 = 0.03125$.

It is very important that the rounding step be done after the scaling, or else the real number will be rounded to an integer and *all* of the fractional precision will be lost (the N least-significant bits of the resulting fixed point value will be zeros). Note that real numbers outside of the range of the fixed point format will overflow during the conversion to the integer format and must be considered invalid.

Converting back to the desired real number is as simple as treating the fixed point number's integer representation as a real number (in C/C++, this is simply a typecast) and then scaling that real number by the $1/2^N$ scaling factor.

The conversion methods between integers and fixed point values are even simpler. This is due to the fact that the scaling factors of the fixed point formats we have discussed are all powers of two. Multiplying an integer by 2^N is equivalent to shifting that integer to the left by N bits. Shifting is an operation that is supplied by the integer math units (arithmetic logic units, or ALUs) of all major CPUs and is extremely fast (free on some CPUs when done at the same time as another math operation). Dividing an integer by 2^N is equivalent to shifting the integer to the right by N bits.

We can use these fast special cases of multiplication and division in our integer/fixed point conversion. The conversion from integer to fixed point can never lose precision (although it will overflow if the integer is not in the range of the fixed point format) and is implemented by shifting the integer to the left by N bits. The conversion from fixed point to integer will never overflow but often loses precision and is implemented by shifting the integer to the right by N bits. Note that shifting to the right truncates the number. In 2's complement, this computes the floor of the number; it does not round it. In order to round during the conversion to integer, we must first add 2^{N-1} (which is equal to 1/2 in the M-dot-N format) and *then* shift the result to the right by N bits. The addition of 1/2 prior to the shift turns the truncation into a form of rounding.

4.4.4 Addition and Subtraction

Addition and subtraction of two fixed point numbers of the same format is extremely simple—they are merely the integer versions of addition and subtraction. This is possible because two M-dot-N fixed point numbers have radix points that line up (just like standard integers). A 4-dot-4 example follows:

	Bits							Integer	Real
	0	0	0	1 . 0	0	0	0	[16]	(1.0)
+	0	0	0	0 . 1	1	0	0	[12]	(0.75)
	0	0	0	1 . 1	1	0	0	[28]	(1.75)

4.4.5 Multiplication

The simplicity of addition and subtraction may lead one to hope that multiplication and division of fixed point numbers are equally simple. However, a quick example shows a problem with this method. For example, let us convert 0.5 and 0.25 into 4-dot-4 fixed point and multiply them together.

We expect a result of 0.125. First, we convert the real numbers to 4-dot-4 fixed point:

$$0.5 \rightarrow 8 = \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0\,.\,1 & 0 & 0 & 0 \\ \hline \end{array}$$

$$0.25 \rightarrow 4 = \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 0\,.\,0 & 1 & 0 & 0 \\ \hline \end{array}$$

Next, we multiply them using the standard integer method:

	Bits							Integer	Real
	0	0	0	0 . 1	0	0	0	[8]	(0.5)
$\times$	0	0	0	0 . 0	1	0	0	[4]	(0.25)
Incorrect!	0	0	1	0 . 0	0	0	0	[32]	(2.0)

The result is clearly incorrect and just as clearly (given the magnitude of the error) not simple rounding error. However, there is a clear reason for this error. Fixed point numbers are not equivalent to their bitwise representation as integers, but rather as their integer representation times the $1/2^N$ scaling value. Thus, the integer bits represent the real number times 2^N. If we recompute the multiplication just done adding these scale values, we find the following:

$$(0.5 \times 2^4) \times (0.25 \times 2^4) = (0.125 \times 2^4) \times 2^4$$

The problem is that each of the two operands brings its own implicit scale value. Multiplying these together causes the result to be too large by exactly the scaling factor. Thus, to reestablish the correct fixed point format, we must divide the result by 2^4:

$$\frac{(0.5 \times 2^4) \times (0.25 \times 2^4)}{2^4} = (0.125 \times 2^4)$$

This method generalizes as follows: to multiply two M-dot-N fixed point numbers, we multiply their representations using the integer multiplication method and then divide the result by 2^N. Using the same observation as we did for integer/fixed-point conversion, we replace the division by 2^N with an N-bit right shift (written as $\gg N$) of the result of the integer multiplication. This gives a method for multiplying two M-dot-N numbers $A_{M.N}$ and $B_{M.N}$ of

$$(A_{M.N} \times B_{M.N}) \gg N$$

Next, we show this multiplication method graphically, computing $1.0 \times 0.375 = 0.375$:

							Bits	Integer	Real
	0	0	0	1 . 0	0	0	0	[16]	(1.0)
×	0	0	0	0 . 0	1	1	0	[6]	(0.375)
	0	1	1	0 . 0	0	0	0	[96]	
≫							4 Bits		
	0	0	0	0 . 0	1	1	0	[6]	(0.375)

4.4.6 Division

Fixed point division suffers from another issue of scale correction but in the inverse manner to what happened with multiplication. Rather than ending up with two scale values multiplying together and requiring correction, in the case of division, the scale values cancel out, and the result is represented as an integer with no fractional precision. We'll demonstrate with the same numbers used in our multiplication example:

$0.5 \rightarrow 8 =$	0	0	0	0 . 1	0	0	0

$0.25 \rightarrow 4 =$	0	0	0	0 . 0	1	0	0

We attempt to divide one by the other using the standard integer method:

							Bits	Integer	Real
	0	0	0	0 . 1	0	0	0	[8]	(0.5)
/	0	0	0	0 . 0	1	0	0	[4]	(0.25)
Incorrect!	0	0	0	0 . 0	0	1	0	[2]	(0.125)

In this case, we can see that the scale values have canceled:

$$\frac{0.5 \times 2^4}{0.25 \times 2^4} = \frac{0.5}{0.25} = 2 = (0.125 \times 2^4)$$

To reestablish the correct fixed point format, we could multiply the result by 2^4. However, the problem with such a method is that the result would have no fractional precision (a 4-dot-4 number times 2^4 is always an integer)! The precision was lost in the division. To avoid this loss of precision, we must multiply the dividend by 2^4:

$$\frac{0.5 \times 2^4 \times 2^4}{0.25 \times 2^4} = \frac{0.5 \times 2^4}{0.25} = 2 \times 2^4 = (2.0 \times 2^4)$$

This division method generalizes as follows: to divide one M-dot-N fixed-point number by another, we multiply the dividend by 2^N and then divide their representations using the integer division method. Using the same observation as we did for integer/fixed-point conversion, we replace the multiplication by 2^N by shifting the dividend to the left (written $\ll$) by N bits. This gives a method for dividing two M-dot-N numbers $A_{M.N}$ and $B_{M.N}$ of

$$\frac{(A_{M.N} \ll N)}{(B_{M.N})}$$

We show this method graphically, computing $0.25/2.0 = 0.125$:

							Bits	Integer	Real
	0	0	0	0 . 0	1	0	0	[4]	(0.25)
$\ll$							4 Bits		
	0	1	0	0 . 0	0	0	0	[64]	
/	0	0	1	0 . 0	0	0	0	[32]	(2.0)
	0	0	0	0 . 0	0	1	0	[2]	(0.125)

4.4.7 Real-World Fixed Point

The astute reader will note that the preceding examples of fixed point operations were rather contrived. In fact, it would have been very easy to generate examples of the algorithms discussed thus far that caused overflow and incorrect results. The very nature of fixed point numbers, the fact that even small real values are represented by large integers (e.g., the real value 1.0 being represented by 65,536 in 16-dot-16) can lead to overflow in surprisingly low-magnitude situations. Additionally, while fixed point has been presented as a fast alternative to floating point on integer-only platforms, it is apparent that there is likely to be some performance penalty for the additional shifting that is required in many fixed point operations.

In fact, several modern processors take both of these issues into account, offering features that can assist the programmer. In one case, such a feature makes the extra shifting operations computationally inexpensive (or even free). In another, a set of extra instructions makes overflow far less of a problem. Details of these situations, as well as both platform-dependent and platform-independent methods of dealing with them, are covered in the following section.

4.4.8 Intermediate Value Overflow and Underflow

The basic method for fixed point multiplication discussed thus far requires that the intermediate multiplied value be shifted downward in magnitude to reestablish the correct position of the radix point. A side effect of this method is that the intermediate value can overflow, even if the final result should be well within the range of the fixed point format. As an introductory example, we demonstrate 1.0×1.0 in 4-dot-4 fixed point. Clearly, the result should be 1.0, obviously within range:

					Bits			Integer	Real
	0	0	0	1 . 0	0	0	0	[16]	(1.0)
$\times$	0	0	0	1 . 0	0	0	0	[16]	(1.0)
Overflow!	0	0	0	0 . 0	0	0	0	[256 = 0]	(0.0)
$\gg$							4 Bits		
Incorrect!	0	0	0	0 . 0	0	0	0	[0]	(0.0)

Even a simple multiplication can result in overflow in the intermediate value. Seeing this situation, one might be tempted to simply move the shift operation up in the process, shifting first and then multiplying the preshifted result. For example, in our 4-dot-4 case imagine a method in which each operand was preshifted by two bits prior to the multiplication. Multiplying $1.0 \times 1.0 = 1.0$:

					Bits			Integer	Real
	0	0	0	1 . 0	0	0	0	[16]	(1.0)
$\gg$							2 Bits		
	0	0	0	0 . 0	1	0	0	[4]	(0.25)
$\times$	0	0	0	1 . 0	0	0	0	[16]	(0.25)

≫				2 Bits					
	0	0	0	0 . 0	1	0	0	[4]	(0.25)
	0	0	0	1 . 0	0	0	0	[16]	(1.0)

Success! However, blindly preshifting operands can result in problems as well. This same method nets much less satisfying results with the following case of $7.0 \times 0.125 = 0.875$. Note what happens to the second operand when preshifted:

							Bits	Integer	Real
	0	0	0	0 . 0	0	1	0	[2]	(0.125)
≫							2 Bits		
	0	0	0	0 . 0	0	0	0	[0]	(0.0)

With one of the operands being truncated to zero, the result of the multiplication will be zero! This is quite a sizable error. Clearly, no single shifting method can satisfactorily deal with all precision and overflow cases.

Extended Precision Hardware Assistance

The method that is perhaps the best at dealing with overflow and underflow requires some outside assistance on the part of the hardware platform. This assistance comes in the form of extended-precision mathematical operations. Such instructions are based on the fact that two N-bit numbers when multiplied together can require up to $2N$ bits to avoid overflow. Numerous modern processors (especially those without floating point units, such as the ARM architecture [97]) include either 16-bit multiplication operations with 32-bit results or 32-bit multiplication operations with 64-bit results. Such operations are practically (if not specifically) tailor-made for fixed point implementations.

For example, imagine our 4-dot-4 (8-bit) fixed point system. As we demonstrated, if we multiply 1.0×1.0 using the direct method, the operation's intermediate value will overflow, leaving an incorrect result. However, with an 8-bit $\times$ 8-bit $\Longrightarrow$ 16-bit instruction, even the larger-magnitude operation 2.0×2.0 can be completed using the direct method. Note that the 16-bit intermediate result is actually the correct answer as well, but in 8-dot-8 format:

								0	0	1	0 . 0	0	0	0	(2.0)
							×	0	0	1	0 . 0	0	0	0	(2.0)
0	0	0	0	0	1	0	0 . 0	0	0	0	0	0	0	0	

$\gg$ 4 Bits

0	1	0	0 . 0	0	0	0	(4.0)

It is important to note that these extended-precision operations do not actually extend the values that can be represented in a given fixed point format. If the result cannot be represented in the final format, the only option would be to change the format used or else reformulate the problem. In the following example, the intermediate result can be represented in the 16-dot-16 intermediate format, but the final result is truncated incorrectly and ends up overflowing:

									0	1	1	0 . 0	0	0	0	(6.0)
$\times$									0	1	1	1 . 0	0	0	0	(7.0)
	0	0	1	0	1	0	1	0 . 0	0	0	0	0	0	0	0	(42.0)
															$\gg$ 4 Bits	
	0	0	0	0	0	0	1	0 . 1	0	1	0	0	0	0	0	(42.0)
Overflow – Incorrect!									1	0	1	0 . 0	0	0	0	(−6.0)

In this case the extended precision was not the answer, as the final result still had to be converted to the shorter format. What extended precision does avoid is overflow in intermediate results. This, in turn, avoids the need to preshift precision from the operands, avoiding unnecessary underflow. With careful programming, these instructions can be used for strings of operations, with the conversion from the long format to the original format happening once at the end. A common instruction form that is given on extended-precision hardware is an extended-precision multiply-accumulate, as follows:

$$\text{32-bit} \times \text{32-bit} + \text{64-bit} \Longrightarrow \text{64-bit}$$

An extended-precision multiply-accumulate instruction is useful for quickly computing a safe, precise 16-dot-16 fixed point dot product. We assume that we begin with the pair of 3-vectors (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2):

1. $X_1 \times X_2 \Longrightarrow Accumulator_{64}$
2. $Y_1 \times Y_2 + Accumulator_{64} \Longrightarrow Accumulator_{64}$
3. $Z_1 \times Z_2 + Accumulator_{64} \Longrightarrow Accumulator_{64}$
4. $Accumulator_{64} \gg 16 \Longrightarrow Result_{32}$

4.4.9 Limits of Fixed Point

To better understand the significant limitations of fixed point, even with hardware-assisted extended-precision, we shall consider one of the most popular fixed point formats in general fixed point libraries, the 32-bit signed 16-dot-16 format. Although applications can pick a wide range of formats, and can even use multiple formats in the same application, 16-dot-16 is representative. We can summarize the minimum and maximum representable values in this format, as well as the value of epsilon (ϵ), the distance between adjacent representable values as follows:

$$\begin{aligned} \text{Maximum representable value:} &\quad \mathit{Max}_{16.16} \approx 32767 \\ \text{Minimum representable value:} &\quad \mathit{Min}_{16.16} \approx -32768 \\ \text{Smallest positive value:} &\quad \mathit{Eps}_{16.16} \approx 1.5 \times 10^{-5} \end{aligned}$$

While these may seem like large amounts of range and precision (and indeed they are), it should be noted that $\sqrt{\mathit{Max}_{16.16}} \approx 181$, and $\sqrt{\mathit{Eps}_{16.16}} \approx 0.004$. In other words, if the application needs to store the value $\mathbf{a} \cdot \mathbf{b}$, then for safety $\|\mathbf{a}\| < 181$ and $\|\mathbf{b}\| < 181$ to avoid overflow. Similarly, to avoid underflow, $\|\mathbf{a}\| > 0.004$ and $\|\mathbf{b}\| > 0.004$.

These suddenly begin to seem like significantly tighter limitations. While issues such as these can be overcome by careful scaling of the data throughout the application, the simple fact is that fixed point requires that programmers keep a very close bound over the required range and precision values of their data throughout the entire application. If the application happens to be a complete and general 3D pipeline, this can be a rather daunting task.

4.4.10 Fixed Point Summary

In today's applications the decision of whether or not to use fixed point is more often than not based entirely on the application's target platform capabilities. Few modern software 3D pipelines choose to use fixed point unless their target platform has no floating point hardware. However, on platforms with fast integer ALUs and no FPUs, fixed point can make the difference between a high-performance, compelling 3D experience and a non-interactive, low frame rate "slide show."

4.5 Floating-Point Numbers

4.5.1 Review: Scientific Notation

In order to better introduce floating point numbers, it is instructive to review the well-known standard representation for real numbers in science and engineering: scientific notation. Computer floating point is very much analogous to scientific notation.

Scientific notation (in its strictest, so-called normalized form) consists of two parts:

1. A decimal number, called the mantissa, such that

$$1.0 \leq |mantissa| < 10.0$$

2. An integer, called the exponent

Together, the exponent and mantissa are combined to create the number:

$$mantissa \times 10^{exponent}$$

Any decimal can be represented in this notation (other than 0, which is simply represented as 0.0), and the representation is unique for each number. In other words, for two numbers written in this form of scientific notation, the numbers are equal if and only if their mantissas and exponents are equal. This uniqueness is a result of the requirements that the exponent be an integer and that the mantissa be "normalized" (i.e., have magnitude between 1.0 and 10.0). Examples of numbers written in scientific notation include

$$\begin{aligned} 102 &= 1.02 \times 10^2 \\ 243{,}000 &= 2.43 \times 10^5 \\ -0.0034 &= -3.4 \times 10^{-3} \end{aligned}$$

Examples of numbers that constitute incorrect scientific notation include

$$\begin{aligned} \text{Incorrect} &= \text{Correct} \\ 11.02 \times 10^3 &= 1.102 \times 10^4 \\ 0.92 \times 10^{-2} &= 9.2 \times 10^{-3} \end{aligned}$$

4.5.2 A Restricted Scientific Notation

To further restrict the standard scientific notation, we will use a special restricted scientific notation, purely for the purpose of introducing the concept of finiteness of representation. We extend the rules for scientific notation:

1. The mantissa must be written with a single, nonzero integral digit.
2. The mantissa must be written with a fixed number of fractional digits (we define as M).

3. The exponent must be written with a fixed number of digits (we define as E).
4. The mantissa and the exponent each have individual signs.

For example, the following number is in a format with $M = 3$, $E = 2$:

$$\pm\boxed{1.1}\boxed{2}\boxed{3} \times 10^{\pm\boxed{1}\boxed{2}}$$

Limiting the number of digits allocated to the mantissa and exponent means that any value that can be represented by this system can be represented uniquely by six decimal digits and two signs. However, this also implies that there are a limited number of values that could ever be represented exactly by this system, namely:

$$\begin{aligned}&(\text{exponents}) \times (\text{mantissas}) \times (\text{exponent signs}) \times (\text{mantissa signs})\\ &= (10^2) \times (9 \times 10^3) \times (2) \times (2)\\ &= 3{,}600{,}000\end{aligned}$$

Note that the leading digit of the mantissa must be nonzero (since the mantissa is normalized), so that there are only nine choices for its value [1, 9], leading to $9 \times 10 \times 10 \times 10 = 9000$ possible mantissas.

This adds finiteness to both the range and precision of the notation. The minimum and maximum exponents are

$$\pm(10^E - 1) = \pm(10^2 - 1) = \pm 99$$

The largest mantissa value is

$$10.0 - (10^{-M}) = 10.0 - (10^{-3}) = 10.0 - 0.001 = 9.999$$

Note that the smallest allowed nonzero mantissa value is still 1.000 due to the requirement for normalization. This format has the following numerical limitations:

Maximum representable value:	9.999×10^{99}
Minimum representable value:	-9.999×10^{99}
Smallest positive value:	1.000×10^{-99}

While one might never use such a restricted form of scientific notation in practice, it demonstrates the basic building blocks of binary floating point, the most commonly used computer representation of real numbers in modern computers.

4.6 Binary "Scientific Notation"

There is no reason that scientific notation must be written in base-10. In fact, in its most basic form, the real number representation known as floating point is similar to a base-2 version of the restricted scientific notation given previously. In base-2, our restricted scientific notation would become

$$SignM \times Mantissa \times 2^{SignE \times Exponent}$$

where $Mantissa$ is a 1-dot-M fixed-point number that is normalized, $Exponent$ is an E-bit integer, and $SignM$ and $SignE$ are independent bits representing the signs of the mantissa and exponent, respectively. Put together, the format involves $M + E + 3$ bits ($M + 1$ for the mantissa, E for the exponent, and two for the signs). Creating an example that is analogous to the preceding decimal case, we analyze the case of $M = 3$, $E = 2$:

$$\pm\boxed{1.0}\boxed{1}\boxed{0} \times 2^{\pm\boxed{0}\boxed{1}}$$

Any value that can be represented by this system can be represented uniquely by 8 bits. The number of values that could ever be represented exactly by this system is

$$\begin{aligned} &(\text{exponents}) \times (\text{mantissas}) \times (\text{exponent signs}) \times (\text{mantissa signs}) \\ &= (2^2) \times (1 \times 2^3) \times (2) \times (2) \\ &= 2^7 = 128 \end{aligned}$$

This seems odd, as an 8-bit number should have 256 different values. However, note that the leading bit of the mantissa must be 1, since the mantissa is normalized (and the only choices for a bit's value are 0 and 1). This effectively fixes one of the bits and cuts the number of possible values in half. We shall see that the most common binary floating-point format takes advantage of the fact that the integral bit of the mantissa is fixed at 1.

In this case, the minimum and maximum exponents are

$$\pm(2^E - 1) = \pm(2^2 - 1) = \pm 3$$

The largest mantissa value is

$$2.0 - 2^{-M} = 2.0 - 2^{-3} = 1.875$$

This format has the following numerical limitations:

$$\begin{aligned} \text{Maximum representable value:} &\quad 1.875 \times 2^3 = 15 \\ \text{Minimum representable value:} &\quad -1.875 \times 2^3 = -15 \\ \text{Smallest positive value:} &\quad 1.000 \times 2^{-3} = 0.125 \end{aligned}$$

From the listed limits, it is quite clear that a floating point format based on this simple 8-bit binary notation would not be useful to most real-world applications. However, it does introduce the basic concepts that are shared by real floating point representations. While there are countless possible floating point formats, the universal popularity of a single set of formats (those described in the IEEE 754 specification [2]) makes it the obvious choice for any discussion of the details of floating point representation. The remainder of this chapter will explain the major concepts of floating point representation as evidenced by the IEEE standard format.

4.7 IEEE 754 Floating-Point Standard

By the early to mid-1970s, scientists and engineers were using floating point very frequently to represent real numbers; at the time, higher-powered computers even included special hardware to accelerate floating point calculations. However, these same scientists and engineers were finding the lack of a floating point standard to be problematic. Their complex (and often very important) numerical simulations were producing different results, depending only upon the make and model of computer upon which the simulation was run. Numerical code that had to run on multiple platforms became riddled with platform-specific code to deal with the differences between different floating point processors and libraries.

In order for cross-platform numerical computing to become a reality, a standard was needed. Over the course of the next decade, a draft standard for floating point formats and behaviors became the *de facto* standard on most floating point hardware. Once adopted, it became known as the IEEE 754 floating point standard [2], and it forms the basis of almost every hardware and software floating point system on the market.

While the history of the standard is fascinating [69], this section will focus on explaining part of the standard itself, as well as using the standard and one of its specified formats to explain the concepts of modern floating-point arithmetic.

4.7.1 Basic Representation

The IEEE standard specifies a 32-bit "single-precision" format for floating-point numbers, as well as a 64-bit "double-precision" format. It is this single-precision format that is of greatest interest for most games and interactive applications and is thus the format which will form the basis of most of the floating point discussion in this text. The two formats are fundamentally similar, so all of the concepts regarding single-precision are applicable to double-precision values as well.

The following diagram shows the basic memory layout of the IEEE single-precision format, including the location and size of the three components of any floating point system: sign, exponent, and mantissa.

Sign	Exponent	Mantissa
1 Bit	8 Bits	23 Bits

The sign in the IEEE floating point format is represented as an explicit bit (the high-order bit). Note that this is the sign of the number itself (the mantissa), not the sign of the exponent. Differentiating between positive and negative exponents is handled in the exponent itself (and is discussed next). The only difference between X and $-X$ in IEEE floating point is the high-order bit. A sign bit of 0 indicates a positive number, and a sign bit of 1 indicates a negative number.

This sign bit format allows for some efficiencies in creating a floating-point math system either in hardware or software. To negate a floating-point number, simply "flip" the sign bit, leaving the rest of the bits unchanged. To compute the absolute value of a floating point number, simply set the sign bit to 0 and leave the other bits unchanged. In addition, the sign bits of the result of a multiplication or division is simply the exclusive "or" of the sign bits of the operands.

As will be seen, this explicit sign bit does lead to the existence of two zero values, one positive and one negative. However, it also simplifies the representation of the mantissa, which is represented as unsigned (positive).

The exponent in this case is stored as a biased number. Biased numbers represent both positive and negative integers (inside of a fixed range) as whole numbers by adding a fixed, positive bias. To represent an integer I, we add a positive bias B (that is constant for the biased format), storing the result as the whole number (nonnegative integer) W. To decode the represented value I from its biased representation W, the formula is simply

$$I = W - B$$

To encode an integer value, the formula is

$$W = I + B$$

Clearly, the minimum integer value that can be represented is

$$I = 0 - B = -B$$

The maximal value that can be represented is related to the maximum whole number that can be represented W_{max}. For example, with an 8-bit biased number, that value is

$$I = W_{max} - B = (2^8 - 1) - B$$

Most frequently, the bias chosen is as close as possible to $W_{max}/2$, giving a range that is equally distributed about zero. Over the course of this chapter, when referring to a biased number, the term *value* will refer to I, while the term *bits* will refer to W.

Such is the case with the IEEE floating point exponent, which uses 8 bits of representation and a bias of 127. This would seem to lead to minimum and maximum exponents of -127 ($= 0 - 127$) and 128 ($= 255 - 127$), respectively. However, for reasons that will be explained, the minimum and maximum values (-127 and 128) are reserved for special cases, leading to an exponent range of $[-126, 127]$. As a reference, these base 2 exponents correspond to base 10 exponents of approximately $[-37, 38]$.

The mantissa is normalized (in almost all cases), as in our discussion of decimal scientific notation (where the units digit was required to have magnitude in the range $[1, 9]$). However, the meaning of "normalized" in the context of a binary system means that the leading bit of the mantissa is always 1. Unlike a decimal digit, a binary digit has only one nonzero value. To optimize storage in the floating point format, this leading bit is omitted, or hidden, freeing all 23 explicit mantissa bits to represent fractional values. To decode the mantissa M (as a 23-bit unsigned integer) into a rational number (ignoring for the moment the exponent), the conversion is

$$1.0 + \frac{M}{2.0^{23}}$$

So, for example, the mantissa bits

$$00000000000000000000000_2 = 0_{10}$$

become the rational number

$$1.0 + \frac{0}{2.0^{23}} = 1.0$$

4.7.2 Range and Precision

The range of single-precision floating point is by definition symmetric, as the system uses an explicit sign bit. With an explicit sign bit, every positive value has a corresponding negative value. This leaves the questions of maximal exponent and mantissa, which when combined will represent the explicit values of greatest magnitude. In the previous section, we found that the maximum base-2 exponent in single-precision floating point is 127. The largest mantissa would be equal to setting all 23 explicit fractional mantissa bits, resulting (along with the implicit 1.0 from the hidden bit) in a mantissa of

$$1.0 + \sum_{i=1}^{23} \frac{1}{2^i} = 1.0 + 1.0 - \frac{1}{2^{23}} = 2.0 - \frac{1}{2^{23}} \approx 2.0$$

The minimum and maximum single-precision floating-point values are then

$$\pm\left(2.0 - \frac{1}{2^{23}}\right) \times 2^{127} \approx \pm 3.402823466 \times 10^{38}$$

The precision of single-precision floating point can be loosely approximated as follows: for a given normalized mantissa, the difference between it and its nearest neighbor is 2^{-23}. To determine the actual spacing between a floating-point number and its neighbor, the exponent must be known. Given an exponent E, the difference between two neighboring single-precision values is

$$\delta_{fp} = 2^E \times 2^{-23} = 2^{E-23}$$

However, we note that in order to represent a value A in single-precision, we must find the exponent E_A such that the mantissa is normalized (i.e., the mantissa M_A is in the range $1.0 \le M_A < 2.0$), or

$$1.0 \le \frac{|A|}{2^{E_A}} < 2.0$$

Multiplying through, we can bound $|A|$ in terms of 2^{E_A}:

$$\begin{aligned} 1.0 &\le \frac{|A|}{2^{E_A}} < 2.0 \\ 2^{E_A} &\le |A| < 2^{E_A} \times 2.0 \\ 2^{E_A} &\le |A| < 2^{E_A+1} \end{aligned}$$

As a result of this bound, we can roughly approximate this entire exponent term 2^{E_A} with $|A|$ and substitute to find an approximation of the distance between neighboring floating-point values around $|A|$ (δ_{fp}) as

$$\delta_{fp} = 2^{E_A - 23} = \frac{2^{E_A}}{2^{23}} \approx \frac{|A|}{2^{23}}$$

From our initial discussion of absolute error, we use general bound on the absolute error equal to half the distance between neighboring representation values:

$$AbsError_A \approx \delta_{fp} \times \frac{1}{2} = \frac{|A|}{2^{23}} \times \frac{1}{2} = \frac{|A|}{2^{24}}$$

This approximation shows that the absolute error of representation in a floating-point number is directly proportional to the magnitude of the value being represented. Having approximated the absolute error, we can approximate the relative error as

$$RelError_A = \frac{AbsError_A}{|A|} \approx \frac{|A|}{2^{24} \times |A|} = \frac{1}{2^{24}} \approx 6 \times 10^{-8}$$

The relative error of representation is thus generally constant, regardless of the magnitude of A. This is the reverse of fixed point, where the absolute error was constant, and the relative error rose in inverse proportion to the values represented. Note that for normalized mantissas, this is *not* true when the value is very close to zero. This will be discussed in detail later.

4.7.3 Arithmetic Operations

The next several sections discuss the basic methods used to perform common arithmetic operations upon floating point numbers. While few users of floating point will ever need to implement these operations at a bitwise level themselves, a basic understanding of the methods is a pivotal step toward being able to understand the limitations of floating point. The methods shown are designed for ease of understanding and do not represent the actual, optimized algorithms that are implemented in hardware.

The IEEE standard specifies that the basic floating point operations of a compliant floating point system must return values that are equivalent to the result computed exactly and *then* rounded to the available precision. The following sections are designed as an introduction to the basics of floating-point operations and do not discuss the exact methods used for rounding the results. At the end of the section, there is a discussion of the programmer-selectable rounding modes specified by the IEEE standard.

The intervening sections include information regarding common issues that arise from these operations, because each operation can produce problematic results in specific situations.

Addition and Subtraction

In order to add a pair of floating point numbers, the mantissas of the two addends must first be shifted such that their radix points are "lined up." In a floating point number, the radix points are aligned if and only if their exponents are equal. If we raise the exponent of a number by one, we must shift its mantissa to the right by one bit. For simplicity, we will first discuss addition of a pair of positive numbers. The standard floating point addition method works (basically) as follows to add two positive numbers $A = S_A \times M_A \times 2^{E_A}$ and $B = S_B \times M_B \times 2^{E_B}$, where $S_A = S_B = 1.0$ due to the current assumption that A and B are nonnegative.

1. Swap A and B if needed so that $E_A \geq E_B$.
2. Shift M_B to the right by $E_A - E_B$ bits. If $E_A \neq E_B$, then this shifted M_B will not be normalized—M_B will be less than 1.0. This is needed to align the radix points.
3. Compute M_{A+B} by adding the shifted mantissas M_A and M_B directly.
4. Set $E_{A+B} = E_A$.
5. The resulting mantissa M_{A+B} may not be normalized (it may have an integral value of 2 or 3). If this is the case, shift M_{A+B} to the right one bit and add 1 to E_{A+B}.

Note that there are some interesting special cases implicit in this method. For example, we are shifting the smaller number's mantissa to the right to align the radix points. If the two numbers differ in exponents by more than the number of mantissa bits, then the smaller number will have all of its mantissa shifted away, and the method will add zero to the larger value. This is important to note, as it can lead to some very strange behavior in applications. Specifically, if an application repeatedly adds a small value to an accumulator, as the accumulator grows, there will come a point at which adding the small value to the accumulator will result in no change to the accumulator's value (the delta value being added will be shifted to zero each iteration)!

Floating point addition must take negative numbers into account as well. There are three distinct cases here:

- Both operands positive. Add the two mantissas as is and set the result sign to positive.

- Both operands negative. Add the two mantissas as is and set the result sign to negative.
- One positive operand, one negative operand. Negate (2's complement) the mantissa of the negative number and add.

In the case of subtraction (or addition of numbers of opposite sign), the result may have a magnitude that is significantly smaller than either of the operands, including a result of zero. If this is the case, there may be considerable shifting required to reestablish the normalization of the result, shifting the mantissa to the left (and shifting zeros into the lowest-precision bits) until the integral bit is 1. This shifting can lead to precision issues (see Section 4.7.6, Catastrophic Cancellation) and can even lead to nonzero numbers that cannot be represented by the normalized format discussed so far (see Section 4.7.5, Very Small Values).

We have purposefully omitted discussion of rounding, as rounding the result of an addition is rather complex to compute quickly. This complexity is due to the fact that one of the operands (the one with the smaller exponent) may have bits that are shifted out of the operation, but still must be considered to meet the IEEE standard of "exact result, then rounded." If the method were simply to ignore the shifted bits of the smaller operand, the result could be incorrect. You may want to refer to [63] for details on the floating point addition algorithm.

Multiplication

Multiplication is actually rather straightforward with IEEE floating point numbers. Once again, the three components that must be computed are the sign, the exponent, and the mantissa. As in the previous section, we will give the example of multiplying two floating point numbers, A and B.

Owing to the fact that an explicit sign bit is used, the sign of the result may be computed simply by computing the exclusive-OR of the sign bits, producing a positive result if the signs are equal and a negative result otherwise. The result of the multiplication algorithm is sign-invariant.

To compute the initial exponent (this initial estimate may need to be adjusted at the end of the method if the initial mantissa of the result is not normalized), we simply sum the exponents. However, since both E_A and E_B contain a bias value of 127, the sum will contain a bias of 254. We must subtract 127 from the result to reestablish the correct bias:

$$E_{A\times B} = E_A + E_B - 127$$

To compute the result's mantissa, we multiply the normalized source mantissas M_A and M_B as 1-dot-23 format fixed-point numbers, producing a (possibly unnormalized) 3-dot-46 result mantissa. Note from the format that the number of integral bits may be 3, as the resulting mantissa could be rounded up to 4.0. Since the source mantissas are normalized, then the resulting mantissa (if it is not 0) must be ≥ 1.0, leading to three possibilities for the mantissa $M_{A\times B}$: it may be normalized, it may be too large by one bit, or it may be too large by two bits. In the latter two cases, we add either 1 or 2 to $E_{A\times B}$ and shift $M_{A\times B}$ to the right by one or two bits until it is normalized.

Rounding Modes

The IEEE specification defines four rounding modes that an implementation must support. These rounding modes are

- Round toward 0
- Round toward $-\infty$
- Round toward ∞
- Round toward nearest

The specification defines these modes with specific references to bitwise rounding methods that we will not discuss here, but the basic ideas are quite simple. We break the mantissa into the part that can be represented (the leading 1 along with the next 23 most-significant bits), which we call M, and the remaining lower-order bits, which we call R. Round toward 0 is also known as "chopping" and is the simplest to understand; in this mode, M is used and R is simply ignored, or "chopped off." Round toward $\pm\infty$ are modes that round toward positive (∞) or negative ($-\infty$) based on the sign of the result and whether $R = 0$ or not, as shown in the following table:

Mode	**Round toward $-\infty$**		**Round toward ∞**	
M and R	$R = 0$	$R \neq 0$	$R = 0$	$R \neq 0$
$M \geq 0$	M	M	M	$M + 1$
$M < 0$	M	$M + 1$	M	M

4.7.4 Special Values

One of the most important parts of the IEEE floating point specification is its definition of numerous special values. While these special values co-opt bit patterns that would otherwise represent specific floating point numbers, this trade-off is accepted as worthwhile, owing to the nature and importance of these special values.

Zero

The representation of 0.0 in floating point is more complex than one might think. Since the high-order bit of the mantissa is assumed to be 1 (and has no explicit bit in the representation), it is not enough to simply set the 23 explicit mantissa bits to zero, as that would simply represent the number $1.0 \times 2^{Exponent-127}$. It is necessary to define zero explicitly, in this case as a number whose exponent bits are all 0 *and* whose explicit mantissa bits are 0. This is sensible, as this value would otherwise represent the smallest possible normalized value. Note that the exponent bits of 0 map to an exponent value of -127, which is reserved for special values such as zero. All other numbers with exponent value -127 (i.e., those with nonzero mantissa bits) are reserved for a class of very small numbers called "denormals," which will be described later.

Another issue with respect to floating point zero arises from the fact that IEEE floating point numbers have an explicit sign bit. The IEEE specification defines both positive and negative 0, differentiated by only the sign bit. To avoid very messy code, the specification does require that floating point comparisons of positive zero to negative zero return "equal." However, the bitwise representations are distinct, which means that applications should never use bitwise equality tests with floating point numbers! The bitwise representations of both zeros are

$+0.0 =$

0	00000000	00000000000000000000000
S	Exponent	Mantissa

$-0.0 =$

1	00000000	00000000000000000000000
S	Exponent	Mantissa

The standard does list the behavior of positive and negative zero explicitly, including the definitions:

$$(+0) - (+0) = (+0)$$
$$-(+0) = (-0)$$

Also, the standard defines the sign of the result of a multiplication or division operation as negative if and only if exactly one of the signs of the operands is negative. This includes zeros. Thus,

$$(+0)(+0) = +0$$
$$(-0)(-0) = +0$$

$$
\begin{aligned}
(-0)(+0) &= -0 \\
(-0)P &= -0 \\
(+0)P &= +0 \\
(-0)N &= +0 \\
(+0)N &= -0
\end{aligned}
$$

where $P > 0$ and $N < 0$.

Infinity

At the other end of the spectrum from zero, the standard also defines positive infinity (∞_{fp}) and negative infinity ($-\infty_{fp}$), along with rules for the behavior of these values. In a sense the infinities are not pure mathematical values. Rather, they are used to represent values that fall outside of the range of valid exponents. For example, 1.0×10^{38} is just within the range of single-precision floating point, but in single-precision:

$$(1.0 \times 10^{38})^2 = 1.0 \times 10^{76} \approx \infty_{fp}$$

The behavior of infinity is defined by the standard as follows (the standard covers many more cases, but these are representative):

$$
\begin{aligned}
\infty_{fp} - P &= \infty_{fp} \\
\frac{P}{\infty_{fp}} &= +0 \\
\frac{-P}{\infty_{fp}} &= -0
\end{aligned}
$$

where

$$0 < P < \infty_{fp}$$

The bitwise representations of $\pm\infty_{fp}$ use the reserved exponent value 128 and all explicit mantissa bits zeros. The only difference between the representations of the two infinities is, of course, the sign bit. The representations are diagrammed as follows:

$\infty_{fp} =$

0	11111111	00000000000000000000000
S	Exponent	Mantissa

$-\infty_{fp} =$

1	11111111	00000000000000000000000
S	Exponent	Mantissa

Floating-point numbers with exponent values of 128 and nonzero mantissa bits do not represent infinities. They represent the next class of special values, nonnumerics.

Nonnumeric Values

All the following function call examples represent exceptional cases:

Function Call	Issue
arcsine(2.0)	Function not defined for argument
sqrt(−1.0)	Result is imaginary
0.0/0.0	Result is indeterminate
$\infty - \infty$	Result is indeterminate

In each of these cases, none of the floating-point values we have discussed will accurately represent the situation. Here we need a value that indicates the fact that the desired computation cannot be represented as a real number. The IEEE specification includes a special pair of values for these cases, known collectively as Not a Number (NaNs). There are two kinds of NaNs: quiet (or silent) NaN (QNaN) and signaling NaN (SNaN). Compare the following representations:

QNaN =

0	11111111	1[22 low-order bits indeterminate]
S	Exponent	Mantissa

SNaN =

0	11111111	0[22 low-order bits not all 0]
S	Exponent	Mantissa

Quiet Not a Numbers (Kahan [69] simply calls them NaNs) represent indeterminate values and are quietly passed through later computations (generally as QNaNs). They are not supposed to signal an exception, but rather allow floating point code to return the fact that the result of the desired operation was indeterminate. Floating point implementations (hardware or software) will generate QNaNs in cases such as those in our comparison.

SNaNs represent unrecoverable mathematical errors and signal an exception. Most FPUs are designed not to generate SNaNs — the original idea was that authors of high-level software math packages could generate them in terminal situations. In addition, compilers could (in debugging builds) set all floating point values to SNaN, ensuring an exception if the programmer left the values uninitialized. The realities of compilers and operating systems make SNaNs less interesting. There have been issues in the support for SNaNs in current compilers [69], resulting in SNaNs being encountered very rarely.

4.7.5 Very Small Values

Normalized Mantissas and the "Hole at Zero"

One side effect of the normalized mantissa is very interesting behavior near zero. To better understand this behavior, let us look at the smallest normalized value (we will look at the positive case; the negative case is analogous) in single-precision floating point, which we will call F_{min}. F_{min} would have an exponent of -126 and zeros in all explicit mantissa bits. The resulting mantissa would have only the implicit units bit set, leading to a value of

$$F_{min} = 2^0 \times 2^{-126} = 2^{-126}$$

The largest value smaller than this in a normalized floating-point system would be 0.0. However, the smallest value larger than F_{min} would differ by only one bit from F_{min} — the least-significant mantissa bit would be set. This value, which we will call F_{next} would be simply:

$$F_{next} = (2^0 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149} = F_{min} + 2^{-149}$$

This leads to a rather interesting situation: the distance between F_{min} and its nearest smaller neighbor (0.0) is 2^{-126}. This distance is much larger than the distance between F_{min} and its nearest *larger* neighbor, F_{next}. The distance between F_{min} and F_{next} is only

$$F_{next} - F_{min} = 2^{-149}$$

In fact, F_{min} has a sequence of approximately 2^{23} larger neighbors that are each a distance of 2^{-149} from the previous. This leaves a large "hole" of numbers between 0.0 and F_{min} that cannot be represented with nearly the accuracy as the numbers slightly larger than F_{min}. This gap in the representation is often referred to as the "hole at zero." The operation of representing numbers in the range $(-F_{min}, F_{min})$ with zero is often called "flushing to zero."

One problem with flush-to-zero is that the subtraction of two numbers that are not equal can result in zero. In other words, with flush-to-zero

$$A - B = 0 \not\Rightarrow A = B$$

How can this be? See the following example:

$$A = 2^{-126} \times (2^0 + 2^{-2} + 2^{-3})$$
$$B = 2^{-126} \times (2^0)$$

Both of these are valid single-precision floating point numbers. In fact, they have equal exponents: -126. Clearly, they are also not equal floating point numbers: A's mantissa has two additional 1 bits. However, their subtraction produces:

$$\begin{aligned} A - B &= (2^{-126} \times (2^0 + 2^{-2} + 2^{-3}) - (2^{-126} \times (2^0)) \\ &= 2^{-126} \times ((2^0 + 2^{-2} + 2^{-3}) - (2^0)) \\ &= 2^{-126} \times (2^{-2} + 2^{-3}) \\ &= 2^{-128} \times (2^0 + 2^{-1}) \end{aligned}$$

which would be returned as zero on a flush-to-zero floating point system. While this is a contrived example, it can be seen that any pair of nonequal numbers whose difference has a magnitude less than 2^{-126} would demonstrate this problem. There is a solution to this and other flush-to-zero issues, however. The solution is known as "gradual underflow," and it is discussed in the next section.

Denormals and Gradual Underflow

The IEEE specification specifies behavior for very small numbers that avoids this so-called hole at zero. The behavior is known as gradual underflow, and this gradual underflow generates values called "denormals," or "denormalized numbers."

The idea is quite simple. Rather than require every floating point number to be normalized, the specification reserves numbers with nonzero explicit mantissa bits and an exponent of -127 for denormals. In a denormal, the implicit high-order bit of the mantissa is 0. This allows numbers with magnitude smaller than 1.0×2^{-126} to be represented. In a denormal, the exponent is assumed to be -126 (even though the actual bits would represent -127), and the mantissa is in the range $[\frac{1}{2^{23}}, 1 - \frac{1}{2^{23}}]$. The smallest nonzero value that can be represented with a denormal is $2^{-23} \times 2^{-126} = 2^{-149}$, filling in the "hole at zero." Note that all nonzero floating point values are still unique, as the specification only allows denormalized mantissas when the exponent is -126, the minimum valid exponent.

As an historical note, gradual underflow and denormalized value handling were perhaps the most hotly contested of all sections in the IEEE floating point specification. Flush-to-zero is much simpler to implement in hardware, which also tends to mean that it performs faster and makes the hardware cheaper to produce. When the IEEE floating point standard was being formulated in the late 1970s, several major computer manufacturers were using the flush-to-zero method for dealing with underflow. Changing to the use of gradual underflow required these manufacturers to design FPU hardware or software that could handle the unnormalized mantissas that are generated by denormalization. This would lead either to more complex FPU hardware or a system that emulated some or all of the denormalized computations in software or microcode. The former could make the FPUs more expensive to produce, while the latter could lead to greatly decreased performance of the floating point system when denormals are generated. However, several manufacturers showed that it could be implemented in floating point hardware, paving the way for this more accurate method to become part of the *de facto* (and later, official) standard. However, performance of denormalized values is still an issue, even today. We will discuss a real-world example of denormal performance on a modern FPU in Section 4.8.2.

4.7.6 Catastrophic Cancelation

We have used relative error as a metric of the validity of the floating point representation of a given number. However, the relative representation errors of the operands to a floating point addition or subtraction operation may not accurately represent the error in the result. The addition or subtraction of a pair of floating point numbers can lead to a result with magnitude much smaller than either of the operands. Specifically, the subtraction of two nearly equal (but different) values will result in such a situation. The following example shows how the subtraction of two numbers of large magnitude can result in a value with much lower magnitude. In this case, the source operands and the result are represented exactly, but as we shall see in a very similar case, the result is more problematic:

$$\begin{aligned}
A_{fp} &= 8,388,609 = 2^{23} \times (2^0 + 2^{-23}) \\
B_{fp} &= 8,388,608 = 2^{23} \times (2^0) \\
A_{fp} - B_{fp} &= (2^{23} \times (2^0 + 2^{-23})) - (2^{23} \times (2^0)) \\
&= 2^{23} \times ((2^0 + 2^{-23}) - (2^0)) \\
&= 2^{23} \times 2^{-23} = 2^0 = 1
\end{aligned}$$

While the result is represented exactly, note that in the last step of the operation, the value must be renormalized. Zeros are shifted into all 23 of the low-order (explicit) mantissa bits (i.e., only the integral bit is 1). In this case the result is correct. As an example of how this process can cause catastrophic cancellation and large relative error, let us analyze the following case. It is very similar to the previous example, but replaces A and B with values that cannot be represented exactly in single-precision floating point:

$$\begin{aligned} A &= 8,388,609.45 \\ B &= 8,388,607.75 \\ A - B &= 1.7 \end{aligned}$$

In single-precision floating point, these values round to the same A_{fp} and B_{fp} values just given. In turn, $A_{fp} - B_{fp}$ is once again 1.0. First, we analyze the relative representation error of A_{fp}:

$$Error_A = \left|\frac{A - A_{fp}}{A}\right| = \frac{8,388,609.45 - 8,388,609}{8,388,609.45} \approx 5.4 \times 10^{-8}$$

which is, by itself, a very small relative error. Similarly, we compute the representation error of B_{fp}:

$$Error_B = \left|\frac{B - B_{fp}}{B}\right| = \frac{8,388,607.75 - 8,388,608}{8,388,607.75} \approx 3.0 \times 10^{-8}$$

an even smaller relative error. However, the overall error in the subtraction $A_{fp} - B_{fp}$ versus the exact $A - B$ is

$$Error_{A-B} = \left|\frac{(A - B) - (A_{fp} - B_{fp})}{A - B}\right| = \frac{1.7 - 1.0}{1.7} \approx 0.41!$$

The relative error in the overall result is about 10,000,000 times worse than the representational error in either A_{fp} or B_{fp}! This is due to the fact that almost all of the bits of precision in the two numbers matched. In other words, all but one of the original mantissa bits in A_{fp} and B_{fp} were canceled out in the subtraction, leaving the least significant bit of the operands as the *most* significant bit of the result. None of the 23 explicit (fractional) bits of the result's mantissa is actual data—they were simply shifted in as zeros. The precision of such a result is very low, indeed. This is catastrophic cancelation; the significant bits are all canceled, causing a catastrophically large growth in the representation error of the result.

The best way to handle catastrophic cancelation in a floating point system is to avoid it. Numerical methods that involve computing a small value as the subtraction or addition of two potentially large values should be reformulated to remove the operation. An example of a common numerical method that uses such a subtraction is the well-known quadratic formula:

$$\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Both of the subtractions in the numerator can involve large numbers whose addition/subtraction can lead to small results. However, refactoring of the formula can lead to better-conditioned results. The following revised version of the quadratic formula can be used in cases where computation of one of the two roots involves subtracting nearly equal values. The refactored formula avoids cancelation by replacing the subtraction with an addition:

$$\frac{2C}{-B \mp \sqrt{B^2 - 4AC}}$$

A root that would be computed with a subtraction in the first ("classic") version of the quadratic formula may be computed with an addition in the second version, and vice versa.

4.7.7 Double Precision

As mentioned, the IEEE 754 specification supports a 64-bit "double-precision" floating point value, known in C/C++ as the intrinsic `double` type. The format is completely analogous to the single-precision format, with the following bitwise layout:

Sign	Exponent	Mantissa
1 Bit	11 Bits	52 Bits

Double-precision values have a range of approximately 10^{308} and can represent values smaller than 10^{-308}. A programmer's common response to the onset of precision or range issues is to switch their code to use double-precision floating point values in the offending section of code (or sometimes even throughout the entire system). While double precision can solve almost all range issues and many precision issues (though catastrophic cancelation can still persist) in interactive 3D applications, there are several drawbacks that should be considered prior to its use:

- Memory. Since double-precision values require twice the storage of single-precision values, memory requirements for an application can grow quickly, especially if arrays of vectors (such as vertices) must be stored as double-precision.

- Performance. At least some of the operations on most hardware FPUs are significantly slower when computing double precision results. Additional expense can be incurred for conversion between single- and double-precision values.
- Platform issues. Not all platforms (especially game-centric platforms) support double precision.

4.8 Real-World Floating Point

While the IEEE floating-point specification does set the exact behavior for a wide range of the possible cases that occur in real-world situations, in real-world applications on real-world platforms, the specification cannot tell the entire story. The following sections will discuss some issues that are of particular interest to 3D game developers.

4.8.1 Internal FPU Precision

Some readers will likely try some of the exceptional cases themselves in small test applications. In doing so, they are likely to find surprising behavior in many situations. For example, examine the following code:

```
main()
{
    float fHuge = 1.0e30f; // valid single-precision
    fHuge *= 1.0e38f; // result = infinity
    fHuge /= 1.0e38f; // ????
}
```

Stepping in a debugger, the following will happen on many major compilers and systems:

1. After the initial assignment, `fHuge = 1.0e30`, as expected.
2. After the multiplication, `fHuge =` ∞_{fp}, as expected.
3. After the division, `fHuge = 1.0e30`!

This seems magical. How can the system divide the single value ∞_{fp} and get back the original number? A look at the assembly code gives a hint. The basic steps the compiler generates are as follows:

1. Load `1.0e30` and `1.0e38` into the FPU.
2. Multiply the two loaded values and return ∞_{fp}, keeping the result in the FPU as well.
3. Divide the previous result (still in the FPU) by `1.0e38` (still in the FPU), returning the correct result.

The important item to note is that the result of each computation was both returned and kept in the FPU for later computation. This step is where the apparent "magic" occurs. The FPU (as per the IEEE standard) uses high-precision (sometimes as long as `long double`) registers in the FPU. The conversion to single-precision happens during the storing from the FPU into memory. While the returned value in `fBig` was indeed ∞_{fp}, the value retained in the FPU was higher-precision and was the correct value, `1.0e68`. When the division occurs, the result is correct, not ∞_{fp}.

However, an application cannot count on this result. If the FPU had to flush the intermediate values out of its registers, then the result of the three lines above would have been quite different. For example, if significant floating point work had to be computed between the above multiplication and the final division, the FPU might have run out of registers and had to evict the high-precision version of `fHuge`. This can lead to odd behavior differences, sometimes even between optimized and debugging builds of the same source code.

4.8.2 Performance

The IEEE floating point standard specifies behavior for floating point systems; it does not specify information regarding performance. Just because a floating point implementation is correct does not mean that it is fast. Furthermore, the speed of one floating point operation (e.g., addition) does not imply much about the speed of another (e.g., square root). Finally, not all input data are to be considered equal in terms of performance. The following sections describe examples of some real-world performance pitfalls found in floating point implementations.

Performance of Denormalized Numbers

During the course of creating a demo for a major commercial 3D game engine, one of the authors found that in some conditions, the performance of the demo dropped almost instantaneously by as much as 20 percent. The code was profiled and it was found that one section of animation code was suddenly running 10–100 times slower than in the previous frames. An examination of the offending code determined that it consisted of nothing more than basic floating point operations, specifically, multiplications and divisions. Moreover, there were no loops in the code, and the number of calls to the code was not increasing. The code itself was simply taking 10–100 times longer to execute.

Further experiments outside of the demo found that a fixed set of input data (captured from tests of the demo) could always reproduce the problem. The developers examined the code more closely and found that very small nonzero values were creeping into the system. In fact, these numbers were denormalized. Adjusting the numbers by hand even slightly outside of the range of denormals and into normalized floating-point values instantly returned the performance to the original levels. The immediate thought was that exceptions were causing the problem. However, all floating point exceptions were disabled (masked) in the test application.

To verify the situation, they wrote an extremely simple test application. Summarized, it was as follows:

```
float TestFunction(float fValue)
{
    return fValue;
}

main()
{
    int i;
    float fTest;
    // Start "normal" timer here
    for (i = 0; i < 10000; i++)
    {
        // 1.0e-36f is normalized in single-precision
        fTest = TestFunction(1.0e-36f);
    }
    // End "normal" timer here
    // Start "denormal" timer here
    for (i = 0; i < 10000; i++)
    {
        // 1.0e-40f is denormalized in single-precision
        fTest = TestFunction(1.0e-40f);
    }
    // End "denormal" timer here
}
```

Having verified that the assembly code generated by the optimizer did indeed call the desired function the correct number of times with the desired arguments, they found that the denormal loop took 30 times as long as the normal loop (even with exceptions masked). A careful reading of Intel's performance recommendations [64] for the Pentium series of CPUs found that *any* operation (including simply loading to a floating point register) that produced or accepted as an operand a denormal value was run using so-called assist microcode, which is known to be much slower than standard FPU instructions. Intel's recommendation was for high-performance code to manually clamp small values to zero as need be.

Intel had followed the IEEE 754 specification, but had made the design decision to allow exceptional cases such as denormals to cause very significant performance degradation. An application that had not known of this slowdown on the Pentium processor may have avoided manually clamping small values to zero, out of fear of slowing the application down with extra conditionals. However, armed with this processor-specific information, it was much easier to justify clamping small numbers that were not already known to be normal. Since the values in question were normalized 4-vectors, the overall length of the vector value should be 1.0. As a result, it was more than safe to clamp small values to zero.

Software Floating Point Emulation

Applications should take extreme care on new platforms to determine whether or not the platform supports hardware-assisted floating point. In order to ensure that code from other platforms ports and executes without major rewriting, some compilers supply software floating point emulation libraries for platforms that do not support floating point in hardware. This is especially common on popular embedded and handheld chip sets such as Intel's StrongARM and XScale processors [64]. These processors have no FPUs, but C/C++ floating point code compiled for these devices will generate valid, working emulation code. The compilers will often do this silently, leaving the uninformed developer with a working program that exhibits horrible floating point performance, in some cases hundreds of times slower than could be expected from a hardware FPU.

It's worth reiterating that not all FPUs support both single- and double-precision. Some major game consoles, for example, will generate FPU code for single-precision values and emulation code for double-precision values. As a result, careless use of double precision can lead to much slower code. In fact, it is important to remember that double precision can be introduced into an expression in subtle ways. For example, remember that in C/C++, floating point constants are double-precision by default, so whenever possible, explicitly specify constants as single-precision, using the `f` suffix.

The difference between double- and single-precision performance can be as simple as `1.0` instead of `1.0f`.

4.8.3 IEEE Specification Compliance

While major floating point errors in modern processors are relatively rare (even Intel was caught off guard by the magnitude of public reaction to what it considered minor and rare errors in the floating-point divider on the original Pentium chips), this does not mean that it is safe to assume that all floating point units in modern CPUs are always fully compliant to IEEE specifications and support both single and double precision. The greatest lurking risk to modern developers assuming full IEEE compliance are conscious design decisions, not errors on the part of hardware engineers. However, in most cases, for the careful and attentive programmer, these new processors offer the possibilities of great performance increases to 3D games.

As more and more FPUs are designed and built for multimedia and 3D applications (rather than the historically important scientific computation applications for which earlier FPUs were designed), manufacturers are starting to deviate from the IEEE specification, optimizing for high-performance over accuracy. This is especially true with respect to the "exceptional" cases in the spec, such as denormals, infinity, and Not a Number.

Hardware vendors make the argument that while these special values are critically important to scientific applications, for 3D games and multimedia, they generally occur only in error cases that are best handled by avoiding them in the first place.

Intel's SSE

An important example of such design decisions involves Intel's Streaming SIMD Extensions (SSE) [64], a new coprocessor that was added to the Pentium series with the advent of the Pentium III. The coprocessor is a special vector processor that can execute parallel math operations on four floating point values, packed into a 128-bit register. The SSE instructions were specifically targeted at 3D games and multimedia, and this is evident from even a cursory view of the design. Several design decisions related to the special-purpose FPU merit mentioning here:

- The original SSE (Pentium III) instructions can only support 32-bit floating point values, not doubles.
- Denormal values can be (optionally) rounded to zero ("flushed to zero"), disabling gradual underflow.

- Full IEEE 754 behavior can be supported as an option but at less than peak performance.

3D-specific FPUs

Other platforms have created graphics-centric FPUs. This 3D graphics focus has given the hardware designers the ability to optimize the floating point behavior of the FPUs very heavily. Unburdened by the need to support any applications other than games, the designers of these FPUs have taken things a step further than Intel's SSE instructions by making the deviations from the IEEE specification permanent, rather than optional.

AMD's 3DNow! [1] extensions to its x86 platforms are one such example. While leaving the main FPU unchanged, AMD added hardware to support up to four floating point instructions per clock cycle. As a further optimization, 3DNow! made some decisions that broke from the IEEE specification, including:

- Cannot accept infinity or NaN as operands
- Generates the maximal normal floating-point value on overflow, rather than infinity
- Flush-to-zero as the only form of underflow (no denormals)
- No support for floating point exceptions

The 3D-centric vector FPUs in some current game consoles have taken similar paths. These differences from the IEEE specification, while severe from a scientific computing perspective, are rarely an issue in correct 3D game code. The console processors that have these limitations are generally designed to allow games to implement geometry pipelines. In most 3D game code, the engine programmer takes great pains to avoid exceptional conditions in their geometry pipelines. Thus, these hardware design decisions tend to merely reflect the common practices of game programmers, rather than adding new limitations upon them.

4.9 Code

Source Code
Library
IvMath

While this text's companion CD-ROM and web site do not include specific code that demonstrates the concepts in this chapter, source code that deals with issues of floating point representation may be found throughout the math library `IvMath`. For example, the source code for `IvMatrix33`, `IvMatrix44`, `IvVector3`, `IvVector4`, and `IvQuat` includes sections of code that avoid denormalized numbers and comparisons to exact floating-point zero.

CPU chipset manufacturers Intel and AMD have been focused on 3D graphics and game performance and have made public many code examples, presentations, and software libraries that detail how to write high-performance floating point code for their processors. Many of these resources may be found on their developer web sites ([1, 64]).

As mentioned earlier, high-performance fixed point code that does not drop precision or overflow in common cases is best accomplished through the use of platform-specific instructions. Major CPU vendors have realized this, and some of them ship libraries or sample code that allow high-performance fixed-point math on their non–floating-point processors. On their developer web site Intel [64] provides their `GPP` library — a set of "graphics performance primitives" that includes basic fixed point math routines — optimized for their StrongARM and XScale processors. Also, the ARM [4] corporation includes technical reports on their developer web site that include code and methods for implementing high-speed fixed point code on devices based on their architecture.

4.10 Chapter Summary

In this chapter we have discussed the details of how computers represent the sets of whole numbers, integers, and real numbers. Each of these representations has inherent limitations that any serious programmer must understand in order to use them efficiently and correctly. The common representations of real numbers, both fixed-point and floating-point, present the most subtle limitations, especially the issues of limited precision. We have also discussed the basics of error metrics for number representations.

Hopefully, this chapter has instilled two important pieces of information in the reader. The first and most basic is an understanding of the inner workings of the number systems that pervade 3D games. This should allow the programmer to truly comprehend the reasons *why* their math-related code behaves (or, more importantly, why it *misbehaves*) as it does. The second piece of information is an appreciation of why one should pay attention to the topic of floating point representation in the first place — namely, to better prepare the 3D game developer to do what they will need to do at some point in the development of a game: optimize or fix a section of slow or incorrect math code. Better yet, it can assist the developer to avoid writing this potentially problematic code in the first place.

For further reading, Kahan's papers on the history and status of the IEEE floating point standard ([69] and related papers and lectures by Kahan, available from the same source) offer fascinating insights into the background of modern floating point computation. In addition, back issues of *Game Developer* magazine (such as [60]) provide frequent discussion of number representations as they relate to computer games.

Part II

RENDERING

Chapter 5
Viewing and Projection

5.1 Introduction

In previous chapters we've discussed how to represent objects, basic transformations we can apply to these objects, and how we can use these transformations to move and manipulate our objects within our virtual world. With that background in place, we can begin to discuss the mathematics underlying the techniques we use to display our game objects on a monitor or other visual display medium.

It doesn't take much justification to understand why we might want to view the game world—after all, games are primarily a visual media. Other sensory outputs are of course possible, particularly sound and haptic (or touch) feedback. Both have become more sophisticated and in their own way provide another representation of the relative 3D position and orientation of game objects. But in the current market, when we think of games, we first think of what we can see.

The first part of the display process (or *graphics pipeline*) involves setting up a virtual viewer or camera, which allows us to control which objects lie in our current view. As we'll see, this camera is just like any other object in the game; we can set the camera's position and orientation based on an affine transformation. Inverting this transformation allows us to transform objects in the world frame into the point of view of the camera object.

From there we will want to transform our objects in view into 2D coordinates so they can be represented in an image. This flattening or projection takes many forms, and we'll discuss several of the most commonly used projections. In particular we'll derive perspective projection, which mimics our viewpoint of the real world most closely. Once projected, we can take the coordinates generated and stretch and translate them to fit a specific portion of the screen, known as the viewport.

Finally, we'll cover how to reverse this process so we can take a mouse click on our two-dimensional screen and use it to select objects in our three-dimensional world. This process, known as "picking," can be useful when building an interface with three-dimensional elements. For example, selecting units in a 3D real-time strategy game is done via picking.

As with other chapters, we'll be discussing how to implement these transformations in production code. Because our examples are written in OpenGL, for the most part we'll be focusing on its pipeline and how it handles the viewing and projective transformations. However, we will also cover the cases where it may differ from graphics APIs, particularly Direct3D.

5.2 The View Frame and View Transformation

5.2.1 Defining a Virtual Camera

In order to render objects in the world, we need to represent the notion of a viewer. This could be the main character's viewpoint in a first-person shooter, or an over-the-shoulder view in a third-person adventure game, or it could be a zoomed-out wide shot in a strategy game. We may want to control properties of our viewer to simulate a virtual camera; for example, we may want to create an in-game scripted sequence where we pan across a screen or follow a set path through a space. We encapsulate these properties into a single entity, commonly called the *camera*.

For now, we'll consider only the most basic properties of the camera needed for rendering. We are trying to answer two questions: Where am I? and Where am I looking? [11]. The answer to the first question is the camera's position, E, which is variously called the *eyepoint*, the *view position*, or the *view space origin*. As we mentioned, this could be the main character's eye position, a location over his shoulder, or pulled back from the action. While this can be placed relative to another object's location, it is usually cleaner and easier to manage if we represent it in the world frame.

A partial answer to the second question is a vector called the *view direction vector*, or $\mathbf{v}_{dir}$, which points along the facing direction for the camera. This could be a vector from the camera position to an object or point of interest, a vector indicating the direction the main character is facing, or a fixed direction if we're trying to simulate an isomorphic view for a strategy game.

For the purposes of setting up the camera, this is also specified in the world frame.

Since there is an infinite number of orientations which align with a single vector, the view direction vector is not enough information. To constrain our possibilities down to one, we specify a second vector orthogonal to the first, called the *view up vector*, or $\mathbf{v}_{up}$. This indicates the direction out of the top of the camera or the character's head. The remaining orthogonal vector is the *view side vector*, or $\mathbf{v}_{side}$, which usually points out towards the camera's right.

All three view vectors are represented in the world frame. Since they are orthogonal, by normalizing them we can create an orthonormal basis. Using this basis together with the view position we can specify a new frame relative to our world coordinate system, known as the *view frame*, or *view space* (Figure 5.1). This is how we determine our camera's position and orientation in the world.

We can of course define the transformation from the view frame to the world frame (also known as the view-to-world transformation) as a 4×4 affine matrix. The origin E of the view frame is translated to the view position, so the translation vector $\mathbf{y}$ is equal to $E - O$. We'll abbreviate this as $\mathbf{v}_{pos}$. Similarly, the view vectors represent how the standard basis vectors in view space are transformed into world space and become columns in the upper left 3×3 matrix $\mathbf{A}$. To build $\mathbf{A}$, however, we need to define which standard basis vector in the view frame maps to a particular view vector in the world frame.

The standard order used by most viewing systems is to map the view frame z-axis to the view direction vector, the view frame y-axis to the view up vector, and the view frame x-axis to the view side vector (Figure 5.2a). This aligns our view coordinates so that in the view frame, x values vary left and right along the plane of the screen and y values vary up and down. In addition, as objects in front of the viewer move farther away, their z values in the view frame will

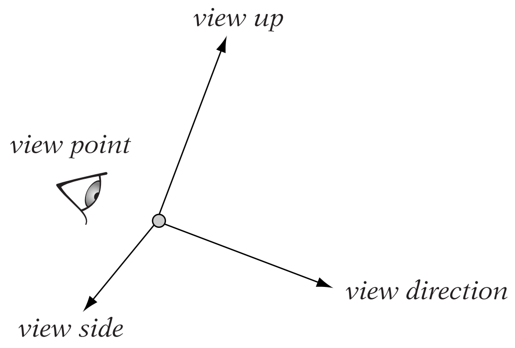


FIGURE 5.1 View frame relative to the world frame.

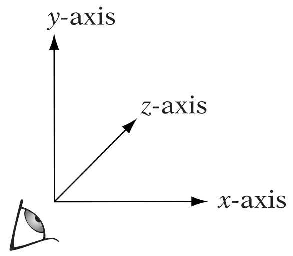


FIGURE 5.2a Standard view frame axes.

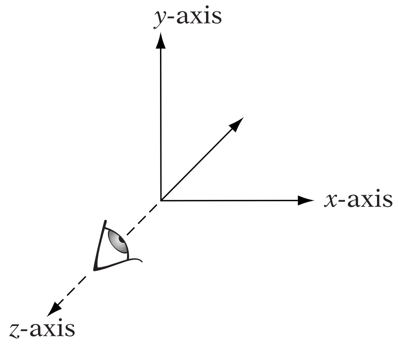


FIGURE 5.2b OpenGL view frame axes.

increase, which is nicely intuitive. The value of z can act as a measure of the distance between the object and the camera, which we can use for hidden object removal.

This mapping indicates which columns the view vectors should be placed in, and the view position takes its familiar place in the right-most column. The corresponding transformation matrix is

$$\mathbf{M}_{view\rightarrow world} = \begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & \hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

Note that in this case we are mapping from a left-handed view frame ($(\hat{\mathbf{v}}_{side} \times \hat{\mathbf{v}}_{up}) \cdot \hat{\mathbf{v}}_{dir} < 0$) to the right-handed world frame, so the upper 3×3 is not a pure rotation but a rotation concatenated with a reflection.

OpenGL does not follow the standard model; instead, it chooses a slightly different approach. It maintains a right-handed system where the view direction is aligned with the frame's negative z-axis (Figure 5.2b). So in this case, the farther away the object is, its $-z$ coordinate gets larger in the view frame.

The corresponding transformation matrix for OpenGL is

$$\mathbf{M}_{view\rightarrow world}\begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & -\hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

In this case, since we are mapping from a right-handed frame to a right-handed frame, no reflection is necessary, and the upper 3×3 matrix is a pure rotation. Not having a reflection can actually be a benefit, particularly with some culling methods.

5.2.2 Controlling the Camera

It's not enough that we have a specification for our camera position and orientation. More often we'll want to move it around the world. Positioning our camera is a simple enough matter of translating the view position, but controlling view orientation is another problem. One way is to specify the view vectors directly and build the matrix as described. This assumes, of course, that we already have a set of orthogonal vectors we want to use for our viewing system.

Source Code
Demo
LookAt

The more usual case is that we only know the view direction. For example, suppose we want to continually focus on a particular object in the world (known as the look-at object). We can construct the view direction by subtracting the view position from the object's position. But whether we have a given view direction or we generate it from the look-at object, we still need two other orthogonal vectors to properly construct an orthogonal basis. We can calculate them by using one additional piece of information: the world up vector. This is a fixed vector representing the direction "up" in the world frame. In our case we'll use the z-axis basis vector $\mathbf{k}$ (Figure 5.3), although in general any vector that we care to call "up" will do. For example, suppose we had a mission on a boat at sea and wanted to give the impression that the boat was rolling from side to side, without affecting the simulation. One method is to change the world up vector over time, oscillating between two keeled-over orientations, and use that to calculate your camera orientation.

For now, however, we'll use $\mathbf{k}$ as our world up vector. Our goal is to compute orthonormal vectors in the world frame corresponding to our view vectors, such that one of them is our view direction vector $\hat{\mathbf{v}}_{dir}$, and our view up vector $\hat{\mathbf{v}}_{up}$ matches the world up vector as closely as possible. Recall that we can use Gram-Schmidt orthogonalization to create orthogonal vectors from a set of nonorthogonal vectors, and so:

$$\mathbf{v}_{up} = \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\hat{\mathbf{v}}_{dir}$$

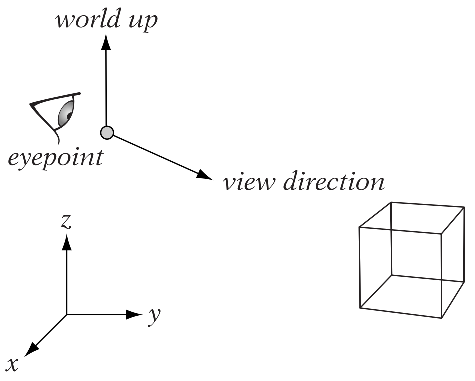


FIGURE 5.3 LookAt representation.

Normalizing gives us $\hat{\mathbf{v}}_{up}$. We can take the cross product to get the view side vector:

$$\hat{\mathbf{v}}_{side} = \hat{\mathbf{v}}_{dir} \times \hat{\mathbf{v}}_{up}$$

We don't need to normalize in this case because the two vector arguments are orthonormal. The resulting vectors can be placed as columns in the transformation matrix as before.

One problem may arise if we are not careful: suppose that $\hat{\mathbf{v}}_{dir}$ and $\mathbf{k}$ are parallel? If they are equal we end up with

$$\begin{aligned}\mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - 1 \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0}\end{aligned}$$

If they point in opposite directions we get

$$\begin{aligned}\mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - (-1) \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0}\end{aligned}$$

Clearly, neither case will lead to an orthonormal basis.

The recovery procedure is to pick an alternative vector that we know is not parallel, such as $\mathbf{i}$ or $\mathbf{j}$. This will lead to what seems like an instantaneous rotation around the z-axis. To understand this, raise your head upward until you are looking at the ceiling. If you keep going, you'll end up looking at the wall behind you, but upside down. To maintain the view looking right-side up, you'd have to rotate your head 180 degrees around (don't try this at home). This is not a very pleasing result, so avoid aligning the view direction with the world up vector whenever possible.

SOURCE CODE
DEMO
Rotation

There is a third possibility for controlling camera orientation. Suppose we want to treat our camera just like a normal object and specify a rotation matrix and translation vector. To do this we'll need to specify a starting orientation Ω for our camera and then apply our rotation matrix to find our camera's final orientation, after which we can apply our translation. Which orientation is chosen is somewhat arbitrary, but some are more intuitive and convenient than others. In our case we'll say that in our default orientation the camera has an initial view direction along the world x-axis, an initial view up along the world z-axis, and an initial view side along the $-y$-axis. This aligns the view up vector with the world up vector, and using the x-axis as the view direction fits the convention we set for objects' local space in Chapter 3.

Substituting these values into the view-to-world matrix for the standard left-handed view frame (equation 5.1) gives

$$\mathbf{\Omega}_s = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The equivalent matrix for the right-handed OpenGL view frame (using equation 5.2) is

$$\mathbf{\Omega}_{ogl} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Whichever system we are using, after this we apply our rotation to orient our frame in the direction we wish and, finally, the translation for the view position. If the three column vectors in our rotation matrix are **u**, **v**, and **w**, then for OpenGL the final transformation matrix is

$$\begin{aligned} \mathbf{M}_{view\to world} &= \mathbf{TR\Omega}_{ogl} \\ &= \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\mathbf{j} & \mathbf{k} & -\mathbf{i} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -\mathbf{v} & \mathbf{w} & -\mathbf{u} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

5.2.3 Constructing the View Transformation

Now that we have a way of representing and setting camera position and orientation, what do we do with it? The first step in the rendering process is to transform all of the objects in our world so that their coordinates are relative to the view frame, instead of the world frame. This gives us a sense of what we can see from our camera position. In the view frame, those objects along the line of the view direction vector (i.e., the $-z$-axis in the case of OpenGL) are in front of the camera and so will most likely be visible in our scene. Those on the other side of the plane formed by the view point, the view side vector, and the view up vector are behind the camera, and therefore not visible. In order to achieve this situation, we need to create a transformation from world space to view space, known as the world-to-view transformation, or more simply, the view transformation.

As it happens, we have a transformation that takes us from view space to world space. To create the reverse operator, we need only to invert the transformation. Since we know that it is an affine transformation, we can invert it as

$$\mathbf{M}_{world \rightarrow view} = \begin{bmatrix} \mathbf{R}^{-1} & -(\mathbf{R}^{-1}\mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{R}$ is the upper 3×3 block of our view-to-world transformation. And since $\mathbf{R}$ is the product of either a reflection and rotation matrix (in the standard case) or two rotations (in the OpenGL case), it is an orthogonal matrix, so we can compute its inverse by taking the transpose:

$$\mathbf{M}_{world \rightarrow view} = \begin{bmatrix} \mathbf{R}^{T} & -(\mathbf{R}^{T}\mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Source Code
Demo
LookAt

In practice this transformation is usually calculated directly, rather than taking the inverse of an existing transformation. For example, OpenGL has a utility call `gluLookAt()` that computes the view transformation assuming a view position, desired view position, and world up vector. One possible implementation is

```
void LookAt( const IvVector3& eye,
             const IvVector3& lookAt,
             const IvVector3& up )
{
  // compute view vectors
  IvVector3 viewDir = lookAt - eye;
  IvVector3 viewSide;
  IvVector3 viewUp;
```

```
    viewDir.Normalize();
    viewUp = up - up.Dot(viewDir)*viewDir;
    viewUp.Normalize();
    viewSide = viewDir.Cross(viewUp);

    // now set up matrices
    // build transposed rotation matrix
    IvMatrix33 rotate;
    rotate.SetRows( viewSide, viewUp, -viewDir );

    // transform translation
    IvVector3 eyeInv = -(rotate*eye);

    // build 4x4 matrix
    IvMatrix44 matrix;
    matrix.Rotation(rotate);
    matrix(0,3) = eyeInv.x;
    matrix(1,3) = eyeInv.y;
    matrix(2,3) = eyeInv.z;

    // set view to world transformation
    ::SetViewTransform( matrix.mV );
}
```

Note that we use the method `IvMatrix33:SetRows()` to set the transformed basis vectors since we're setting up the inverse matrix, namely, the transpose. There is also no recovery code if the view direction and world up vectors are collinear — it is assumed that any external routine will ensure this does not happen. The call `::SetViewTransform()` stores the calculated view transformation and is discussed in more detail in Section 5.7.

5.3 Projective Transformation

5.3.1 Definition

Now that we have a method for controlling our view position and orientation, and for transforming our objects into the view frame, we can look at taking our three-dimensional space and transforming it into a form suitable for display on a two-dimensional medium. This process of transforming from $\mathbb{R}^3$ to $\mathbb{R}^2$ is called *projection*.

We've already seen one example of projection: using the dot product to project one vector onto another. In our current case, we want to project the points that make up the vertices of an object onto a plane, called the *projection plane* or the *view plane*. We do this by following a *line of projection* through each point and determining where it hits the plane. These lines could be perpendicular to the plane, but as we'll see they don't have to be.

To understand how this works, we'll look at a very old form of optical projection known as the *camera obscura* (Latin for "dark room"). Suppose one enters a darkened room on a sunny day, and there is a small hole allowing a fraction of sunlight to enter the room. This light will be projected onto the opposite wall of the room, displaying an image of the world outside, albeit upside down and flipped left to right (Figure 5.4). This is the same principle that allows a pinhole camera to work; the hole is acting like the focal point of a lens. In this case all the lines of projection pass through a single *center of projection*. We can determine where a point will transform to on the plane by constructing a line through both the original point and the center of projection and calculating where it will intersect the plane of projection.

This sort of projection is known as *perspective projection*. Note that this relates to our perceived view in the real world. As an object moves farther away, its corresponding projection will shrink on the projection plane. Similarly, lines that are parallel in view space will appear to converge as their extreme points move farther away from the view position. This gives us a result consistent with our expected view in the real world. If we stand on some railroad tracks and look down a straight section, the rails will converge in the distance, and the ties will appear to shrink in size and become closer together. In most cases, since we are rendering real-world scenes — or at least, scenes that we want to be perceived as real-world — this will be the projection we will use.

There is, of course, one minor problem: the projected image is upside down and backwards. One possibility is just to flip the image when we display it on our medium. This is what happens with a camera: the image is captured on film upside down, but we can just rotate the negative or print to view it properly. This is not usually done in graphics. Instead, the projection plane

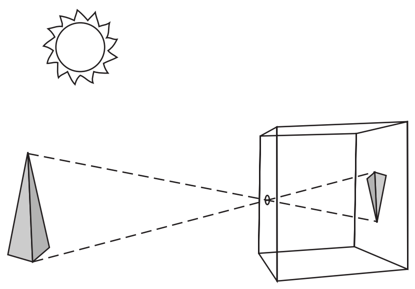


FIGURE 5.4 Camera obscura.

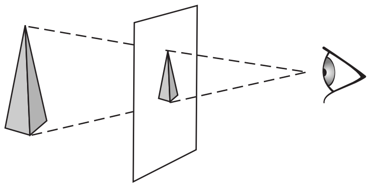


Figure 5.5 Perspective projection.

is moved to the other side of the center of projection, which is now treated as our view position (Figure 5.5). As we'll see, the mathematics for projection in this case are quite simple, and the objects located in the forward direction of our view will end up being projected right-side up. The objects behind the view will end up projecting upside-down, but (a) we don't want to render them anyway and (b) as we'll see there are ways of handling this situation.

An alternate type of projection is *parallel projection*, which can be thought of as a perspective projection where the center of projection is infinitely distant. In this case the lines of projection do not converge; they always remain parallel (Figure 5.6), hence the name. The placement of the view position and view plane are irrelevant in this case, but we place them in the same relative location to maintain continuity with perspective projection.

Parallel projection produces a very odd view if used for a scene: objects remain the same size no matter how distant they are, and parallel lines remain parallel. Parallel projections are usually used for CAD programs, where maintaining parallel lines is important. They are also useful for rendering 2D elements like interfaces; no matter how far from the eye a model is placed, it will always be the same size, presumably the size we expect.

A parallel projection where the lines of projection are perpendicular to the view plane is called an *orthographic projection*. By contrast, if they are not perpendicular to the view plane, this is known as an *oblique projection* (Figure 5.7). Two common oblique projections are the *cavalier projection*, where the projection angle is 45 degrees, and the *cabinet projection*, where the projection angle is $\cot^{-1}(1/2)$. When using cavalier projections, projected lines

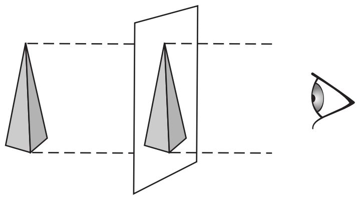


Figure 5.6 Orthographic parallel projection.

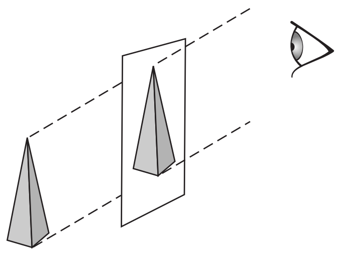


Figure 5.7 Oblique parallel projection.

have the same length as the original lines, so there is no perceived foreshortening. This is useful when printing blueprints, for example; any line can be measured to find the exact length of material needed to build the object. With cabinet projections, lines perpendicular to the projection plane foreshorten to half their length (hence the $\cot^{-1}(1/2)$), which gives a more realistic look without sacrificing the need for parallel lines.

We can also have oblique perspective projections where the line from the center of the view window to the center of projection is not perpendicular to the view plane. For example, suppose we need to render a mirror. To do so, we'll render the space using a plane reflection transformation and clip it to the boundary of the mirror. The plane of the mirror is our projection plane, but it may be at an angle to our view direction (Figure 5.8). For now, we'll concentrate on constructing projective transformations perpendicular to the projection plane and examine these special cases later.

As a side note, oblique projections can occur in the real world. The classic pictures we see of tall buildings, shot from the ground but with parallel sides, are done with a "view camera." This device has an accordian-pleated hood that allows the photographer to bend and tilt the lens up while keeping the film parallel to the side of the building. Ansel Adams also used such a camera to capture some of his famous landscape photographs.

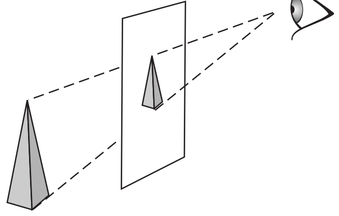


Figure 5.8 Oblique perspective projection.

5.3.2 The View Frustum

It is not possible to map the entire infinite view plane to a display device. Instead we set a *view window*, which frames the rectangular area on the view plane that will be mapped to the device. We could, naively, project all of the objects in the world to the view plane and then, when converting them to pixels, ignore those pixels that lie outside of the view window. However, for a large number of objects this would be very inefficient. It would be better to constrain our space to a convex volume, specified by a set of six planes. Anything inside these planes will be rendered; everything outside them will be ignored. This volume is known as the *view frustum*, or *view volume*.

To constrain what we render in the view frame xy directions, we specify four planes aligned with the edges of the view window. For perspective projection each plane is specified by the view position and two adjacent vertices of the view window (Figure 5.9), producing a semi-infinite pyramid. The angle between the upper plane and the lower plane is called the vertical *field of view*.

There is a relationship between field of view, view window size, and view plane distance: given two, we can easily find the third. For example, we can fix the view window size, adjust the field of view, and then compute the distance to the view plane. As the field of view gets larger, the distance to the view plane needs to get smaller to maintain the view window size. Similarly, a small field of view will lead to a longer view plane distance. Alternatively, we can set the distance to the view plane to a fixed value and use the field of view to determine the size of our view window. The larger the field of view, the larger the window and the more objects are visible in our scene. This gives us a primitive method for creating telephoto (narrow field of view) or wide-angle

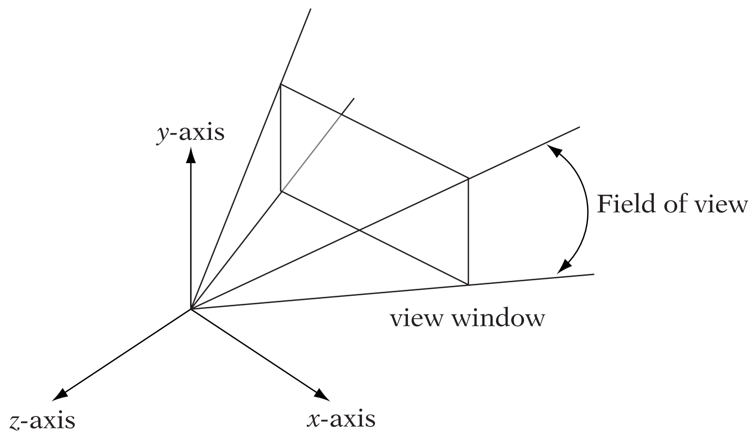


Figure 5.9 Perspective view frustum (right-handed system).

(wide field of view) lenses. We will discuss the relationship among these three quantities in more detail when we cover perspective projection.

Usually the field of view chosen needs to match the display medium, as the user perceives it, as much as possible. For a standard monitor placed about three feet away, the monitor only covers about a 25–30 degree field of view from the perspective of the user, so we would expect that we would use a field of view of that size in the game. However, this constrains the amount we can see in the game to a narrow area, which feels unnatural because we're used to a 180 degree field of view in the real world. The usual compromise is to set the field of view to the range of 60–90 degrees. The distortion is not that perceptible and it allows the user to see more of the game world. If the monitor were stretched to cover more of your personal field of view, as in some virtual reality systems, a larger field of view would be appropriate. And of course, if the desired effect is of a telephoto or wide-angle lens, a narrower or wider field of view, respectively, is appropriate.

For parallel projection, the xy culling planes are parallel to the direction of projection, so opposite planes are parallel and we end up with a parallelpiped that is open at two ends (Figure 5.10). There is no concept of field of view in this case.

In both cases, to complete a closed view frustum we also define two planes which constrain objects in the view frame z-direction: the near and far planes (Figure 5.11). With perspective projection it may not be obvious why we need

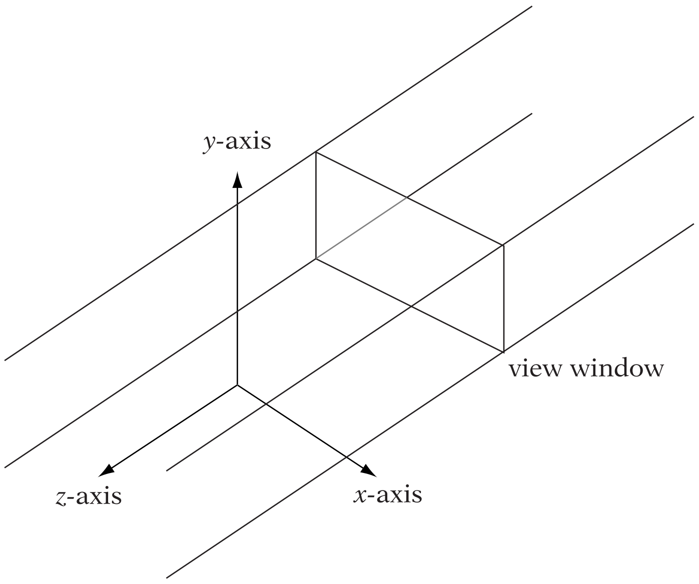


FIGURE 5.10 Parallel view frustum (right-handed system).

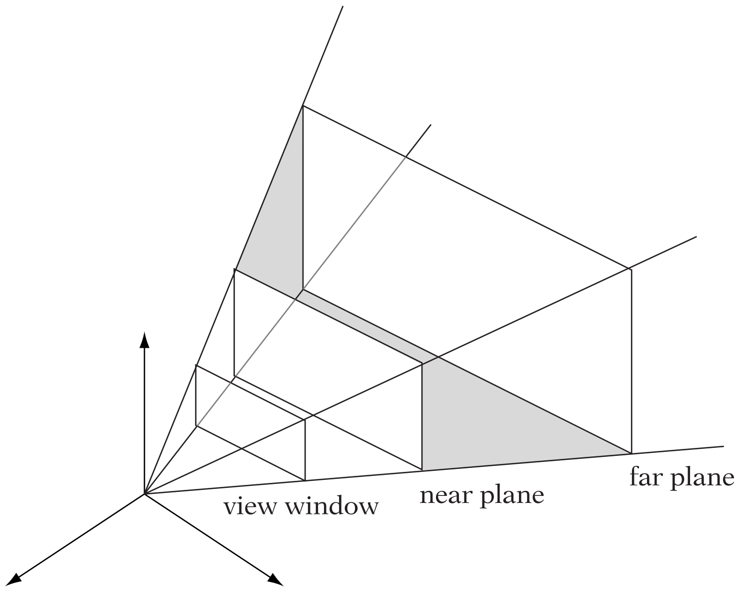


Figure 5.11 View frustum with near plane and far plane.

a near plane, since the xy-planes converge at the center of projection, closing the viewing region at that end. However, as we will see when we start talking about the perspective transformation, rendering objects at the view frame origin (which in our case is the same as the center of projection) can lead to a possible division by zero. This would adversely affect our rendering process. We could also, like some viewing systems, use the view plane as the near plane, but not doing so allows us a little more flexibility.

In some sense, the far plane is optional. Since we don't have an infinite number of objects or an infinite amount of game space, we could forgo using the far plane and just render everything within the five other planes. However, the far plane is useful for culling objects and area from our rendering process, so having a far plane is good for efficiency's sake. It is also extremely important in the hidden surface removal method of z-buffering; the distance between the near and far planes is a factor in determining the precision we can expect in our z-values.

5.3.3 Normalized Device Coordinates

Currently our objects are in view frame coordinates. However, as mentioned we will be projecting from $\mathbb{R}^3$ to $\mathbb{R}^2$, so we will need a frame for the space of the

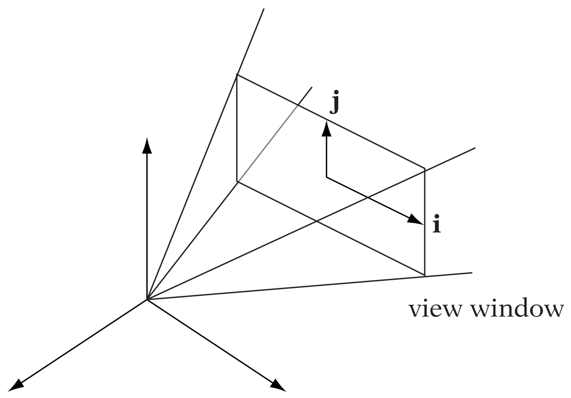


FIGURE 5.12a NDC frame in view window.

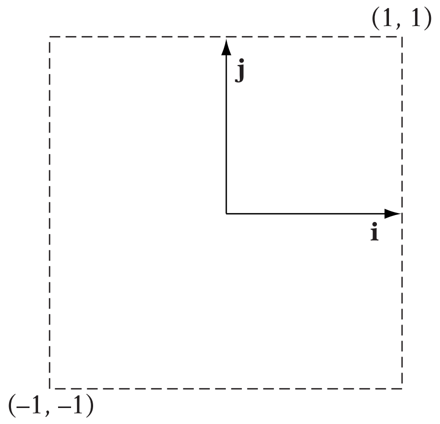


FIGURE 5.12b View window after NDC transformation.

view plane. We'll use as our origin the center of the view window, and create basis vectors that align with the sides of the view window, with magnitudes of half the width and height of the window, respectively (Figure 5.12a). Within this frame, our view window is transformed into a square two units wide and centered at the origin, bounded by the $x = 1$, $x = -1$, $y = 1$, and $y = -1$ lines (Figure 5.12b).

Using this as our frame provides a certain amount of flexibility when mapping to devices of varying size. Rather than transform directly to our screen area, which could be of variable width and height, we use this normalized form as an intermediate step to simplify our calculations and then do the screen conversion as our final step. Because of this, coordinates in this frame are known as *normalized device coordinates*.

To take advantage of the normalized device coordinate frame, or *NDC space*, we'll want to create our projection so that it always gives us the -1 to 1 behavior, regardless of the exact view configuration. This helps us to compartmentalize the process of viewing (just as the view matrix did).

To simplify this mapping to the NDC frame, we will begin by using a view window in the view frame with a height of 2 units. This means that for the case of a centered view window, xy coordinates on the view plane will be equal to the projected coordinates in the NDC frame. In this way we can consider the projection as related to the view plane in view coordinates and not worry about a subsequent transformation. When adjusting our field of view, we will move the view plane relative to the center of projection, rather than changing the size of the view window.

5.3.4 Homogeneous Coordinates

Previously we stated that a point in $\mathbb{R}^3$ can be represented by $(x, y, z, 1)$ without explaining much about what that might mean. This representation is part of a more general representation for points known as homogeneous coordinates, which prove useful to us when handling perspective projections. In general, homogeneous coordinates work as follows: if we have a "standard" representation in n-dimensional space, then we can represent the same point in a $(n+1)$–dimensional space by scaling the original coordinates by a single value and then adding the scalar to the end as our final coordinate. Since we can choose from an infinite number of scalars, a single point in $\mathbb{R}^n$ will be represented by an infinite number of points in the $(n+1)$–dimensional space. This $(n+1)$–dimensional space is called a *real projective space* or $\mathbb{R}P^n$. In computer graphics parlance, the real projective space $\mathbb{R}P^3$ is also often called *homogeneous space*.

Suppose we start with a point (x, y, z) in $\mathbb{R}^3$, and we want to map it to a point (x', y', z', w) in homogeneous space. We pick a scalar for our fourth element w, and scale the other elements by it, to get (xw, yw, zw, w). As we might expect, our standard value for w will be 1, so (x, y, z) maps to $(x, y, z, 1)$. To map back to three-dimensional space, divide the first three coordinates by w, so (x', y', z', w) goes to $(x'/w, y'/w, z'/w)$. Since our standard value for w is just 1, we could just drop the $w : (x', y', z', 1) \rightarrow (x', y', z')$. However, in the cases that we'll be concerned with next, we need to perform the division by w.

What happens when $w = 0$? In this case a point in $\mathbb{R}P^3$ doesn't represent a point in $\mathbb{R}^3$, but a vector. We can think of this as a "point at infinity." While we will try to avoid cases where $w = 0$, they do creep in, so checking for this before performing the homogeneous division is often wise.

5.3.5 Perspective Projection

Source Code Demo
Perspective

Since this is the most common projective transform we'll encounter, we'll begin by constructing the mathematics necessary for the perspective projection. To simplify things, let's take a 2D view of the situation on the yz-plane and ignore the near and far planes for now (Figure 5.13). We have the y-axis pointing up, as in the view frame, and the projection direction along the negative z-axis as it would be in OpenGL. The point on the left represents our center of projection, and the vertical line our view plane. The diagonal lines represent our y culling planes.

Suppose we have a point P_v in view coordinates that lies on one of the view frustum planes, and we want to find the corresponding point P_s that lies on the view plane. Finding the y coordinate of P_s is simple: we follow the line of projection along the plane until we hit the top of the view window. Since the height of the view window is 2 and is centered on 0, the y coordinate of P_s is half the height of the view window, or 1. The z coordinate will be negative since we're looking along the negative z-axis and will have a magnitude equal to the distance d from the view position to the projection plane. So the z coordinate will be $-d$.

But how do we compute d? As we see, the cross section of the y view frustum planes are represented as lines from the center of projection through the extents of the view window $(1, d)$ and $(-1, d)$. The angle between these lines is our field of view θ_{fov}. We'll simplify things by considering only the area that lies above the negative z-axis; this bisects our field of view to an angle of $\theta_{fov}/2$. If we look at the triangle bounded by the negative z-axis, the cross section of the upper view frustum plane, and the cross section of the projection plane, we can use trigonometry to compute d. Since we know the distance between

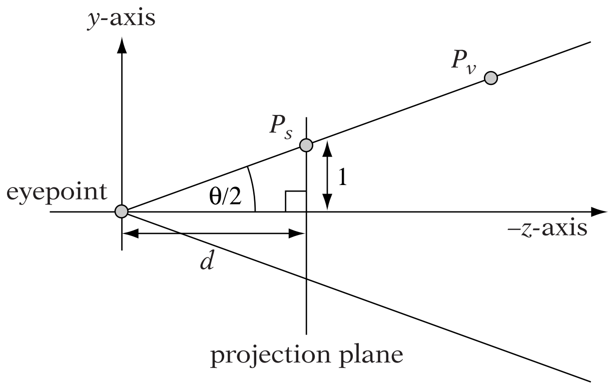


Figure 5.13 Perspective projection construction.

the negative z-axis and the extreme point P_s is 1, we can say that

$$\frac{1}{d} = \tan(\theta_{fov}/2)$$

Rewriting this in terms of d, we get

$$\begin{aligned} d &= \frac{1}{\tan\left(\frac{\theta_{fov}}{2}\right)} \\ &= \cot\left(\frac{\theta_{fov}}{2}\right) \end{aligned}$$

So for this fixed view window size, as long as we know the angle of field of view, we can compute the distance d, and vice versa.

This gives the coordinates for any point that lies on the upper y view frustum plane; in this 2D cross section they all project down to a single point $(1, -d)$. Similarly, points that lie on the lower y frustum plane will project to $(-1, -d)$. But suppose we have a general point (y_v, z_v) in view space. We know that its projection will lie on the view plane as well, so its z_{ndc} coordinate will be $-d$. But how do we find y_{ndc}?

We can compute this by using similar triangles (Figure 5.14). If we have a point (y_v, z_v), the length of the sides of the corresponding right triangle in our diagram are y_v and $-z_v$ (since we're looking down the $-z$-axis, any visible z_v is negative, so we need to negate it to get a positive value). The length of sides of the right triangle for the projected point are y_{ndc} and d.

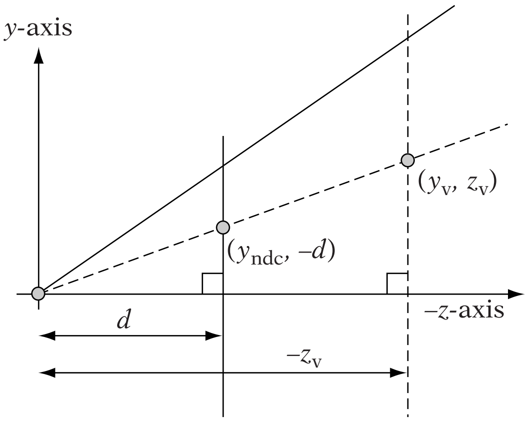


FIGURE 5.14 Perspective projection similar triangles.

By similar triangles (both have the same angles), we get

$$\frac{y_{ndc}}{d} = \frac{y_v}{-z_v}$$

Solving for y_{ndc}, we get

$$y_{ndc} = \frac{dy_v}{-z_v}$$

This gives us the coordinate in the y direction. If our view region was square, then we could use the same formula for the x direction. Most, however, are rectangular to match the relative dimensions of a computer monitor or other viewing device. We must correct for this by the aspect ratio of the view region. The aspect ratio a is defined as

$$a = \frac{w_v}{h_v}$$

where w_v and h_v are the width and height of the view rectangle, respectively. We're going to assume that the NDC view window height remains at 2 and correct the NDC view width by the aspect ratio. This gives us a formula for similar triangles of

$$\frac{ax_{ndc}}{d} = \frac{x_v}{-z_v}$$

Solving for x_{ndc}:

$$x_{ndc} = \frac{dx_v}{-az_v}$$

So our final projection transformation equations are

$$x_{ndc} = \frac{dx_v}{-az_v}$$
$$y_{ndc} = \frac{dy_v}{-z_v}$$

The first thing to notice is that we are dividing by a z coordinate, so we will not be able to represent the entire transformation by a matrix operation, since it is neither linear nor affine. However, it does have some affine elements, scaling by d and d/a for example, which can be performed by a transformation matrix. This is where the conversion from homogeneous space comes in.

Recall that to transform from $\mathbb{R}P^3$ to $\mathbb{R}^3$ we need to divide the other coordinates by the w value. If we can set up our matrix to map $-z_v$ to our w value, we can take advantage of the homogeneous divide to handle the nonlinear part of our transformation. We can write the situation before the homogeneous divide as a series of linear equations:

$$\begin{aligned} x' &= \frac{d}{a}x \\ y' &= dy \\ z' &= dz \\ w' &= -z \end{aligned}$$

and treat this as a four-dimensional linear transformation. Looking at our basis vectors, $\mathbf{e}_0$ will map to $(d/a, 0, 0, 0)$, $\mathbf{e}_1$ to $(0, -d, 0, 0)$, $\mathbf{e}_2$ to $(0, 0, d, -1)$, and $\mathbf{e}_3$ to $(0, 0, 0, 0)$ since w is not used in any of the equations.

Based on this, our homogeneous perspective matrix is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

As expected, our transformed w value will no longer be 1. Also note that the right-most column of this matrix is all zeros, which means that this matrix has no inverse. This is to be expected, since we are losing one dimension of information. Individual points in view space that lie along the same line of projection will project to a single point in NDC space. Given only the points in NDC space, it would be impossible to reconstruct their original positions in view space.

Let's see how this matrix works in practice. If we multiply it by a generic point in view space, we get

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} dx_v/a \\ dy_v \\ dz_v \\ -z_v \end{bmatrix}$$

Dividing out the w (also called the reciprocal divide), we get

$$x_{ndc} = \frac{dx_v}{-az_v}$$

$$y_{ndc} = \frac{dy_v}{-z_v}$$
$$z_{ndc} = -d$$

which is what we expect.

So far, we have dealt with projecting x and y and completely ignored z. In the preceding derivation all z values map to $-d$, the negative of the distance to the projection plane. While losing a dimension makes sense conceptually— we are projecting from a 3D space down to a 2D plane, after all— for practical reasons it is better to keep some measure of our z values around for z-buffering and other depth comparisons (discussed in more detail in Chapter 8). Just as we're mapping our x and y values within the view window to an interval of $[-1, 1]$, we'll do the same for our z values within the near plane and far plane positions. We'll specify the near and far values n and f relative to the view position, so points lying on the near plane have a z_v value of $-n$, which maps to a z_{ndc} value of -1. Those points lying on the far plane have a z_v value of $-f$ and will map to 1 (Figure 5.15).

We'll derive our equation for z_{ndc} in a slightly different way than our xy coordinates. There are two parts to mapping the interval $[-n, -f]$ to $[-1, 1]$. The first is scaling the interval to a width of 2, and the second is translating it to $[-1, 1]$. Ordinarily, this would be a straightforward linear process, however we also have to contend with the final w divide. Instead, we'll create a perspective matrix with unknowns for the scaling and translation factors and use the fact that we know the final values for $-n$ and $-f$ to solve for the unknowns.

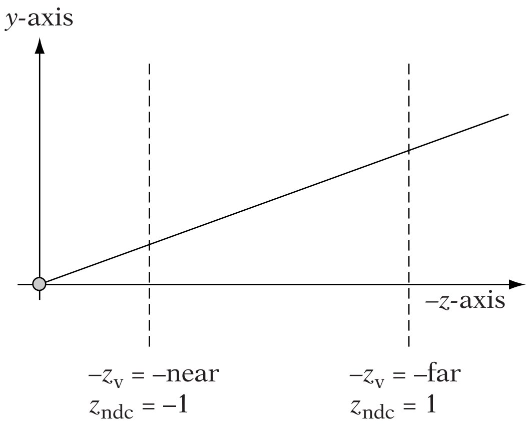


FIGURE 5.15 Perspective projection: z values.

Our starting perspective matrix, then, is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where A and B are our unknown scale and translation factors, respectively. If we multiply this by a point $(0, 0, -n)$ on our near plane:

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -An + B \\ n \end{bmatrix}$$

Dividing out the w gives

$$z_{ndc} = -A + \frac{B}{n}$$

We know that any point on the near plane maps to a normalized device coordinate of -1, so we can substitute -1 for z_{ndc} and solve for B, which gives us

$$B = (A - 1)n \tag{5.3}$$

We'll substitute equation 5.3 into our original matrix and multiply by a point $(0, 0, -f)$ on the far plane now:

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & (A-1)n \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Af + (A-1)n \\ f \end{bmatrix}$$

This gives us a z_{ndc} of

$$\begin{aligned} z_{ndc} &= -A + (A-1)\frac{n}{f} \\ &= -A + A\left(\frac{n}{f}\right) - \frac{n}{f} \\ &= A\left(\frac{n}{f} - 1\right) - \frac{n}{f} \end{aligned}$$

Setting z_{ndc} to 1 and solving for A, we get

$$
\begin{aligned}
A\left(\frac{n}{f}-1\right)-\frac{n}{f} &= 1\\
A\left(\frac{n}{f}-1\right) &= \frac{n}{f}+1\\
A &= \frac{\frac{n}{f}+1}{\frac{n}{f}-1}\\
&= \frac{n+f}{n-f}
\end{aligned}
$$

If we substitute this into equation 5.3, we get

$$B = \frac{2nf}{n-f}$$

So our final perspective matrix is

$$
\mathbf{M}_{persp} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}
$$

The matrix that we have generated is the same one produced by an OpenGL call: `gluPerspective()`. This function takes the field of view[1], aspect ratio, and near and far plane settings, builds the perspective matrix, and multiplies it by the current matrix.

It is important to be aware that this matrix will not work for all viewing systems. For one thing, for most other viewing systems (i.e., other than OpenGL), our view frame looks down the positive z-axis, so this affects both our xy and z transformations. For example, in this case we have mapped $[-n, -f]$ to $[-1, 1]$. With the standard system we would want to begin by mapping $[n, f]$ to the NDC z range. In addition, this range is not always set to $[-1, 1]$. Direct3D, for one, maps to $[0, 1]$ in the z direction.

1. Recall that our value d is generated from the field of view by $d = \cot(\theta_{fov}/2)$.

Using the standard view frame and this mapping gives us a perspective transformation matrix of

$$\mathbf{M}_{pD3D} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix can be derived using the same principles described above.

When setting up a perspective matrix, it is good to be aware of the issues involved in rasterizing z values. In particular, to maintain z precision keep the near and far planes as close together as possible. More details on managing perspective z precision can be found in Chapter 8.

5.3.6 Oblique Perspective

Source Code
Demo
Stereo

The matrix we constructed in the previous section is an example of a standard perspective matrix, where the direction of projection through the center of the view window is perpendicular to the view plane. A more general example of perspective is generated by the OpenGL `glFrustum()` call. This call takes six parameters: the near and far z distances, as before, and four values that define our view window on the near z plane: the x interval $[l, r]$ (left, right) and the y interval $[b, t]$ (bottom, top). Figure 5.16a shows how this looks in $\mathbb{R}^3$, and Figure 5.16b shows the cross section on the yz plane. As we can see, these values need not be centered around the z-axis, so we can use them to generate an oblique projection.

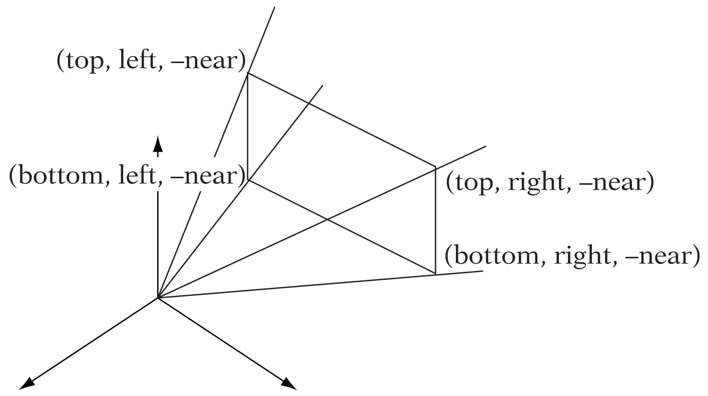


Figure 5.16a View window for `glFrustum`, 3D view.

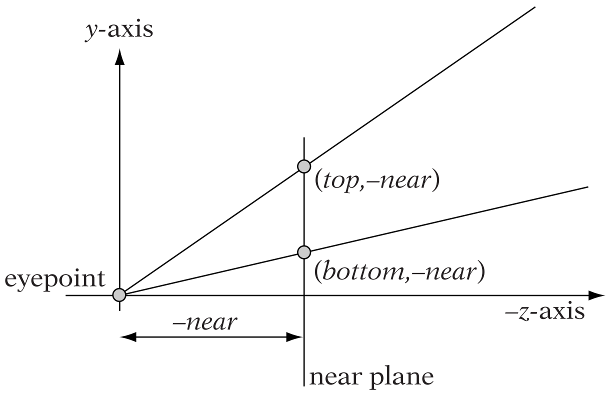


FIGURE 5.16b View window for `glFrustum`, cross-section.

To derive this matrix, once again we begin by considering similar triangles in the y-direction. Remember that given a point $(y_v, -z_v)$, we project to a point on the view plane $(dy_v/-z_v, -d)$, where d is the distance to the projection. However, since we're using our near plane as our projection plane, this is just $(ny_v/-z_v, -n)$. The projection remains the same, we're just moving the window of projected points that lie within our view frustum.

With our previous derivation, we could stop at this point because our view window on the projection plane was already in the interval $[-1, 1]$. However, our new view window lies in the interval $[b, t]$. We'll have to adjust our values to properly end up in NDC space. The first step is to translate the center of the window, located at $(t+b)/2$, to the origin. Applying this translation to the current projected y coordinate gives us

$$y' = y - \frac{(t+b)}{2}$$

We now need to scale to change our interval from a magnitude of $(t-b)$ to a magnitude of 2 by using a scale factor $2/(t-b)$:

$$y_{ndc} = \frac{2y}{t-b} - \frac{2(t+b)}{2(t-b)} \tag{5.4}$$

If we substitute $ny_v/-z_v$ for y and simplify, we get

$$y_{ndc} = \frac{2n\dfrac{y_v}{-z_v}}{t-b} - \frac{2(t+b)}{2(t-b)}$$

$$= \frac{2n\frac{y_v}{-z_v}}{t-b} - \frac{(t+b)\frac{-z_v}{-z_v}}{t-b}$$
$$= \frac{1}{-z_v}\left(\frac{2n}{t-b}y_v + \frac{t+b}{t-b}z_v\right)$$

A similar process gives us the following for the x direction:

$$x_{ndc} = \frac{1}{-z_v}\left(\frac{2n}{r-l}x_v + \frac{r+l}{r-l}z_v\right)$$

We can use the same A and B from our original perspective matrix, so our final projection matrix is

$$\mathbf{M}_{oblpersp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

A casual inspection of this matrix gives some sense of what's going on here. We have a scale in the x, y, and z directions, which provides the mapping to the interval $[-1, 1]$. In addition, we have a translation in the z direction to align our interval properly. However, in the x and y directions, we are performing a z shear to align the interval, which provides us with the oblique projection.

The equivalent Direct3D matrix is

$$\mathbf{M}_{opD3D} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

5.3.7 Orthographic Parallel Projection

Source Code
Demo
Orthographic

After considering perspective projection in two forms, orthographic projection is much easier. Examine Figure 5.17, which shows a side view of our projection space as before, with the lines of projection passing through the view plane and the near and far planes shown as vertical lines. This time the lines of projection are parallel to each other (hence this is a parallel projection) and parallel to the z-axis (hence an orthographic projection).

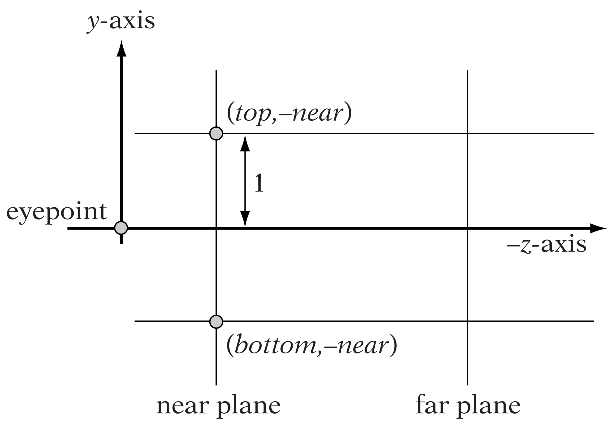


FIGURE 5.17 Orthographic projection construction.

We can use this to help us generate the matrix for the OpenGL `glOrtho()` call. Like `glFrustum()`, this call takes six parameters: the near and far z distances, and four values l, r, b, and t that define our view window on the near z plane. As before, the near plane is our projection plane, so a point (y_v, z_v) projects to a point $(y_v, -n)$. Note that since this is a parallel projection, there is no division by z or scale by d; we just use the y value directly. Like `glFrustum()` we now need to consider only values between t and b and scale and translate them to the interval $[-1, 1]$. Substituting y_v into our range transformation equation 5.4, we get

$$y_{ndc} = \frac{2y_v}{t-b} - \frac{t+b}{t-b}$$

A similar process gives us the equation for x_{ndc}. We can do the same for z_{ndc}, but since our viewable z values are negative and our values for n and f are positive, we need to negate our z value and then perform the range transformation. The result of all three equations is

$$\mathbf{M}_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There are a few things we can notice about this matrix. First of all, multiplying by this matrix gives us a w value of 1, so we don't need to perform the homogeneous division. This means that our z values will remain linear; that is, they will not compress as they approach the far plane. This gives us better z resolution at far distances than the perspective matrices. It also means that this is a linear transformation matrix and possibly invertible.

Secondly, in the x and y directions, what was previously a z-shear in the oblique perspective matrix has become a translation. Before, we had to use shear because for a given point the displacement was dependent on the distance from the view position. Because the lines of projection are now parallel, all points displace equally, so only a translation is necessary.

The Direct3D equivalent matrix is

$$\mathbf{M}_{orthoD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.3.8 Oblique Parallel Projection

SOURCE CODE
DEMO
Oblique

While most of the time we'll want to use orthographic projection, we may from time to time need an oblique parallel projection. For example, suppose for part of our interface we wish to render our world as a set of schematics or display particular objects with a 2D CAD/CAM feel. This set of projections will achieve our goal.

Neither OpenGL nor Direct3D have a particular routine that handles oblique parallel projections, so we'll derive one ourselves. We will give our projection a slight oblique angle ($\cot^{-1}(1/2)$, which is about 63.4 degrees), which gives a 3D look without perspective. More extreme angles in x and y tend to look strangely flat.

Figure 5.18 is another example of our familiar cross section, this time showing the lines of projection for our oblique projection. As we can see, we move one unit in the y direction for every two units we move in the z direction. Using the formula of $\tan(\theta) = opposite/adjacent$, we get

$$\tan(\theta) = \frac{2}{1}$$
$$\cot(\theta) = \frac{1}{2}$$
$$\theta = \cot^{-1}\frac{1}{2}$$

which confirms the expected value for our oblique angle.

As before, we'll consider the yz case first and extrapolate to x. Moving 1 unit in y and 2 units in $-z$ gives us the vector $(1, -2)$, so the formula for the

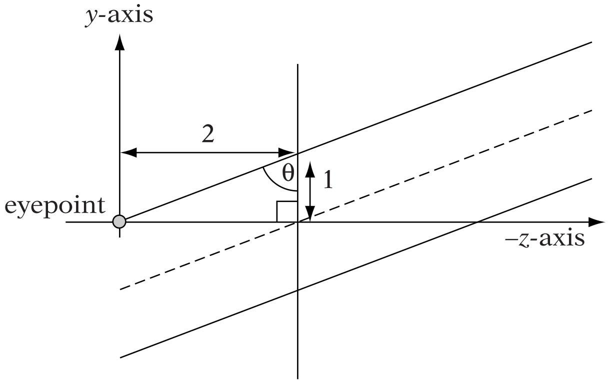


FIGURE 5.18 Example of oblique parallel projection.

line of projection for a given point P is

$$L(t) = P + t(1, -2)$$

We're only interested in where this line crosses the near plane, or where

$$P_z - 2t = -n$$

Solving for t:

$$t = \frac{1}{2}(n + P_z)$$

Plugging this into the formula for the y-coordinate of $L(t)$, we get

$$y' = P_y + \frac{1}{2}(n + P_z)$$

Finally, we can plug this into our range transformation equation 5.4 as before to get

$$\begin{aligned} y_{ndc} &= 2\frac{\left[y_v + \frac{1}{2}(n + z_v)\right]}{t - b} - \frac{t + b}{t - b} \\ &= \frac{2y_v}{t - b} - \frac{t + b}{t - b} + \frac{z_v + n}{t - b} \end{aligned}$$

Once again, we examine our transformation equation more carefully. This is the same as the orthographic transformation we had before, with an additional z-shear, as we'd expect for an oblique projection. In this case the shear plane is the near plane rather than the xy plane, so we add an additional factor of $\frac{n}{t-b}$ to take this into account.

A similar process can be used for x. Since the oblique projection has a z-shear, z is not affected and so:

$$\mathbf{M}_{cab} = \begin{bmatrix} \frac{2}{r-l} & 0 & \frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & \frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Direct3D equivalent matrix is

$$\mathbf{M}_{cabD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & \frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & \frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.4 Culling and Clipping

5.4.1 Why Cull or Clip?

In order to improve rendering, both for speed and appearance's sake, it is necessary to cull and clip objects. Culling is the process of removing objects from consideration for some process, whether it be rendering, simulation, or collision detection. In this case that means we want to ignore any models or whole pieces of geometry that lie outside of the view frustum, since they will never end up being projected to the view window. In Figure 5.19, the lighter objects lie outside of the view frustum and so will be culled for rendering.

Clipping is the process of cutting geometry to match a boundary, whether it be a polygon or, in our case, a plane. Vertices that lie outside the boundary will be removed and new ones generated for each edge that crosses the boundary. For example, in Figure 5.20 we see a cube being clipped by a plane, showing the extra vertices created where each edge intersects the plane. We'll use this for any models that cross the view frustum, cutting the geometry so that it fits within the frustum. We can think of this as slicing a piece of geometry off for every frustum plane.

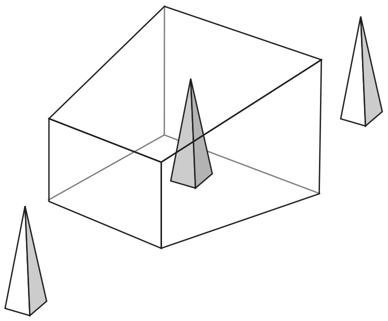


FIGURE 5.19 View frustum culling.

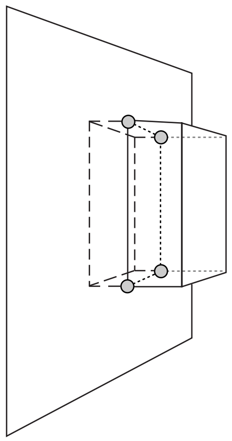


FIGURE 5.20 View frustum clipping.

Why should we want to use either of these for rendering? For one thing, it is more efficient to remove any data that will not ultimately end up on the screen. While copying the transformed object to the frame buffer (a process called rasterization) is almost always done in hardware and thus is fast, it is not free. Anywhere we can avoid unnecessary work is good.

But even if we had infinite rasterization power, we would still want to cull and clip when performing perspective projection. Figure 5.21 shows one example why. Recall that we finessed the problem of the camera obscura inverting images by moving the view plane. However, we still have the same problem if an object is behind the view position; it will end up projected upside down. The solution is to cull objects that lie behind the view position.

Figure 5.22a shows another example. Suppose we have a polygon edge that crosses the $z = 0$ plane. With the correct projection, the line segment starts at the middle of the view, moves up, and wraps around to reemerge at

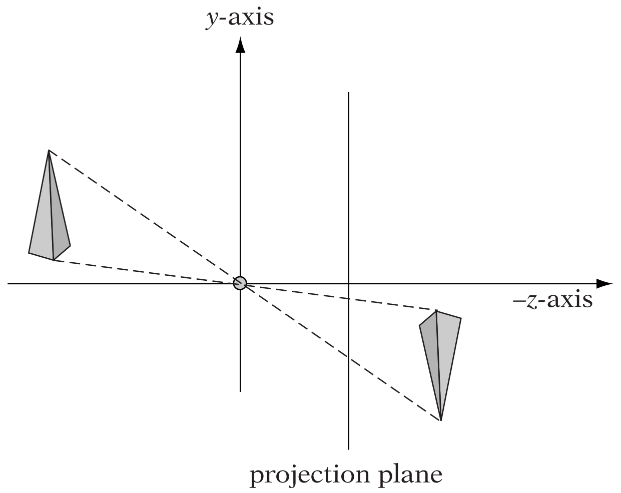


FIGURE 5.21 Projection of objects behind the eye.

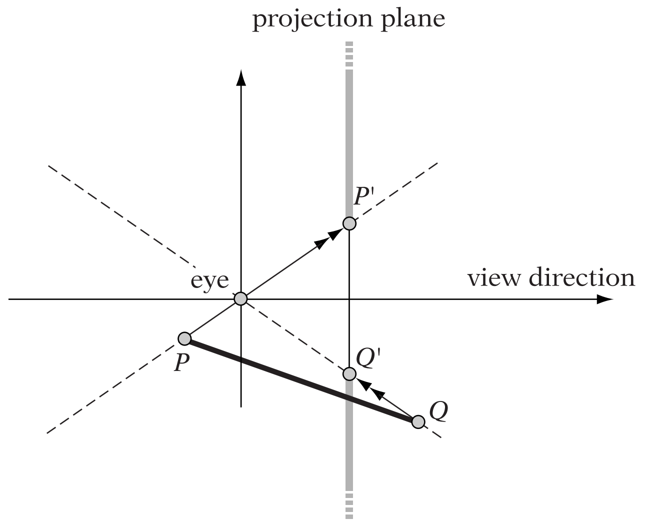


FIGURE 5.22a Projection of line segment crossing behind view point.

the bottom of the view. In practice, however, the rendering hardware has only the two projected vertices as input. It will end up taking the short route and rasterizing the wrong line segment between the two vertices (Figure 5.22b). If we clip the line segment to only the section that is viewable (Figure 5.22c), we end with only a portion of the line segment, but at least it is from the correct projection.

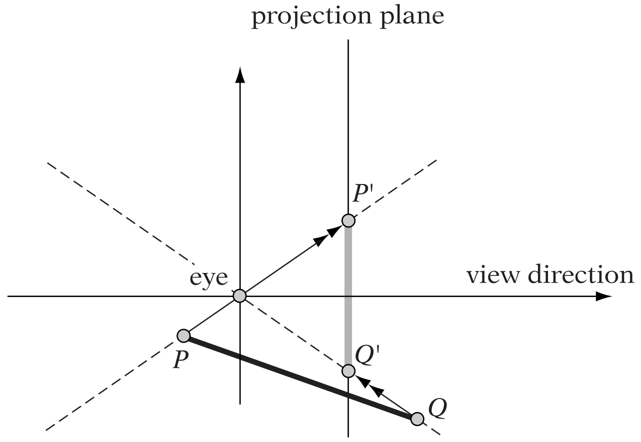


Figure 5.22b Incorrect line segment rendering based on projected endpoints.

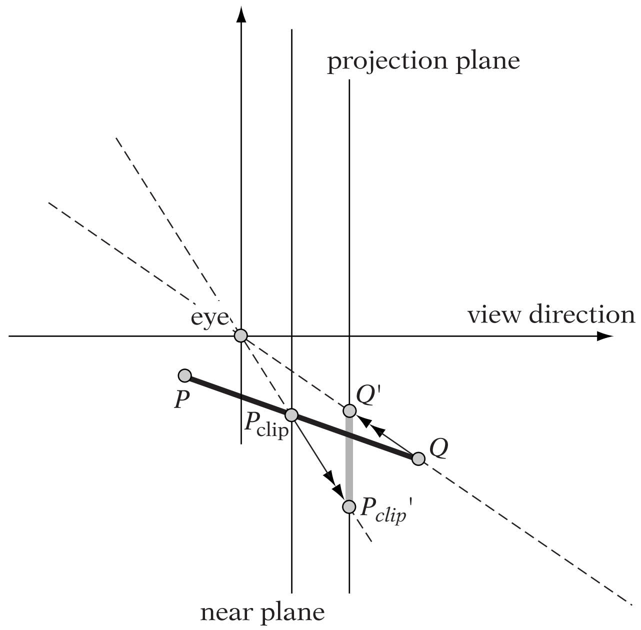


Figure 5.22c Line segment rendering when clipped to near plane.

There is also the problem of vertices that lie on the $z = 0$ plane. When transformed to homogeneous space by the perspective matrix, a point $(x, y, 0, 1)$ will become $(x', y', z', 0)$. The resulting transformation into NDC space will be a division by 0, which is not valid.

To avoid all of these issues, at the very least we need to set a near plane that lies in front of the eye so that the view position itself does not lie within the view frustum. We first cull any objects that lie on the same side of the near plane as the view position. We then clip any objects that cross the near plane. This avoids both the potential of dividing by 0 (although it is sometimes prudent to check for it anyway, at least in a debug build) and trying to render any line segments passing through infinity.

While clipping to a near plane is a bare minimum, clipping to the top, bottom, left, and right planes is useful as well. While the windowing hardware will usually ignore any pixels that lie outside of a window's visible region (this is commonly known as *scissoring*), it is faster if we can avoid unnecessary rasterization. Also, if we want to set a viewport that covers a subrectangle of a window, not clipping to the border of the viewport may lead to spurious geometry being drawn (although most hardware allows for adjustable scissoring regions; in particular, OpenGL and D3D provide interfaces to set this).

Finally, some hardware has a limited range for screen space positions, for example, 0 to 4095. The viewable area might lie in the center of this range, say from a minimum point of (1728,1808) to a maximum point of (2688,2288). The area outside of the viewable area is known as the *guard band* — anything rendered to this will be ignored, since it won't be displayed. In some cases we can avoid clipping in x and y, since we can just render objects whose screen space projection lies within the guard band and know that they will be handled automatically by the hardware. This can improve performance considerably, since clipping can be quite expensive. However, it's not entirely free. Values that lie outside the maximum range for the guard band will wrap around. So a vertex that would normally project to coordinates that should lie off the screen, say (6096,6096), will wrap to (2000,2000) — right in middle of the viewable area. Unfortunately, the only way to solve this problem is what we were trying to avoid in the first place: clipping in the x and y directions. On the other hand, using the guard band carefully can reduce the amount of clipping that we have to do overall.

5.4.2 Culling

A naive method of culling a model against the view frustum is to test each of its vertices against each of the frustum planes in turn. We designate the plane normal for each plane as pointing towards the "inside" half-space. If for one plane $ax + by + cz + d < 0$ for every vertex $P = (x, y, z)$, then the model lies outside of the frustum and we can ignore it. Conversely, if for all the frustum planes and all the vertices $ax + by + cz + d > 0$, then we know the model lies entirely inside the frustum and we don't need to worry about clipping it.

While this will work, for models with large numbers of vertices this becomes expensive, probably outweighing any savings we might gain by not rendering the objects. Instead, culling is usually done by approximating the object with a convex bounding volume, such as a sphere, that contains all of the vertices for the object. Rather than test each vertex against the planes, we test only the bounding object. Since it is a convex object and all the vertices are contained within it, we know that if the bounding object lies outside of the view frustum, all of the model's vertices must lie outside as well. More information on computing bounding objects and testing them against planes can be found in Chapter 11.

Bounding objects are usually placed in the world frame to aid with collision detection, so culling is often done in the world frame as well. This requires storing a representation of each frustum plane in world coordinates, but the additional 24 values required is worth the speedup gained. We can find each x or y clipping plane in the view frame by using the view position and two corners of the view window to generate the player. The two z planes (in OpenGL) are $z = -near$ and $z = -far$, respectively. Transforming them to the world frame is a simple case of using the technique for transforming plane normals, as described in Chapter 3.

While view frustum culling can remove a large number of objects from consideration, it's not the only culling method. In Chapter 6 we'll discuss backface culling, which allows us to determine which polygons are pointing away from the camera and ignore them. There also are a large number of culling methods that break up the scene in order to cull objects that aren't visible. This can help with interior levels, so you don't render rooms that may be within the view frustum but not visible because they're blocked by a wall. Such methods are out of the purview of this book but are described in detail in many of the references cited in the following sections.

5.4.3 General Plane Clipping

Source Code
Demo
Clipping

To clip polygons, we first need to know how to clip a polygon edge (i.e., a line segment) to a plane. As we'll see, the problem of clipping a polygon to a plane degenerates to handling this case. Suppose we have a line segment $\overline{PQ}$, with endpoints P and Q, that crosses a plane. We'll say that P is inside our clip space and Q is outside. Our clipped line segment will be $\overline{PR}$, where R is the intersection of the line segment and the plane (Figure 5.23).

To find R, we take the line equation $P + t(Q - P)$, plug it into our plane equation $ax + by + cz + d = 0$, and solve for t. To simplify the equations, we'll define $\mathbf{v} = Q - P$. Substituting the parameterized line coordinates for x, y, and z, we get

$$\begin{aligned}0 &= a(P_x + tv_x) + b(P_y + tv_y) + c(P_z + tv_z) + d \\ &= aP_x + tav_x + bP_y + tbv_y + cP_z + tcv_z + d\end{aligned}$$

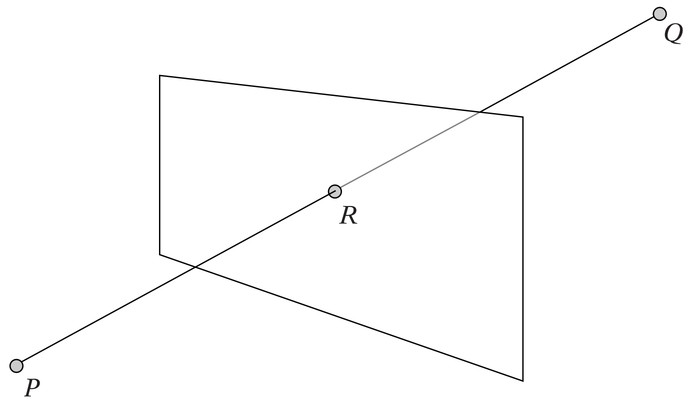


FIGURE 5.23 Clipping edge to plane.

$$
\begin{aligned}
&= aP_x + bP_y + cP_z + d + t(av_x + bv_y + cv_z) \\
t &= \frac{-aP_x - bP_y - cP_z - d}{av_x + bv_y + cv_z} \\
&= \frac{(aP_x + bP_y + cP_z + d)}{(aP_x + bP_y + cP_z + d) - (aQ_x + bQ_y + cQ_z + d)}
\end{aligned}
$$

We can use Blinn's notation [11], slightly modified, to simplify this to

$$t = \frac{BCP}{BCP - BCQ}$$

where BCP is the result from the plane equation (the boundary coordinate) when we test P against the plane, and BCQ is the result when we test Q against the plane. The resulting clip point R is

$$R = P + \frac{BCP}{BCP - BCQ}(Q - P)$$

To clip a polygon to a plane, we need to clip each edge in turn. A standard method for doing this is to use the Sutherland-Hodgeman algorithm [107]. For each edge we first test it against the plane. Depending on what the result is, we output particular vertices for the clipped polygon. There are four possible cases for an edge from P to Q (Figure 5.24). If both are inside, then we output P. The vertex Q will be output when we consider it as the start of the next edge. If both are outside, we output nothing. If P is inside and Q is outside, then we compute R, the clip point, and output P and R. If P is outside and Q

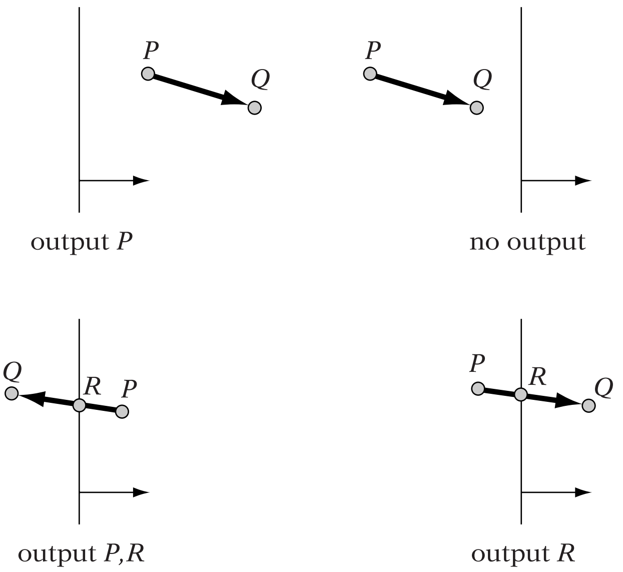


FIGURE 5.24 Four possible cases of clipping an edge against a plane.

is inside, then we compute R and output just R—as before, Q will be output as the start of the next edge. The sequence of vertices generated as output will be the vertices of our clipped polygon.

We now have enough information to build a class for clipping vertices, which we'll call `IvClipper`. We can define this as

```
class IvClipper
{
public:
    IvClipper()
    {
        mFirstVertex = true;
    }
    ~IvClipper();

    void ClipVertex( const IvVector3& end )

    inline void StartClip() { mFirstVertex = true; }
    inline void SetPlane( const IvPlane& plane ) {mPlane = plane;}

private:
    IvPlane   mPlane;   // current clipping plane
    IvVector3 mStart;   // current edge start vertex
    float     mBCStart; // current edge start boundary condition
```

```
        bool  mStartInside; // whether current start vertex is inside
        bool  mFirstVertex; // whether expected vertex is start vertex
    };
```

Note that `IvClipper::ClipVertex()` takes only one argument: the end vertex of the edge. If we send the vertex pair for each edge down to the clipper, we'll end up duplicating computations. For example, if we clip P_0 and P_1, and then P_1 and P_2, we have to determine whether P_1 is inside or outside twice. Rather than do that, we'll feed each vertex in order to the clipper. By storing the previous vertex (`mStart`) and its plane test information (`mBCStart`) in our `IvClipper` class, we need to calculate data only for the current vertex. Of course, we'll need to prime the pipeline by sending in the first vertex, not treating it as part of an edge, and just storing its boundary information.

Using this, clipping an edge based on the current vertex might look like

```
    void IvClipper::ClipVertex( const IvVector3& end )
    {
      float BCend = mPlane.Test(end);
      bool endInside = ( BCend >= 0 );
      if (!mFirstVertex)
      {
         // if one of the points is inside
         if ( mStartInside || endInside )
         {
           // if the start is inside, just output it
           if (mStartInside)
              Output( mStart );
           // if one of them is outside, output clip point
           if ( !(mStartInside && endInside) )
           {
              if (endInside)
              {
                float t = BCend/(BCend - mBCStart);
                Output( end - t*(end - mStart) );
              }
              else
              {
                float t = mBCStart/(mBCStart - BCend);
                Output( mStart + t*(end - mStart) );
              }
           }
         }
      }
```

```
        mStart = end;
        mBCStart = BCend;
        mStartInside = endInside;
        mFirstVertex = false;
    }
```

Note that we generate *t* in the same direction for both clipping cases — from inside to outside. Polygons will often share edges. If we were to clip the same edge for two neighboring polygons in different directions, we may end up with two slightly different points due to floating-point error. This will lead to visible cracks in our geometry, which is not desirable. Interpolating from inside to outside for both cases avoids this situation.

To clip against the view frustum, or any other convex volume, we need to clip against each frustum plane. The output from clipping against one plane becomes the input for clipping against the next, creating a clipping pipeline. In practice, we don't store the entire clipped polygon, but pass each output vertex down as we generate it. The current output vertex and the previous one are treated as the edge to be clipped by the next plane. The `Output()` call above becomes a `ClipVertex()` for the next stage.

Note that we have only handled generation of new positions at the clip boundary. There are other parameters that we can associate with an edge vertex, such as colors, normals, and texture coordinates (we'll discuss exactly what these are in Chapters 6–8). These will have to be clipped against the boundary as well. We use the same *t* value when clipping these parameters, so the clip part of our previous algorithm might become

```
    // if one of them is outside, output clip vertex
    if ( !(mStartInside && endInside) )
    {
        ...
        clipPosition = startPosition + t*(endPosition - startPosition);
        clipColor = startColor + t*(endColor - startColor);
        clipTexture = startTexture + t*(endTexture - startTexture);
        // Output new clip vertex
    }
```

This is only one example of a clipping algorithm. In most cases, it won't be necessary to write any code to do clipping. The hardware will handle any clipping that needs to be done for rendering. However, for those who have the need or interest, other examples of clipping algorithms are the Liang-Barsky [73] , Cohen-Sutherland (found in [36] as well as other graphics texts), and Cyrus-Beck [22] methods. Blinn [11] describes an algorithm for lines that combines many of the features from the previously mentioned techniques; with minor modifications it can be made to work with polygons.

5.4.4 Homogeneous Clipping

In the presentation above, we clip against a general plane. When projecting, however, Blinn and Newell [9] noted that we can simplify our clipping by taking advantage of some properties of our projected points prior to the division by w. Recall that after the division by w, the visible points will have normalized device coordinates lying in the interval $[-1, 1]$, or

$$\begin{aligned}-1 &\le x/w \le 1\\ -1 &\le y/w \le 1\\ -1 &\le z/w \le 1\end{aligned}$$

Multiplying these equations by w provides the intervals prior to the w division:

$$\begin{aligned}-w &\le x \le w\\ -w &\le y \le w\\ -w &\le z \le w\end{aligned}$$

In other words, the visible points are bounded by the six planes:

$$\begin{aligned}w &= x\\ w &= -x\\ w &= y\\ w &= -y\\ w &= z\\ w &= -z\end{aligned}$$

Instead of clipping our points against general planes in the world frame or view frame, we can clip our points against these simplified planes in $\mathbb{R}P^3$ space. For example, the plane test for $w = x$ is $w - x$. The full set of plane tests for a point P are

$$\begin{aligned}BCP_{-x} &= w + x\\ BCP_x &= w - x\end{aligned}$$

$$BCP_{-y} = w + y$$
$$BCP_y = w - y$$
$$BCP_{-z} = w + z$$
$$BCP_z = w - z$$

The previous clipping algorithm can be used, with these plane tests replacing the `IvPlane::Test()` call. While these tests are cheaper to compute in software, their great advantage is that since they don't vary with the projection they can be built directly into hardware, making the clipping process very fast. Because of this, OpenGL supports a two-stage clipping process. First of all, a point is transformed into the view frame. Then it is clipped against any user-defined clipping planes set by the `glClippingPlane()` call. Then the point is multiplied by the projection matrix, clipped in homogeneous space, and finally the coordinates are divided by w to place the clipped point in the NDC frame.

There is one wrinkle to homogeneous clipping, however. Figure 5.25 shows the visible region for the x-coordinate in homogeneous space. However, our plane tests will clip to the upper triangle region of that hourglass shape — any points that lie in the lower region will be inadvertently removed. With the projections that we have defined, this will happen only if we use a negative value for the w value of our points. And since we've chosen 1 as the standard w value for points, this shouldn't happen. However, if you do have points that for some reason have negative w values, Blinn [11] recommends the following procedure: transform, clip, and render your points normally.

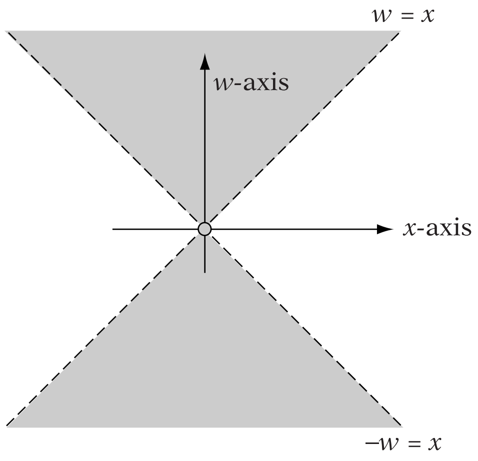


FIGURE 5.25 Homogeneous clip regions for NDC interval [−1,1].

Then multiply your projection matrix by −1, then transform, clip, and render again.

5.5 Screen Transformation

Now that we've covered projection and clipping, our final step in transforming our object in preparation for rendering is to map its geometric data from the NDC frame to the screen or device frame. This could represent a mapping to the full display, a window within the display, or an offscreen pixel buffer.

Remember that our coordinates in the NDC frame range from a lower left corner of $(-1, -1)$ to an upper right corner of $(1, 1)$. Real device space coordinates usually range from an upper left corner $(0, 0)$ to lower right corner (w_s, h_s), where w_s (screen width) and h_s (screen height) are usually not the same. In addition, in screen space the y axis is commonly flipped so that y values increase as we move down the screen. Some windowing systems allow you to use the standard y direction, but we'll assume the default (Figure 5.26).

What we'll need to do is map our NDC area to our screen area (Figure 5.27). This consists of scaling it to the same size as the screen, flipping our y direction, and then translating it so that the upper left hand corner becomes the origin.

Let's begin by considering only the y direction, because it has the special case of the axis flip. The first step is scaling it. The NDC window is 2 units high, whereas the screen space window is h_s high, so we divide by 2 to scale

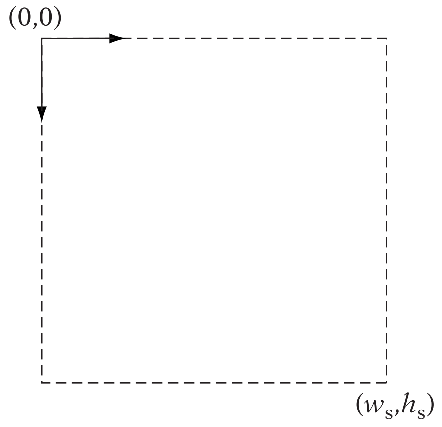


Figure 5.26 View window in standard screen space frame.

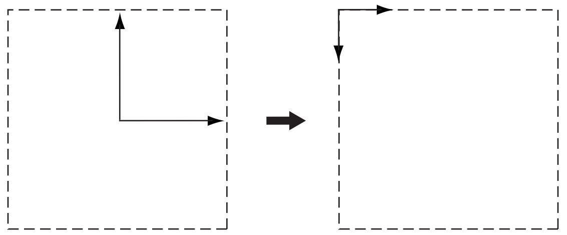


FIGURE 5.27 Mapping NDC space to screen space.

the NDC window to unit height, and then multiply by h_s to scale to screen height:

$$y' = \frac{h_s}{2} y_{ndc}$$

Since we're still centered around the origin, we can do the axis flip by just negating:

$$y'' = -\frac{h_s}{2} y_{ndc}$$

Finally, we need to translate downwards (which is now the positive y direction) to map the top of the screen to the origin. Since we're already centered on the origin, we need to translate only half the screen height, so:

$$y_s = -\frac{h_s}{2} y_{ndc} + \frac{h_s}{2}$$

Another way of thinking of the translation is that we want to map the extreme point $-h_s/2$ to 0, so we need to add $h_s/2$.

A similar process, without the axis flip, gives us our x transformation:

$$x_s = \frac{w_s}{2} x_{ndc} + \frac{w_s}{2}$$

This assumes that we want to cover the entire screen with our view window. In some cases, for example in a split-screen console game, we want to cover only a portion of the screen. Again, we'll have a width and height of our screen space area, w_s and h_s, but now we'll have a different upper left corner position for our area: (s_x, s_y). The first part of the process is the same; we scale the NDC window to our screen space window and flip the y-axis. Now, however, we want to map $(-w_s/2, -h_s/2)$ to (s_x, s_y), instead of $(0, 0)$. The final translation will be $(w_s/2 + s_x, h_s/2 + s_y)$. This gives us our generalized screen transformation in xy as

$$x_s = \frac{w_s}{2}x_{ndc} + \frac{w_s}{2} + s_x \tag{5.5}$$

$$y_s = -\frac{h_s}{2}y_{ndc} + \frac{h_s}{2} + s_y \tag{5.6}$$

Our z coordinate is a special case. As mentioned, we'll want to use z for depth testing, which means that we'd really prefer it to range from 0 to d_s, where d_s is usually 1. This mapping from $[-1, 1]$ to $[0, d_s]$ is

$$z_s = \frac{d_s}{2}z_{ndc} + \frac{d_s}{2} \tag{5.7}$$

We can, of course, express this as a matrix:

$$\mathbf{M}_{ndc \rightarrow screen} = \begin{bmatrix} \frac{w_s}{2} & 0 & 0 & \frac{w_s}{2} + s_x \\ 0 & -\frac{h_s}{2} & 0 & \frac{h_s}{2} + s_y \\ 0 & 0 & \frac{d_s}{2} & \frac{d_s}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Most of the time it is expected that the aspect ratio a chosen in the projection will match the aspect ratio w_s/h_s of the final screen transformation. Otherwise, the resulting image will be distorted. For example, if we use a square aspect ratio ($a = 1.0$) for the projection and a standard aspect ratio of 4:3 for the screen transformation, the image will appear compressed in the y direction. If your image does not quite look right, it is good practice to ensure that these two values are the same.

An exception to this practice arises when your final display has a different aspect ratio than the offscreen buffers that you're using for rendering. For example, NTSC televisions have 448 scan lines, with 640 analog pixels per scan line, so it is common practice to render to a 640×448 area and then send that to the NTSC converter to be displayed. Using the offscreen buffer size would give an aspect ratio of 10:7. But the actual television screen has a 4:3 aspect ratio, so the resulting image will be distorted, producing stretching in the y direction. The solution is to set $a = 4/3$ despite the aspect ratio of the offscreen buffer. The image in the offscreen buffer will be compressed in the y direction, but then will be proportionally stretched in the y direction when the image is displayed on the television, thereby producing the correct result.

5.6 Picking

Source Code Demo Picking

Now that we understand the mathematics necessary for transforming an object from world coordinates to screen coordinates, we can consider the opposite case. In our game we may have enemy objects that we'll want to target. The interface we have chosen involves tracking them with our mouse and then clicking on the screen. The problem is, How do we take our click location and use that to detect which object we've selected, if any? We need a method that takes our 2D screen coordinates and turns them into a form that we can use to detect object intersection in 3D game space.

For the purposes of discussion, we'll assume that we are using the basic OpenGL perspective matrix. Similar derivations can be created using other projections. Figure 5.28 is yet another cross section showing our problem. Once again, we have our view frustum, with our top and bottom clipping planes, our projection plane, and our near and far planes. Point P_S indicates our click location on the projection plane. If we draw a ray (known as a pick ray) from the view position through P_S, we pass through every point that lies underneath our click location. So to determine which object we have clicked on, we need only generate this point on the projection plane, create the specific ray, and then test each object for intersection with the ray. The closest object to the eye will be the object we're seeking.

To generate our point on the projection plane, we'll have to find a method for going backwards from screen space into view space. To do this we'll have to find a means to "invert" our projection. Matrix inversion seems like the solution, but it is not the way to go. The standard projection matrix has zeros in the right-most column, so it's not invertible. But even using the z-depth projection matrix doesn't help us because (a) the reciprocal divide makes the

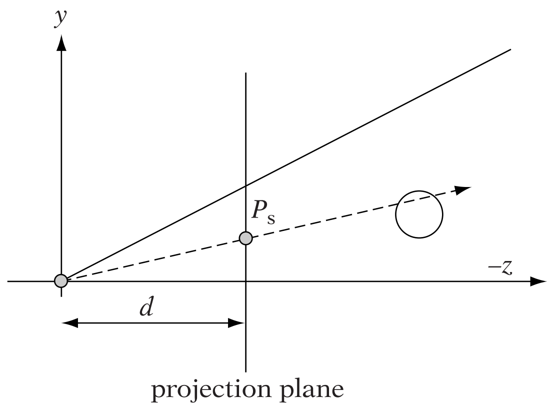


Figure 5.28 Pick ray.

process nonlinear and (b) in any case our click point doesn't have a z value to plug into the inversion.

Instead, we begin by transforming our screen space point (x_s, y_s) to an NDC space point (x_{ndc}, y_{ndc}). Since our NDC to screen space transform is affine, this is easy enough: we need only invert our previous equations 5.5 and 5.6. That gives us

$$\begin{aligned} x_{ndc} &= \frac{2(x_s - s_x)}{w_s} - 1 \\ y_{ndc} &= -\frac{2(y_s - s_y)}{h_s} + 1 \end{aligned}$$

Now the tricky part. We need to transform our point in the NDC frame to the view frame. We'll begin by computing our z_v value. Looking at Figure 5.28 again, this is straightforward enough. We'll assume that our point lies on the projection plane so the z value is just the z location of the plane or $-d$. This leaves our x- and y-coordinates to be transformed. Again, since our view region covers a rectangle defined by the range $[-a, a]$ (recall that a is our aspect ratio) in the x direction and the range $[-1, 1]$ in the y direction, we only need to scale to get the final point. The view window in the NDC frame ranges from $[-1, 1]$, so no scale is needed in the y direction and we scale by a in the x direction. Our final screen space to view space equations are

$$\begin{aligned} x_v &= \frac{2a}{w_s}(x_s - s_x) - 1 \\ y_v &= -\frac{2}{h_s}(y_s - s_y) + 1 \\ z_v &= -d \end{aligned}$$

And since this is a system of linear equations, we can express this as a 3×3 matrix as follows:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \begin{bmatrix} \frac{2a}{w_s} & 0 & -\frac{2a}{w_s}s_x - 1 \\ 0 & -\frac{2}{h_s} & \frac{2}{h_s}s_y + 1 \\ 0 & 0 & -d \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

From here we have a choice. We can try to detect intersection with an object in the view frame, we can detect in the world frame, or we can detect in the object's local frame. The first involves transforming every object into the view frame and then testing against our pick ray. The second involves transforming our pick ray into the world frame and testing against the world coordinates of each object. If we're using a scene graph, we're already pre-generating our world location and bounding information. So if we're only concerned with testing for intersection against bounding information, it can be more efficient to go with testing in world space. However, usually we test in local space so we can check for intersection within the frame of the stored model vertices, without having to transform them into the world frame or the view frame.

In order to do that, we'll have to transform our view space point by the inverse of the viewing transformation. Unlike the perspective transformation, however, this inverse is much easier to compute. Recall that since the view transformation is an affine matrix, we can invert it to get the view-to-world matrix $\mathbf{M}_{view \to world}$. So multiplying $\mathbf{M}_{view \to world}$ by our click point in the view frame gives us our point in world coordinates:

$$P_w = \mathbf{M}_{view \to world} \cdot P_v$$

We can transform this and our view position E from world coordinates into local coordinates by multiplying by the inverse of the local-to-world matrix:

$$P_l = \mathbf{M}_{world \to local} \cdot P_w$$
$$E_l = \mathbf{M}_{world \to local} \cdot E$$

Then the formula for our pick ray in local space is

$$R(t) = E_l + t(P_l - E_l)$$

We can now use this ray in combination with our models to find the particular object the user has clicked on. Chapter 11 discusses how to determine intersection between a ray and an object and other intersection problems.

5.7 Management of Viewing Transformations

Source Code
Library
IvEngine
Filename
IvGLHelp

Up to this point we have presented a set of transformations and corresponding matrices without giving some sense of how they would fit into a game engine. While the thrust of this book is not about writing renderers, we can still provide a general sense of how some renderers and APIs manage these matrices, and how to set transformations for a standard API.

The view, projection, and screen transformations change only if the camera is moved. As this happens rarely, these matrices are usually computed once, stored, and then concatenated with the new world transformation every time a new object instance is rendered. How this is handled depends on the API used. The most direct approach is to concatenate the newly set world transform matrix with the others, creating a single transformation all the way from local space to prehomogeneous divide screen space:

$$M_{local \to screen} = M_{ndc \to screen} \cdot M_{projection} \cdot M_{world \to view} \cdot M_{local \to world}$$

Multiplying by this single matrix and then performing three homogeneous divisions per vertex generates the screen coordinates for the object. This is extremely efficient, but ignores any clipping we might need to do. In this case, we can concatenate up to homogeneous space, also known as clip space:

$$M_{local \to clip} = M_{projection} \cdot M_{world \to view} \cdot M_{local \to world}$$

Then we transform our vertices by this matrix, clip against the view frustum, perform the homogeneous divide, and either calculate the screen coordinates using equations 5.5–5.7 or multiply by the NDC to screen matrix, as before.

With more complex renderers, we end up separating the transformations further. For example, OpenGL handles lighting and some clipping prior to projection, so it has separate `GL_MODELVIEW` and `GL_PROJECTION` matrix stacks, to which the appropriate matrices have to be concatenated. The vertices are transformed by the top matrix in the `GL_MODELVIEW` stack, lighting and user-defined clipping is computed, and then the vertices are transformed by the top matrix in the `GL_PROJECTION` matrix. The resulting vertices are clipped in homogeneous space, the reciprocal divide is performed as before, and finally they are transformed to screen space.

In our program, we can set the view and projection matrices in OpenGL by

```
IvMatrix44 projection, viewTransform;

// compute projection and view transformation
...

// set in OpenGL
glMatrixMode(GL_PROJECTION);
glLoadMatrix( projection );

glMatrixMode(GL_MODELVIEW);
glLoadMatrix( viewTransform );
```

And when we render an object, concatenating the world matrix can be done by

```
glMatrixMode(GL_MODELVIEW);

// push copy of view matrix to top of stack
```

```
glPushMatrix();

// multiply by world matrix
glMultMatrix( worldTransform );

// render
...

// pop to view matrix
glPopMatrix();
```

The push/pop calls provide a means for storing the view transformation without reloading it into the stack. The call `glPushMatrix()` copies the current matrix—in this case, the view matrix—to a new entry on the top of the stack. The subsequent `glMultMatrix()` will postmultiply the world matrix by the copy of the view matrix at the top of the stack. The resulting local-to-view matrix will be used to transform the vertices of our object. Finally, `glPopMatrix()` removes the current matrix from the top of the stack, restoring the view transformation as the top matrix. The effect is to save the view transformation, multiply by the world transformation and use the result to transform the vertices, and then restore the original view transformation.

Direct3D takes this one step further, and manages storage of the view transformation by having three separate matrices: one each for the projective, view, and world transformations. These can be set by using the `IDirect3DDevice*::SetTransform()` method, and any concatenation is handled internally to the API.

This leaves the NDC to screen space transformation. Usually the graphics API will not require a matrix but will perform this operation directly. In the xy directions the user is only expected to provide the dimensions and position of the screen window area, also known as the viewport. In OpenGL this is set by using the call `glViewport()`. For the z direction, OpenGL provides a function `glDepthRange()`, which maps $[-1, 1]$ to $[near, far]$, where the defaults for $near$ and far are 0 and 1. Similar methods are available for other APIs.

In our case we have decided not to overly complicate things and are providing simple convenience routines:

```
::IvSetWorldMatrix()
::IvSetViewMatrix()
::IvSetProjectionMatrix()
::IvSetViewport()
```

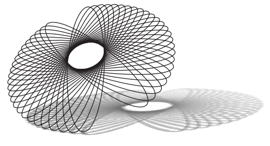
which act as wrappers for the OpenGL calls described.

5.8 Chapter Summary

Manipulating objects in the world frame is only as useful as the techniques that we use to present that data. In this chapter we have discussed the viewing, projection, and screen transformations necessary for rendering objects on a screen or image. While we have focused on OpenGL as our rendering API, the same principles apply to Direct3D or any other rendering system. We transform the world to the perspective of a virtual viewer, project it to a view plane, and then scale and translate the result to fit our final display. We also covered how to reverse those transformations to allow one to select an object in view or world space by clicking on the screen. In the following chapters, we will discuss how to use the data generated by these transformations to actually set pixels on the screen.

For those who are interested in reading further, most graphics textbooks — such as Möller and Haines [79] and Foley and van Dam [36] — describe the graphics pipeline in great detail. In addition, one of Blinn's collections [11] is almost entirely dedicated to this subject. Various culling techniques are discussed in Möller and Haines [79], as well as Eberly [27]. Finally, the *OpenGL Programming Guide* [83] discusses the particular implementation of the graphics pipeline used in OpenGL.

Chapter 6 Geometry, Shading, and Texturing



6.1 Introduction

Having discussed in detail in the preceding chapters how to represent, transform, view, and animate geometry, the next three chapters form a sequence that describes the second half of the "rendering pipeline." The second half of the rendering pipeline is specifically focused on visual matters: the representation, computation, and usage of color.

This chapter will discuss how we connect the points we have been transforming and projecting to form solid surfaces, as well as the extra information we use to represent the unique appearance of each surface. All visual representations of geometry require the computation of colors; this chapter will discuss the data structures used to store colors and perform basic color computations. It will also discuss methods used to assign static colors to geometry, including image-based texturing.

Chapter 7 will detail common, real-time 3D approximations to dynamic lighting, including light sources, surface materials, lighting models, and their applications. Chapter 7 will complete our discussion of the so-called geometry pipeline, having taken our objects from model space to screen space and from colorless vectors to lit, textured surfaces.

As the concluding chapter in this sequence, Chapter 8 will cover the final step in the overall rendering pipeline — rasterization, or the method of determining how to draw the colored surfaces to pixels on the display device. This will complete the discussion of the rendering pipeline.

Each section in these chapters will relate the basic OpenGL concepts, data structures, and functions that affect the creation, rendering, and coloring of geometry. As we move from geometry representation through shading, lighting, and rasterization, OpenGL information will become increasingly frequent, as the implementation of the final stages of the rendering pipeline are very much system-dependent. However, the basic rendering concepts discussed will apply to most rendering systems.

As a note, we use the phrase *OpenGL implementation* to refer to the underlying software or "driver" that maps our application calls to OpenGL into commands for a particular piece of graphics hardware. The OpenGL implementation for a particular piece of graphics hardware is generally supplied with the device by the hardware vendor. It is not something that users of OpenGL will have to write or even use directly. In fact, the main purpose of OpenGL is to provide a standard interface on top of these widely varying hardware/software 3D systems.

6.2 Color Representation

6.2.1 The RGB Color Model

To represent color, this chapter will use the additive *RGB* (red, green, blue) color model that is almost universal in real-time 3D systems. Approximating the physiology of the human visual system (which is tuned to perceive color based on three primitives that are close to these red, green, and blue colors), the RGB system is used in all common display devices used by real-time 3D graphics systems. Color cathode ray tubes (or CRTs, such as traditional televisions and computer monitors), flat-panel liquid crystal displays (LCDs), plasma displays, and video projector systems are all based upon the additive RGB system. While some colors cannot be accurately displayed using the RGB model, it does support a very wide range of colors, as proven by the remarkable color range and accuracy of modern television and computer displays. For a detailed discussion of color vision and the basis of the red, green, blue color model, see [74].

The RGB color model involves mixing different amounts of three predefined *primary* colors of light. These carefully defined primary colors are each named by the named colors that most closely match them; red, green, and blue. By mixing independently controlled levels of these three colors of light, a wide range of *brightnesses*, *tones*, and *shades* may be created. For example, a few very general color mixes and the named color that results are

$$\text{Equal parts red and green} \rightarrow \text{yellow}$$

Two parts red, one part green → orange

Equal parts of all three colors → black, gray, or white

Note that no mention of the exact levels of these colors is given. Brighter or darker versions of these colors can be created by changing the overall amounts of all three primary components. The next few sections will define much more specifically how we build and represent colors using this method.

As mentioned, the levels of each of these three primary colors are independent. In a sense, this is similar to a subset of $\mathbb{R}^3$, but with a "basis" consisting of the red, green, and blue "axes," or components. While these can be thought of as a "basis" for our display device's color space, they are not a basis in any true sense for color in general.

Monochrome or grayscale displays are quite similar to color displays, but have only a single color component instead of three. For the purposes of this chapter, we will discuss only methods that are designed to supply full-color displays with the data they require. The monochrome situation may be simulated by using only gray values between black and white.

6.2.2 Colors as "Vectors"

The representation of colors as amounts of independent red, green, and blue primaries is conceptually very similar to our ideas of a vector space. In this case, our "basis vectors" represent the three color primaries. As we shall see, while this is a useful implementation method, the behavior of colors does not always map directly into the concept of a real vector space. However, many of the concepts of real vector spaces are useful in describing color representation and operations.

Our colors will be represented by 3-vectors, with the following basis vectors:

$$(1, 0, 0) \rightarrow red$$

$$(0, 1, 0) \rightarrow green$$

$$(0, 0, 1) \rightarrow blue$$

Often, as a form of shorthand, we will refer to the red component of a color $\mathbf{c}$ as $\mathbf{c}_r$ and to the green and blue components as $\mathbf{c}_g$ and $\mathbf{c}_b$, respectively.

The following sections will describe some of the vector operations (and vectorlike operations) we will apply to colors, as well as discussions of how these abstract color vectors map onto their final destinations, namely hardware display devices.

6.2.3 Operations on Colors

Adding RGB colors is done using a method equivalent to vector addition; the colors are added componentwise. This has the same effect as combining the light from two light sources whose colors are equal to those of the operands; for example, adding red ($\mathbf{r} = (1, 0, 0)$) and green ($\mathbf{g} = (0, 1, 0)$) gives yellow:

$$\mathbf{r} + \mathbf{g} = (1, 0, 0) + (0, 1, 0) = (1, 1, 0)$$

The operation of adding colors will be used through our lighting computations to represent the addition of light from multiple light sources and to add the multiple forms of light that each source can apply to a surface.

Scalar multiplication of RGB colors ($s\mathbf{c}$) is computed in the same way as with vectors, multiplying the scalar times each component, and is ubiquitous in lighting and other color computations. It has the result of increasing ($s > 1.0$) or decreasing ($s < 1.0$) the luminance of the color by the amount of the scalar factor. Scalar multiplication is most frequently used to represent light attenuation due to various physical and geometric lighting properties.

One important vector operation that is used somewhat rarely with colors is vector length. While it might seem that vector length would be an excellent (if expensive) way to compute the "luminance" of a color, the nature of human color perception does not match the Euclidean norm of the linear RGB color space. Luminance is a "norm" that is affected by the device used to display the color, human physiology, and mathematics. The human eye is most sensitive to green, then red, and finally to blue. As a result, the equal weighting given to all components by the Euclidean norm means that blue contributes to the Euclidean norm far more than it contributes to luminance.

Although there are numerous methods used to compute the luminance of RGB colors as displayed on a screen, a common method for modern CRT screens (assuming nonnegative color components) is

$$luminance(\mathbf{c}) = 0.2125\mathbf{c}_r + 0.7154\mathbf{c}_g + 0.0721\mathbf{c}_b$$

The three color-space transformation coefficients used to scale the color components are basically constant for modern, standard CRT screens but do not neccessarily apply to television screens, which use a different set of luminance conversions. Discussion of these may be found in [90]. Note that luminance is *not* equivalent to perceived brightness. The luminance as we've computed it is linear with respect to the source linear RGB values. Brightness as perceived by the human visual system is nonlinear and subject to the overall brightness of the viewing environment, as well as the viewer's adaptation to it. See [20] for a related discussion of the physiology of human visual perception.

An operation that is rarely applied to vectors but is used very frequently with colors is componentwise multiplication. Componentwise multiplication takes two colors as operands and produces another color as its result. We will represent the operation of componentwise multiplication of colors as "·", or in shorthand by placing the colors next to one another (as we would multiply scalars), and the operation is defined as follows:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{ab} = (\mathbf{a}_r \mathbf{b}_r, \mathbf{a}_g \mathbf{b}_g, \mathbf{a}_b \mathbf{b}_b)$$

This operation is often used to represent the *filtering* of one color of light through an object of another color. In such a situation, one operand is assumed to be the light color, while the other operand is assumed to be the amount of light of each component that is passed by the filter. Another use of componentwise color multiplication is to represent the reflection of light from a surface — one color represents the incoming light and the other represents the amount of each component that the given surface reflects. We will use this frequently in the next chapter when computing lighting. For example, a color $\mathbf{c}$ and a filter (or surface) $\mathbf{f} = (1, 1, 1)$ result in

$$\mathbf{cf} = \mathbf{c}$$

In this case the filter was a perfectly efficient piece of clear glass — all light passed through (or a perfect mirror, with all light reflecting in the surface example). However, if the filter color were to have been $\mathbf{f} = (1, 0, 0)$, the result would be

$$\mathbf{cf} = (\mathbf{c}_r, 0, 0)$$

or the equivalent of a pure red filter; only the red component of the light was passed, while all other light was blocked. This operation will be used constantly in color lighting computations.

6.2.4 Color Range Limitation

The theoretical RGB color space is semi-infinite in all three axes. There is an absolute zero value for each component, bounding the negative directions, but the positive directions are (theoretically) unbounded. The reality of physical display devices imposes severe limitations on the color space. In fact, when limited to the colors that can be represented by a specific display device, the RGB color space is not infinite in any direction. Real display devices for real-time 3D, such as CRTs (standard "tube" monitors), LCD panel displays, and video projectors all have limits of both brightness and darkness in each color component; these are basic physical limitations of the technologies that these displays use to emit light. For details on the functionality and limitations of display device hardware, Hearn and Baker [56] detail many popular display devices.

Displays generally have minimum and maximum brightnesses in each of their three color axes, which can be represented as the color vector containing all three minima $\mathbf{c}_{min}$ and the color vector containing all three maxima $\mathbf{c}_{max}$. For all of these displays, some form of "black," $\mathbf{k}$ (very low, often nonzero, roughly equal amounts of all components) and "white," $\mathbf{w}$ (very high, roughly equal amounts of all components) form the minimum and maximum points in the RGB spaces of these devices. Generally, it is useful to have $\mathbf{c}_{min} = \mathbf{k}$ and $\mathbf{c}_{max} = \mathbf{w}$. While it might be possible to create extrema that are not pure black and white, these are unlikely to be useful in a general display device. For example, most applications would have no use for a $\mathbf{c}_{max}$ that was a bright, saturated red.

Every display device is likely to have different exact values for $\mathbf{k}$ and $\mathbf{w}$, so it is convenient to use a standard color space for all devices as sort of "normalized device color (or NDC)" coordinates. This color space is built such that

$$(0, 0, 0) \rightarrow \mathbf{k}$$
$$(1, 1, 1) \rightarrow \mathbf{w}$$

The general mapping from these device-independent colors to the range of the display is then

$$(r, g, b) \rightarrow (\mathbf{k}_r + r(\mathbf{w}_r - \mathbf{k}_r), \mathbf{k}_g + g(\mathbf{w}_g - \mathbf{k}_g), \mathbf{k}_b + b(\mathbf{w}_b - \mathbf{k}_b))$$

This kind of device mapping is normally handled by the device driver or low-level graphics API, and as a result the rest of this chapter and the following chapter will work in these normalized color coordinates. This space defines an RGB "color cube," with black at the origin, white at $(1, 1, 1)$, gray levels down the main diagonal between them (a, a, a), and the other six corners representing pure, maximal red $(1, 0, 0)$, green $(0, 1, 0)$, blue $(0, 0, 1)$, cyan $(0, 1, 1)$, magenta $(1, 0, 1)$, and yellow $(1, 1, 0)$.

Although devices cannot generally display colors outside of the range defined by the $(0, 0, 0) \ldots (1, 1, 1)$ cube, colors outside of this cube are often seen during intermediate color computations such as lighting. In fact, the very nature of lighting can lead to final colors with components outside of the $(1, 1, 1)$ limit. During lighting computations, these are generally allowed, but prior to assigning final colors to the screen, all colors must be within the normalized cube. This requires either the hardware, the device driver software, or the application to somehow limit the values of colors that do not fall within the "safe" unit cube.

The simplest and easiest method is to clamp the color on a per-component basis:

$$safe(\mathbf{c}) = (clamp(\mathbf{c}_r), clamp(\mathbf{c}_g), clamp(\mathbf{c}_b))$$

where

$$clamp(x) = max(min(x, 1.0), 0.0)$$

However, it should be noted that such an operation can cause significant perceptual changes to the color. For example, the color (1.0, 1.0, 10.0) is predominantly blue, but its clamped version is pure white (1.0, 1.0, 1.0). In general, clamping a color can lead to the color becoming less *saturated,* or less colorful. While this might seem unsatisfactory, it can actually be beneficial in lighting, as it tends to make overly bright objects appear to "wash out," an effect that can appear rather natural perceptually.

Another, more computationally expensive method is to rescale all three color components of any color with a component greater than 1.0 such that the maximal component is 1.0. This may be written as

$$safe(\mathbf{c}) = \frac{(max(\mathbf{c}_r, 0), max(\mathbf{c}_g, 0), max(\mathbf{c}_b, 0))}{max(\mathbf{c}_r, \mathbf{c}_g, \mathbf{c}_b, 1)}$$

Note the appearance of 1 in the *max* function in the denominator to ensure that colors already in the unit cube will not change—it will never increase the color components. While this method does tend to avoid changing the overall saturation of the color, it can produce some unexpected results. The most common issue is that extremely bright colors that are scaled back into range can actually end up appearing darker than colors that did not require scaling. For example, comparing the two colors $\mathbf{a} = (1, 1, 0)$ and $\mathbf{b} = (10, 5, 0)$, we find that after scaling, $\mathbf{b} = (1, 0.5, 0)$, which is significantly darker than $\mathbf{a}$.

Scaling works best when it is applied equally to all colors in a scene, not to each color individually. There are numerous methods for this, but one such method involves finding the maximum color component of any object in the scene, and scaling all colors equally such that this maximum maps to 1.0. This is somewhat similar to a camera's auto-exposure system. By scaling the entire scene by a single scalar, color ratios between objects in the scene are preserved.

6.2.5 Alpha Values

Frequently, RGB colors are augmented with a fourth component, called *Alpha*. Such colors are often written as *RGBA* colors. Unlike the other three components, the alpha component does not represent a specific color basis, but rather defines how the combined color interacts with other colors. The most frequent use of the alpha component is as an opacity value, which defines how much of the surface's color is controlled by the surface itself and how much is controlled by the colors of objects that are behind the given surface. When alpha is at its maximum (we will define this as 1.0), then the color of the surface is independent of any objects behind it. The red, green, and blue components of the surface color may be used directly; for example, in representing a solid concrete wall. At its minimum (0.0), the RGB color of the surface is ignored and the object is invisible, as with a pane of clear glass for instance. At an intermediate alpha value such as 0.5, the colors of the two objects are blended together; in the case of alpha equaling 0.5, the resulting color will be the componentwise average of the colors of the surface and the object behind the surface.

For the most part, alpha will be treated like any other color component until rasterization. We will discuss the uses of the alpha value (known as *alpha blending*) in Chapter 8 on rasterization. In a few cases, OpenGL handles alpha a little differently from other color components (mention will be made of these situations as needed).

6.2.6 Color Storage Formats

While we have discussed color values as real numbers, floating-point storage of colors in a frame-buffer is not popular in graphics systems at this time. The most popular format is to use unsigned 8-bit values per component, leading to 3 bytes per RGB color, a system known as *24-bit color*, or in some cases, by the misnomer "true color." With an alpha value, the format becomes 32 bits per pixel, which aligns well on modern 32-bit CPU architectures. Another common format is to use 5 bits each for red and blue and 6 bits for green, a format that requires 16 bits per pixel. This system, which sometimes goes by the name *high color*, is interesting in that it includes different amounts of precision for green than for red or blue. As we've discussed, the human eye is most sensitive to green, so the additional bit in the 16-bit format is assigned to it. However, the number of pure gray values in this format is still $2^5 = 32$.

Research has shown that the human visual system (depending on lighting conditions, etc.) can perceive between 1 million and 7 million colors, which leads to the (erroneous) theory that 24-bit color display systems, with their $2^{24} \approx 16.7$ million colors are more than sufficient. While it is true that the number of different color "names" in a 24-bit system (where a color is "named" by its 24-bit RGB triple) is a greater number than the human visual system can discern, this does not take into account the fact that the colors being generated on current display devices do not map directly to the 1–7 million colors that can be discerned by the human visual system. Current display devices cannot display the entire range of colors that the human eye can discern. In addition, in some color ranges, different 24-bit color "names" appear the same to the human visual system (the colors are closer to one another than the human eye's *just noticable difference*, or JND). In other words, 24-bit color wastes precision in some ranges, while lacking sufficient precision in others. Current 24-bit "true color" display systems are not sufficient to cover the entire range of human vision, either in range or in precision. Having said this, current display devices are still *quite* convincing to the human eye and will continue to improve.

The traditional reason for using these lower-precision formats is one of storage requirements. Even 32 bits per pixel requires one-quarter the amount of storage that is needed for floating-point RGBA values. Using full floating-point numbers for output colors (the colors that are drawn to the output LCD or CRT screen) is actually overkill, due to the limitations of current display device color resolution. For example, current CRTs and LCD displays have dynamic ranges (the ratio of luminance between the brightest and darkest levels that can be displayed by the devices) of between 200:1 and 500:1. These ratios mean that current display devices cannot deliver anywhere near the eye's full range of perceived brightness or darkness. There are display technologies on the horizon that will be able to represent more than 24-bit color. At that point, device-level color representations will require more bits per component in order to avoid wasting the added precision available from these new displays. Common "next-generation" device color formats include 30-bit color (10 bits per component) and 48-bit color (16 bits per color component).

Some 3D hardware devices do support higher-resolution colors inside of the rendering pipeline, mainly due to the advent of complex *pixel shading* hardware, which allow for advanced rendering techniques. The additional bits of precision (or even a version of floating point) can be used to avoid losing precision during multi-operation color computations, but today even these hardware devices generally output to an 8-to-10 bit per component display system.

6.2.7 Colors in OpenGL

OpenGL is very flexible in terms of color representation. It can represent colors as vectors of single- or double-precision floating-point values, as well as `byte`s, `short`s, and `int`s, all either signed or unsigned. When represented as floating-point values, 0.0 and 1.0 represent the unit cube in normalized color coordinates. However, the integral types are handled a little differently. With both signed and unsigned `byte`s, `short`s, and `int`s, a zero value (all zeros) maps to our 0.0 normalized color value, and the maximum representable positive value for the format (e.g., with signed bytes this would be 127) maps to the 1.0 normalized color value.

In the case of signed integral types, this leaves the negative half of the range to map to the values −1.0 to 0.0 in normalized colors. The most negative representable value in each number format maps to −1.0. While these negative values have no use as final device colors (they are clamped to 0.0), they can be useful as source values in lighting computations.

Floating-point values are used directly in lighting computations, even if they fall outside of the [−1.0, 1.0] range, but the final resulting color will be clamped to the [0.0, 1.0] range before using it to draw the geometry.

The most common color formats used by OpenGL applications are vectors of single-precision floating point (for ease of application use and for the flexibility of range) and unsigned bytes (because they are compact and can still represent the color precision of most display devices). Colors of these two formats are set using:

```
glColor3f(GLfloat r, GLfloat g, GLfloat b);
glColor3ub(GLubyte r, GLubyte g, GLubyte b);
```

Or, for efficiency, the vector format allows a single argument of an in-memory "vector" or array of the components:

```
GLfloat floatColor[3];
// ...
glColor3fv(floatColor);

GLubyte byteColor[3];
// ...
glColor3ubv(byteColor);
```

Almost all functions in OpenGL that require colors take this form. The exact use of these functions in a larger context will be described in the next section on vertices.

All colors in OpenGL have an alpha value, either implicit or explicit. When a color is set using `glColor3*`, the alpha value is automatically set to 1.0. To set an explicit alpha value in a color, use `glColor4*` or `glColor4*v`, where the fourth argument or array element (respectively) controls the alpha component. The OpenGL Programming Guide [83] details all of the common color representations.

6.3 Points and Vertices

So far, we have discussed points as our sole geometry representation. As we begin to abstract to the higher level of a surface, points will become insufficient for representing the attributes of an object or for that matter the object itself. The first step in the move toward a way of defining an object's surface is to associate additional data with each point. Combined together (often into a single data structure), each point and its additional information form what is often called a "vertex." In a sense, a vertex is a "heavy point": a point with addition information that defines some properties of the surface around it.

6.3.1 Per-Vertex Attributes

Within a vertex, the most basic value is the position of the vertex, generally a 3D point that we will refer to as P_V in later sections.

Other than vertex position, perhaps the most basic of vertex attributes are colors. Common additions to a vertex data structure, vertex colors are used in many different ways when drawing geometry. Much of the remainder of this chapter will discuss the various ways that per-vertex colors can be assigned to geometry, as well as the different ways that these vertex colors are used to draw geometry to the screen. We will generally refer to the vertex color as C_V (and will sometimes specifically refer to the vertex alpha as A_V, even though it is technically a component of the overall color).

Another data element that can add useful information to a vertex is a vertex normal. This is a unit-length 3-vector that defines the "orientation" of the surface in an infinitely small neighborhood of the vertex. If we assume that the surface passing through the vertex is locally planar (at least in an infinitely small neighborhood of the vertex), the surface normal is the normal vector to this plane (recall the discussion of plane normal vectors from Chapter 1). Normally, this vector is defined in the same space as the vertices, generally model (or object) space. As will be seen later, the normal vector is a pivotal component in lighting computations. We will generally refer to the normal as $\hat{\mathbf{n}}_V$.

Another vertex attribute that we will use frequently later in this chapter is a texture coordinate. This will be discussed in detail in Sections 6.7–6.11 on texturing and in parts of the following two chapters; basically, they are real-valued 2-vectors (most frequently, although they may also be scalars or 3-vectors) that define the position of the vertex within a smooth parameterization of the overall surface. These are used to map two-dimensional images onto the surface in a shading process known as texturing.

Vertices in OpenGL

Source Code Demo BasicSphere

OpenGL has the notion of a "current vertex," at least when it is in the midst of drawing. While we will describe how vertices are actually drawn, for the moment we will simply introduce the concept of specifying a vertex. In order to “push” a vertex down to OpenGL, an application uses one of the functions in the set glVertex*. You can refer to OpenGL reference [83] for details, but two commonly used versions are

```
glVertex3f(GLfloat x, GLfloat y, GLfloat z);
glVertex3fv(GLfloat* vert);
```

Both of these pass a 3D (floating point) vertex position down to OpenGL. The second version assumes that X, Y, and Z are packed together in an array of floats.

When specifying vertices, there is the notion of current values for all of the possible vertex attributes (color, normal, etc.). These current values can be set using the functions glColor*, glNormal*, and so forth (see OpenGL reference text [83]). Note that these calls do not generate vertices; they only set the current value of that attribute. When glVertex* is called, it generates a vertex using the current values of color, normal, and the like. Multiple calls to glColor*, glNormal*, and so on are ignored; only the last call to each prior to a glVertex* call matters. Additional calls to these attribute functions simply waste processor cycles. The following code generates three vertices at different positions with different colors but with the same normal (pointing along the Y axis);

```
glNormal3f(0.0, 1.0f, 0.0f);
glColor3f(1.0, 0.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 1.0f);
glColor3f(0.0, 1.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glColor3f(0.0, 0.0f, 1.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
```

6.4 Surface Representation

This section will discuss another important concept used to represent and render objects in real-time 3D graphics: the concept of a surface and the most common representation of surfaces in interactive 3D systems, sets of triangles. These concepts will allow us to build realistic-looking objects from the sets of vertices that we have discussed thus far.

Chapter 1 introduced the concept of a triangle, a subset of a plane defined by the convex combination of three noncollinear points. In this chapter we will build upon this foundation and make frequent use of triangles, the normal vector to a triangle, and barycentric coordinates. A quick review of the sections of Chapter 1 covering these topics is recommended.

While most of the remainder of this chapter will focus only on the assignment of colors to objects for the purposes of rendering, the object and surface representations we will discuss are useful for far more than just rendering. Collision detection, picking, and even artificial intelligence all make use of these representations.

6.4.1 Vertices and Surface Ambiguity

Unstructured collections of vertices (sometimes called *point clouds*) generally cannot represent a surface unambiguously. For example, draw a set of 10 or so dots representing points on a piece of paper. There are numerous ways one could connect these two-dimensional points into a closed curve (a one-dimensional "surface") or even into several smaller curves. This is true even if the vertices include normal vectors, as these normal vectors only define the orientation of the surface in an infinitely small neighborhood of the vertex. We can see that without implicit or explicit additional structure, a finite set of points rarely defines an unambiguous surface.

A cloud of points that is infinitely dense on the desired surface can represent that surface. Obviously, such a directly stored collection of unstructured points would be far too large to render in real time (or even store) on a computer. We need a method of representing an infinitely dense surface of points that requires only a finite amount of representational data.

There are numerous methods of representing surfaces, including

- Parametric surfaces (see the chapters on curves and surfaces in [36]). A parametric surface is defined as a 2-dimensional subset of $\mathbb{R}^3$ such that all points on the surface are generated by $\mathbf{v} = \mathbf{f}(s, t)$, where $s, t \in \mathbb{R}$ are the parameters. Examples of parametric surfaces include bicubic "patches" of all sorts, as well as surfaces of revolution.
- Implicit surfaces (see Blinn's [10]). An implicit surface is defined as the set of all points $\mathbf{v} \in \mathbb{R}^3$ such that a given scalar-valued function $\mathbf{f}(\mathbf{v}) = c$ for a fixed constant c. Examples of these include so-called blobby objects, or "metaballs."

While each of the methods listed above can represent some subset of all possible surfaces perfectly, both methods can be complicated and/or expensive and are not suited for all surfaces. These methods can also be difficult for artists to control at the fine scale they desire—changes to one part of such a surface can have unintended effects on other parts of the surface. Also, such methods do not always lend themselves to an obvious or direct method of rendering. Finally, they cannot necessarily make direct use of the conveniently defined vertices that our geometry pipeline can generate.

6.4.2 Triangles

The most common method used to represent 3D surfaces in real-time graphics systems is simple, scalable, requires little additional information beyond the existing vertices, and allows for direct rendering algorithms; it is called approximation of surfaces with triangles, or *tessellation*. Tessellation refers not only to the process that generates a set of triangles from a surface but also to the triangles and vertices that result.

Triangles, each represented and defined by only three points on the surface, are connected point to point and edge to edge to create a locally flat ("faceted") approximation of the surface. By varying the number and density of triangles used to represent a surface, an application may make any desired trade-off between compactness/rendering speed and accuracy of representation.

One concept that we will use frequently with triangles is that of barycentric coordinates. From the discussion in Chapter 1, we know that any point in a triangle may be represented by an element of $\mathbb{R}^2$ (s, t) such that $0.0 \leq s, t \leq 1.0$. These coordinates uniquely define each point on a nondegenerate triangle (i.e., a triangle with nonzero area). We will often use barycentric coordinates as the domain when mapping functions defined across triangles, such as color.

Triangles in OpenGL

Source Code
Demo
BasicSphere

OpenGL has numerous methods for rendering triangles. The simplest (but not the most efficient computationally) is via vertex by vertex specification. Using this method, a primitive is opened with a function call, vertices are passed to OpenGL one at a time (with sets of vertices defining triangles), and then the primitive is closed. As an example, the following code draws a tetrahedron:

```
glBegin(GL_TRIANGLES);
glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 0.0f, 1.0f);

glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 0.0f, 1.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
```

```
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);

    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 1.0f);
    glEnd();
```

The function call `glBegin` starts the primitive (in this case, `GL_TRIANGLES`, which groups each set of three vertices into a triangle), the `glVertex` calls pass down the vertices, and the `glEnd` call closes the primitive. Because this method requires three OpenGL function calls per triangle, it is quite expensive. A more efficient method, indexed geometry, is detailed in Section 6.4.4.

6.4.3 Triangle Attributes

In some graphics systems, triangles can have their own attributes beyond those of the vertices that comprise the triangle. These attributes can either override or supplement the per-vertex attributes. We will describe several common per-triangle attributes. As triangles are often referred to by the term *faces* (which is a more general term that refers to a general n-sided polygon), these attributes are often called *face attributes*.

Colors are a very common per-triangle attribute. They are used in ways analogous to vertex colors, but describe a color that is applied to the entire triangle, rather than describing the color at or near a given vertex. These will be used frequently during lighting computations, especially with so-called flat, or per-triangle, shading. We will generally refer to the triangle color as C_F (for "face color").

A per-triangle normal (we will call this vector $\hat{\mathbf{n}}_T$) is typically generated directly from the plane of the triangle itself, using the method described in Chapter 1:

$$\hat{\mathbf{n}}_T = \frac{(P_{V2} - P_{V1}) \times (P_{V3} - P_{V2})}{|(P_{V2} - P_{V1}) \times (P_{V3} - P_{V2})|}$$

Since this normal is a purely geometric quantity that (along with any of the three vertices) represents the plane of the triangle, it is used in many different algorithms, including lighting, collision detection, picking, and culling (as a way of quickly determining which triangles are visible to the camera).

As an example of these applications, let us examine triangle culling. As previously discussed, triangles that fall outside of the view, either to the side or behind the camera, are generally culled out of the system (called *view frustum* *culling*) and are not considered during the latter stages of the rendering pipeline. In a similar way, triangles are often considered to be "sided"; that is, a triangle is drawn differently (or not at all), depending on whether the triangle's front or back face is currently facing the camera. Culling based on this is generally known as *backface culling*, as it culls out the triangles that are "back-facing" with respect to the camera. This can result in the culling of a large number of the triangles not already culled by the view frustum.

Backface culling is very inexpensive to compute; much less expensive than rendering the triangle. The plane defined by any of the triangle vertices (P_V) and the per-triangle normal $\hat{\mathbf{n}}_T$ define the plane of the triangle. The plane equation for the triangle is thus all points X such that

$$\begin{aligned} X : (\hat{\mathbf{n}}_T \cdot X) - c &= 0 \\ X : (\hat{\mathbf{n}}_T \cdot X) - (\hat{\mathbf{n}}_T \cdot P_V) &= 0 \\ X : \hat{\mathbf{n}}_T \cdot (X - P_V) &= 0 \end{aligned}$$

If we consider the camera's center of projection to be located at the point Q, then backface culling is simply computed by evaluating the dot product of the two vectors in the final equation and testing the sign of the result. The two vectors to be tested are the triangle normal $\hat{\mathbf{n}}_T$ and the vector from any of the triangle's vertices to the camera location ($Q - P_V$). If

$$(Q - P_V) \cdot \hat{\mathbf{n}}_T > 0$$

then the camera location Q is on the front side of the triangle, and the triangle is front-facing. For all points Q such that

$$(Q - P_V) \cdot \hat{\mathbf{n}}_T \le 0$$

then the camera location Q is on the back side of the triangle, and the triangle is back-facing (Figure 6.1a).

Backface culling can also be computed in 2D screen space (as will be discussed subsequently). While this is even less expensive to compute than 3D backface culling, the triangle must continue farther down the graphics pipeline (into screen space) before this 2D backface culling can be computed and can require more computation per triangle. In either case, the most expensive stage in the pipeline, rasterization, is skipped for back-facing triangles — generally, a significant optimization.

OpenGL and Triangle Attributes

SOURCE CODE DEMO BasicSphere

OpenGL does not include the concept of specifying per-triangle attributes explicitly. Each vertex has the option of specifying attributes such as the color and normal. However, if the application sets an attribute once and

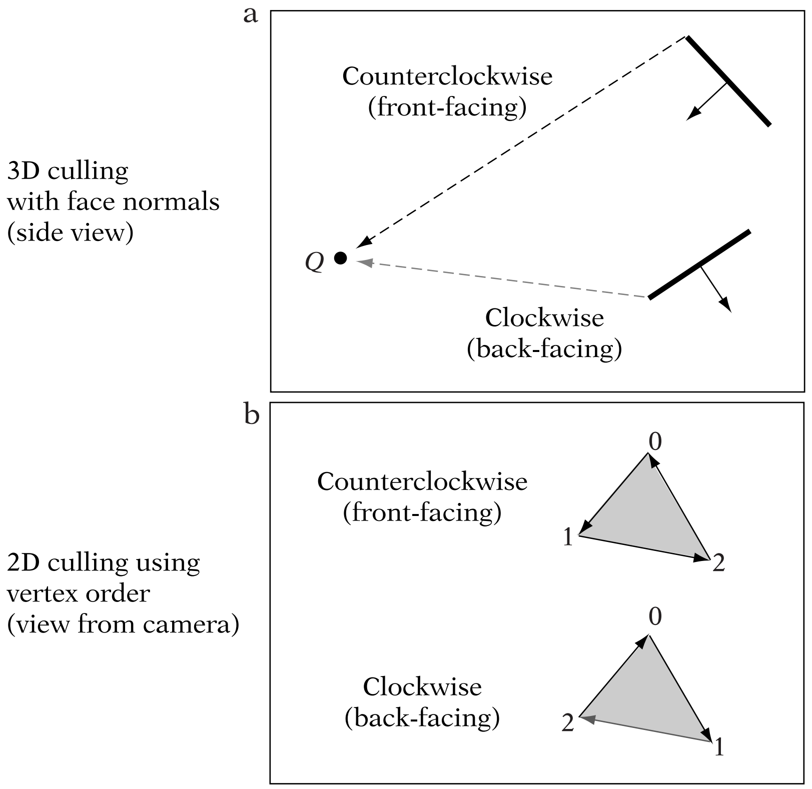


FIGURE 6.1 Culling triangles.

generates three vertices without changing that attribute, then the attribute will be constant across that triangle. In practice, the OpenGL implementation could detect this and treat the attribute as a per-triangle value internally. For example, the following triangle would have three equal normals, which in this case happens to be (and could be treated as) the per-triangle face normal.

```
glBegin(GL_TRIANGLES);

glNormal3f(0.0f, 1.0f, 0.0f);

glVertex3f(0.0f, 0.0f, 0.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 0.0f, 1.0f);

glEnd();
```

OpenGL and Triangle Culling

OpenGL allows the concept of "clockwise vs. counterclockwise" triangle vertices to be mapped as desired onto the concept of "front vs. back facing" triangles on a per-primitive basis. The linking of these concepts is handled via the function `glFrontFace`. Calling this function with an argument of `GL_CCW` will cause OpenGL to consider triangles whose vertices are ordered counterclockwise from the camera location to be front-facing. `GL_CW` sets the reverse: clockwise triangles are front-facing. The default mode is `glFrontFace(GL_CCW)`.

Note that OpenGL does *not* require objects to have normals of any kind for culling to occur; culling in OpenGL is done just prior to rasterization and is a 2D process that requires only the screen-space vertex positions. Culling in OpenGL is accomplished by determining whether the 2D, screen-space triangle vertices are in clockwise or counterclockwise order. This clockwise or counterclockwise ordering is combined with the `glFrontFace` setting just described to determine whether the given triangle is front- or back-facing. This is shown from the camera's point of view in Figure 6.1b.

6.4.4 Vertex Indices

Most real-world surfaces are, to some degree, closed and smooth. In representing these surfaces, we do not want to have empty space between neighboring triangles. The best way of ensuring this is to use vertices that have equal positions in neighboring triangles. Figure 6.2a depicts an example of a fan of six triangles (defining a hexagon) that meet in a single point.

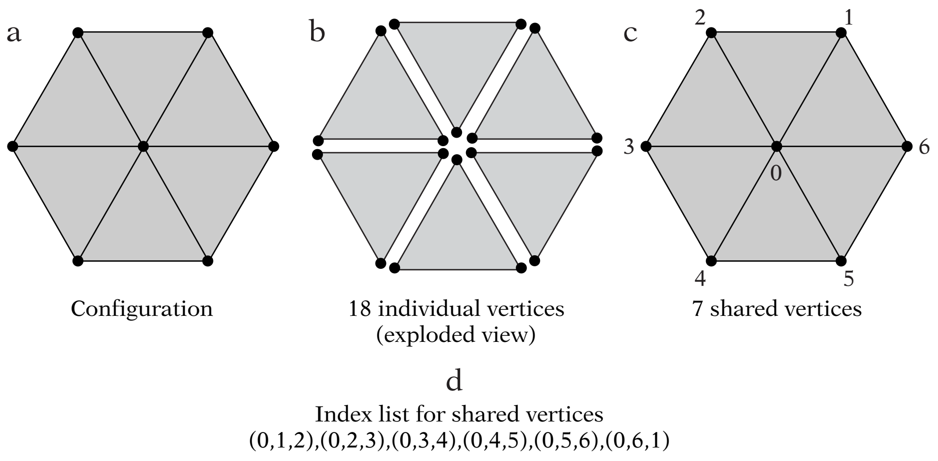


Figure 6.2 A hexagonal configuration of triangles.

With six triangles, we require 18 vertices — three for each of the six triangles, as shown in Figure 6.2b. However, only seven of these vertices have unique positions. In fact, on a closed surface, most triangles will share the positions of several (or all) of their vertices with multiple other triangles in that object. Rather than blindly generating $3T$ vertices for any set of T triangles, many graphics systems (including OpenGL) allow the idea of external triangle index information, also known as *indexed geometry*.

Indexed geometry defines an object with two arrays, one for the vertices and one for the triangle indices. The array containing the vertices contains only the N *unique* vertices. In our hexagon example, this would be an array of seven vertices, six around the edge and one in the center. Figure 6.2c shows these seven vertices, numbered with their indices in the vertex array. This array does not define any information about the triangles in the object.

The second array is an array of $3T$ indices. Each set of three indices represents a triangle. The indices are used to look up vertices in the vertex array; the three vertices are joined into a triangle. Figure 6.2d shows the index list for the hexagon example.

Note that index arrays are arrays of unsigned integers (either 16 or 32 bits) and thus generally require *far* less memory than an array of vertices with the same number of elements (since a vertex generally consists of at least three floating-point values). There is overhead for the vertex array, but for most surfaces (where the average vertex appears in several triangles), the memory savings (and in some 3D hardware systems, the overall performance gains) can be very significant.

For example, we can compute the memory savings of indexed geometry for our hexagon, assuming that vertices are as lightweight as possible (this will actually skew the results in favor of nonindexed geometry). In the nonindexed case, there are six triangles, giving a memory usage of

$$Nonindexed = triangles \times \frac{vertices}{triangle} \times \frac{floats}{vertex} \times \frac{bytes}{float}$$

$$= 6 \times 3 \times 3 \times 4 = 216 \text{ bytes}$$

Assuming 16-bit `unsigned short` indices for the index list, the indexed case has the following combined memory usage for its two arrays:

$$Indexed = triangles \times \frac{indices}{triangle} \times \frac{bytes}{index} + vertices \times \frac{floats}{vertex} \times \frac{bytes}{float}$$

$$= 6 \times 3 \times 2 + 7 \times 3 \times 4 = 36 + 84 = 120 \text{ bytes}$$

A significant savings, even in this simple case. If the vertices had included normals, the difference between the two memory requirements would have been even larger.

6.4.5 OpenGL Vertex Indices

Source Code Demo IndexedGeom

In OpenGL indexed geometry can be implemented in one of several ways, the most widely available being vertex arrays, which became a part of the standard in OpenGL 1.1. To enable vertex arrays, an application must enable the handling of arrays for each vertex component it would like to specify via an array. For example, to enable vertex arrays for positions, the function is

```
glEnableClientState(GL_VERTEX_ARRAY)
```

To turn off the handling of array-based vertices (and switch back to non-indexed mode), the call is

```
glDisableClientState(GL_VERTEX_ARRAY)
```

Having enabled the handling of vertex arrays, the entire array of vertices can be passed to OpenGL in a single call. For example, imagine creating the vertices for a tetrahedron:

```
glEnableClientState(GL_VERTEX_ARRAY)

static GLfloat verts[4 * 3] = { 0.0f, 0.0f, 0.0f,
                               1.0f, 0.0f, 0.0f,
                               0.0f, 1.0f, 0.0f,
                               0.0f, 0.0f, 1.0f };

glVertexPointer(3, GL_FLOAT, 0, verts);
```

The function `glVertexPointer` specifies the array of vertices:

1. The first argument specifies the number of components per vertex position (in this case, an array of 3D vertex positions).
2. The second argument specifies the format of each vertex component.
3. The third argument specifies any additional spacing, or “padding” (in bytes) between each vertex.
4. The final argument is the pointer to the vertices themselves.

This function causes OpenGL to store the pointer and does not copy the data (in fact, this would not even be possible, since the function does not specify the number of vertices in the array!). As such, the storage for the array that is passed in by the application must be valid for the entire time that it is to be used to render.

The code for our tetrahedron contains no information about indices. Without any triangle indices, nothing will be drawn. So, we must create the index array. The array that follows defines 12 indices, three for each of the four triangles in the tetrahedron. Then, it calls `glDrawElements`, which can draw an entire array of primitives in a single call.

1. The first argument defines the type of primitive (`GL_TRIANGLES` causes each subsequent triple of indices to form a triangle).
2. The second argument supplies the number of indices (in the case of `GL_TRIANGLES`, this number should be three times the number of triangles).
3. The third argument defines the type of these elements in the index array.
4. The final parameter is the address of the base of the array:

```
GLushort indices[12] = { 0, 1, 3, // Y=0 plane triangle
                         0, 3, 2, // X=0 plane triangle
                         0, 2, 1, // Z=0 plane triangle
                         1, 2, 3  // Diagonal plane triangle
                       };

glDrawElements(GL_TRIANGLES, 3*4, GL_USHORT, indices);
```

While the preceding code does not appear to be much shorter than the original vertex-at-a-time version, the vertex array version requires fewer OpenGL function calls and can often be rendered at much higher speed than the individual vertex method (as the vertices and indices for all triangles are specified at once). Furthermore, it is possible that the indexed version will have to transform only four vertices, while the individual method will have to transform all 12 individually.

OpenGL (as well as most other rendering APIs) supports a wide range of indexed geometry. Indexed triangle lists, such as the ones we've introduced, are simple to understand but are not as optimal as other representations. The most popular of these more optimal representations are *triangle strips*, or *tristrips*. In a triangle strip, the first three vertex indices represent a triangle, just as they do in a triangle list. However, in a triangle strip, each additional vertex (the fourth, fifth, etc.) generates another triangle — each index generates a triangle out of itself and the two indices that preceded it (e.g., 0-1-2, 1-2-3, 2-3-4 . . .). This forms a ladderlike strip of triangles (note that each triangle is assumed to have the reverse orientation of the previous triangle; counterclockwise, then clockwise, then counterclockwise again, etc.). Then, too, whereas triangle lists require $3T$ indices to generate T triangles, triangle strips require only $T + 2$ indices to generate T triangles. Much research has gone into generating optimal strips by maximizing the number of triangles while minimizing the number of strips, since there is a two vertex "overhead" to generate the first triangle in a strip. The longer the strip, the lower the average number of indices required per strip. Most consumer 3D hardware that is available today renders triangle strips at peak performance. OpenGL renders triangle strips using an argument of `GL_TRIANGLE_STRIP` as the primitive type (replacing `GL_TRIANGLES`).

6.5 Coloring a Surface

The following sections describe a wide range of methods to assign colors to surface geometry. From the simplest methods (such as assigning a single, fixed color per object) all the way to the most expensive (such as effects requiring the application of multiple image-based "textures"), each has its benefits, limitations, costs, and mathematical issues.

The basic goal of each method is the same; given an object O, a triangle $T \in O$, made up of vertices $V1$, $V2$, and $V3$, along with barycentric coordinates (s, t) defining a unique point in T, return a color that is associated with this point on the geometric object. Many of the methods we will describe will require additional data both within the triangle and its vertices and within the scene as a whole. However, in each case, the coloring function $Color(O, T, (s, t))$ takes the information that describes the point (or "sample") and returns an RGB color.

The first sections will deal with constant colors assigned prior to rendering, which are generally the simplest methods. Later sections will progress to the more dynamic, per sample, per frame methods such as dynamic lighting.

6.6 Using Constant Colors

The method of coloring geometry that produces the highest runtime performance is to assign colors to geometry prior to rendering, either by having an artist assign colors to every surface during content creation time, or else to use an off-line process to generate static colors for all geometry. With these static colors assigned, there is relatively little that must be done to select the correct color for a given sample.

Put simply, constant colors mean that given O, T, and (s, t), $Color(O, T, (s, t))$ will never change. No environmental information like dynamic lighting will be factored into the final color. The function $Color$ is the *shading* function for the geometry. In 3D rendering, a *shading function* (or *shading method*, or *shader*) is simply a method that assigns colors to every point on the geometry. It should not be confused with *lighting* (to be described in great detail later), which is one way of generating source colors used in the shading process.

6.6.1 Per-Object Colors

The simplest form of useful coloring is to assign a single color per object. The coloring function is thus

$$Color(O, T, (s, t)) = C_O$$

The color value C_O is simply added to the data structure describing O. Note that this function does not depend on T or (s, t). If we are taking multiple samples using this function, we only need to look up C_O again only if we sample a different object. Constant coloring of an entire object is of very limited use, since the entire object will appear to be flat, with no color variation. The viewer will be able to determine only where the object "is" and "is not." At best, only the outline of the object will be visible against the backdrop. As a result, except in some special cases, per-object color is rarely used as the final shading function for an object.

6.6.2 Per-Triangle Colors

A similar, but finer-grained and more powerful method for assigning colors to geometry is to assign a color to each triangle. This is known as *faceted*, or *flat* shading, because the resulting geometry appears planar on a per-triangle basis. The function used to assign colors is very similar to the per-object function:

$$Color(O, T, (s, t)) = C_F$$

Source Code
Demo
Shading

Normally, this requires adding a color field (C_F) to each triangle. However, OpenGL does not specifically support a separate per-triangle color value. In explicit vertex-by-vertex mode, a single-color triangle may be specified by setting the current color once and then "pushing" three vertex positions to render a triangle with no intervening colors. All three vertices will be assigned the same color:

```
// Flat-shaded blue triangle
glColor3f(0.0, 0.0f, 1.0f);
glVertex3f(1.0f, 0.0f, 1.0f);
glVertex3f(1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
```

However, flat shading can also be enabled globally at the OpenGL level, in which case the color of one triangle vertex (the final vertex) will be used for the entire triangle, even if the three vertex colors differ. Flat shading is enabled in OpenGL with the function call

```
glShadeModel(GL_FLAT);
```

and disabled (switching to smooth shading) via the function call

```
glShadeModel(GL_SMOOTH);
```

With OpenGL vertex arrays, per-triangle colors are specified indirectly—the color of one of the triangle's vertices is used as the color of the entire triangle. The OpenGL specification details which vertex is used in each mode, but for `GL_TRIANGLES` the vertex used is the last (third) vertex in the triangle. Since OpenGL does not have a notion of a polygon color (only vertex colors), the face color must be associated with the final vertex that is used to generate the triangle. This can be problematic in the case of indexed geometry, where some vertices may have to be used as the third vertex for more than one triangle (it is common and very easy to generate indexed geometry that has more triangles than vertices). In such cases, it may be necessary to duplicate vertices in order to be able to specify triangle-specific colors.

6.6.3 Per-Vertex Colors

Many of the surfaces approximated by tessellated objects are smooth, meaning that the goal of coloring these surfaces is to emphasize the smoothness of the original surface, not the artifacts of its approximation with flat triangles. This fact makes flat shading a very poor choice for many tessellated objects. A shading method that can generate the appearance of a smooth surface is needed. Per-vertex coloring, along with a method called *Gouraud shading* (after its inventor, Henri Gouraud) does this. Gouraud shading is based on the existence of some form of per-vertex colors, assigning a color to any point on a triangle by linearly interpolating the three vertex colors over the surface of the triangle. As with the other shading methods we have discussed, Gouraud shading is independent of the source of these per-vertex colors; the vertex colors may be assigned explicitly by the application, or generated on the fly via per-vertex lighting and so on. This linear interpolation is both simple and smooth and can be expressed as a mapping of barycentric coordinates (s, t) as follows:

$$Color(O, T, (s, t)) = sC_{V1} + tC_{V2} + (1 - s - t)C_{V3}$$

Examining the terms of the equation, it can be seen that Gouraud shading is simply an affine transformation from barycentric coordinates (as homogeneous points) in the triangle to RGB color space. The mapping may be written as the 3×3 matrix transform

$$Color(O, T, (s, t)) = \begin{bmatrix} (C_{V1} - C_{V3})_R & (C_{V2} - C_{V3})_R & (C_{V3})_R \\ (C_{V1} - C_{V3})_G & (C_{V2} - C_{V3})_G & (C_{V3})_G \\ (C_{V1} - C_{V3})_B & (C_{V2} - C_{V3})_B & (C_{V3})_B \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

or simply

$$Color(O, T, (s, t)) = \begin{bmatrix} (C_{V1} - C_{V3}) & (C_{V2} - C_{V3}) & C_{V3} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

An important feature of per-vertex smooth colors is that color discontinuities can be avoided at triangle edges. This was a major drawback of per-triangle colors, as any triangles that shared an edge would either have to be the same color or else have a sharp color discontinuity at the shared edge. This can be avoided with per-vertex colors.

Internal to each triangle, the colors are interpolated smoothly, as can be seen from the fact that Gouraud shading interpolation is an affine mapping from barycentric coordinates to RGB color space. At triangle edges, color discontinuities can be avoided by ensuring that the two vertices defining a shared edge in one triangle have the same color as the matching pair of vertices in the other triangle. At a shared edge between two triangles, the color of the third vertex in each triangle (the vertices that are not an endpoint of the shared edge) does not factor into the color along that shared edge. This is an added degree of freedom over per-triangle colors. This can be shown as follows. Assume we have a triangle with vertex colors C_{V1}, C_{V2}, and C_{V3}.

By our definition of barycentric coordinates, the barycentric coordinate of $V1$ is (1,0), and the barycentric coordinate of $V3$ is (0,0). Thus, in barycentric coordinates, the edge between $V1$ and $V3$ is defined by

$$(s, t) = (1 - r, 0)$$

where $0 \leq r \leq 1$. Thus, the colors across the edge are

$$\begin{aligned} Color &= sC_{V1} + tC_{V2} + (1 - s - t)C_{V3} \\ &= (1 - r)C_{V1} + (0)C_{V2} + (1 - (1 - r) - 0)C_{V3} \\ &= (1 - r)C_{V1} + (r)C_{V3} \end{aligned}$$

which does not involve C_{V2}. Similar derivations show that analogous cases are true for any triangle edge. As a result, there will be no color discontinuities across triangle boundaries, as long as the shared vertices between any pair of triangles are the same in both triangles. In fact, with fully shared, indexed geometry, this happens automatically (since co-located vertices are shared via indexing). Figure 6.3 allows a comparison of geometry drawn with per-face colors and with per-vertex colors.

The linear interpolation used for Gouraud shading is completely defined by the three vertices of a triangle. Gouraud shading across a general quadrilateral is dependent on how that quadrilateral is decomposed into triangles. In Figure 6.4, we see a quadrilateral with its assigned vertex colors. The figure shows that simply by changing the way the quadrilateral is broken into triangles, the Gouraud shading can change significantly. Note that the two cases

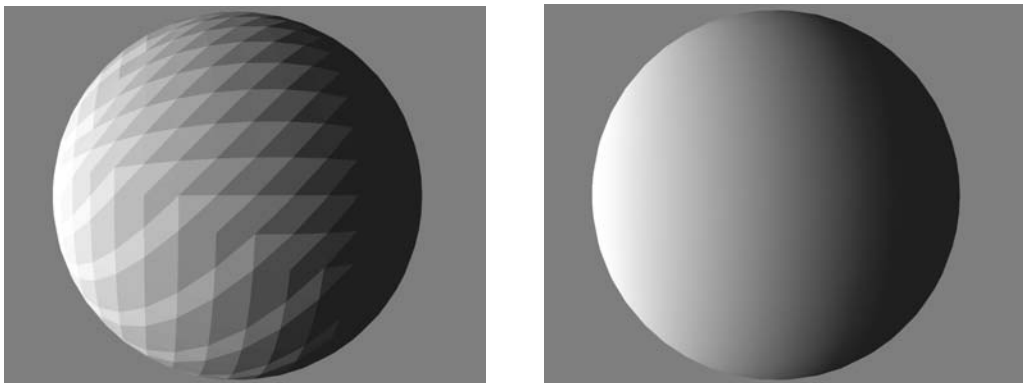


FIGURE 6.3 Flat (per-face) and Gouraud (per-vertex) shading.

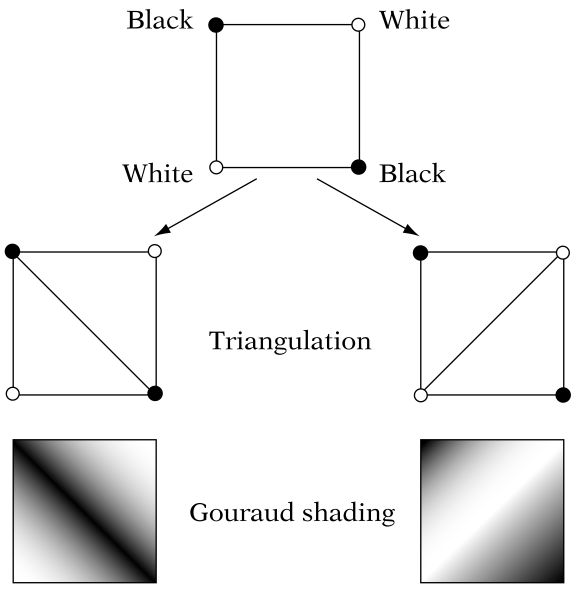


FIGURE 6.4 Gouraud shading in a quadrilateral.

use the same vertex colors and vertex positions but are simply triangulated differently.

While the colors need not have any triangle boundary discontinuities, there are often discontinuities in the *derivative* of the color at an edge. In more visual terms, the *slope* of the color (how rapidly it is changing across the face of a triangle) is defined by all three triangle vertices. As a result, even if the colors match on a triangle edge, there is often a sharp change in the way colors are interpolated *across* that edge. Even though the shared vertices have the same color, the fact that the derivative of color changes sign across the boundary (the direction of color change reverses) makes the edge visible. If measured as a change in derivative, this appears subtle, but the human visual system actually enhances the discontinuity in an effect called *mach banding*. Mach banding is a physiological trait of the human visual system that causes these color gradients to appear even sharper than they are, meaning that even Gouraud shading cannot completely hide artifacts of tessellation. For a far more detailed discussion of the physiological perception of color, see [20].

As mentioned in passing earlier, Gouraud shading is enabled in OpenGL via the function call

```
glShadeModel(GL_SMOOTH);
```

SOURCE CODE DEMO BasicShading

For far more details on the rendering of flat versus smooth (or Gouraud) shaded triangles, see Chapter 8. Both flat and Gouraud shading are used to interpolate colors generated by dynamic lighting. For a detailed discussion of dynamic lighting, see Chapter 7.

Sharp Edges

Not all tessellations represent completely smooth objects. In some cases, sharp geometric edges in the tessellation really do represent the original surface accurately. In addition, the edge between two triangles may mark the boundary between two different colors on the surface of the object. In these situations, interpolating smoothly across triangle boundaries is not the desired behavior. The vertices along an edge need to have different colors in the two triangles. In general, when Gouraud shading is used, these situations require coincident vertices to be duplicated, so that the two coincident copies of the vertex can have different colors. Figure 6.5 provides an example of a cube drawn with entirely shared vertices and with duplicated vertices to allow per-vertex, per-face colors. Note that the cube is not flat-shaded in either case — there are still color gradients across each face. The example with duplicated vertices and sharp shading edges looks more like a cube.

In this context, a “sharp” edge is not necessarily a geometric property. It is nothing more than an edge that is shared by two adjacent triangles where the triangle colors on either side of the edge are different. This produces a visible, sharp line between the two triangles where the color changes.

Sharp edges in OpenGL are a nonissue if you are using vertex-at-a-time triangle specification. In this case, each vertex of each triangle is already being specified independently, making it easy to specify different vertex colors for

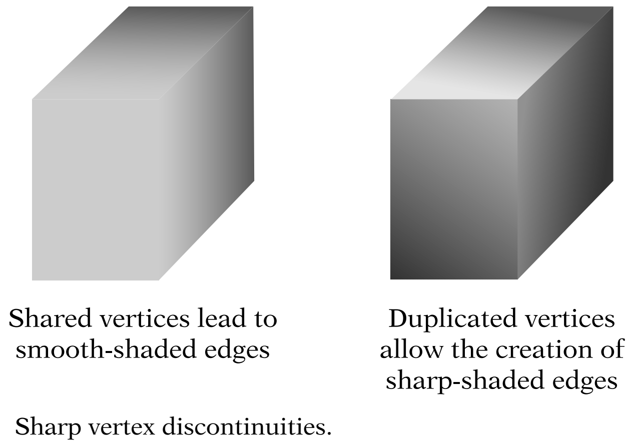

Shared vertices lead to smooth-shaded edges
Duplicated vertices allow the creation of sharp-shaded edges

FIGURE 6.5 Sharp vertex discontinuities.

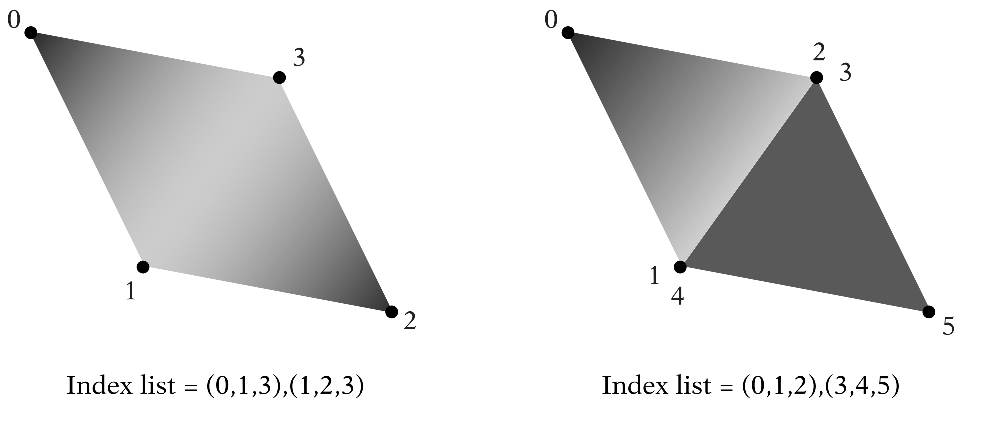

0
3
1
2
Index list = (0,1,3),(1,2,3)
Triangles share adjacent vertices
0
2
3
1
4
5
Index list = (0,1,2),(3,4,5)
Triangles do not share adjacent vertices

Figure 6.6 Duplicating indexed vertices for sharp color edges.

multiple co-located vertices. In the case of vertex arrays, however, duplicating vertices may be required, so that co-located vertices in different triangles can have different colors. The issue arises because the function `glDrawElements` uses the same index to look up a vertex's color as it does the vertex's position. As a result, the color of a vertex and its position are directly linked. When using vertex arrays, the vertices defining any sharp, shared edge must be duplicated. The more sharp edges there are in a vertex array primitive, the less vertex sharing is possible (i.e., more duplicated vertices), decreasing the efficiency of the method. Figure 6.6 provides a visual representation of a pair of triangles with and without a sharp color edge.

6.6.4 Limitations of Basic Shading Methods

Real-world surfaces often have detail at many scales. The shading/coloring methods described so far require that colors be assigned only at tessellation-level features, either per-triangle or per-vertex. While this works well for surfaces whose colors change at geometric boundaries, many surfaces do not fit this restriction very well, making flat shading and Gouraud shading inefficient at best.

For example, imagine a flat sheet of paper with text written upon it. The flat, rectangular sheet of paper itself can be represented by as few as two triangles. However, in order to use Gouraud shading to represent the text, the piece of paper would have to be subdivided into triangles at the edges of every character written upon it. None of these boundaries represents geometric features, but rather are needed only to allow the color to change from white (the paper's color) to black (the color of the ink). Each character could easily require hundreds of vertices to represent the fine stroke details. This could lead to a simple, flat piece of paper requiring tens of thousands of vertices. Clearly, we require a shading method that is capable of representing detail at a finer scale than the level of tessellation.

6.7 Texture Mapping

6.7.1 Introduction

One method of adding detail to a rendered image without increasing geometric complexity is called *texture mapping*, or more specifically *image-based texture mapping*. The physical analogy for texture mapping is to imagine wrapping a flat, paper photograph onto the surface of a geometric object. While the overall shape of the object remains unchanged, the overall surface detail is increased greatly by the image that has been wrapped around it. From some distance away, it can be difficult to even distinguish what pieces of visual detail are the shape of the object and which are simply features of the image applied to the surface.

A real-world physical analogy to this is theatrical set construction. Often, details in the set will be painted on planar pieces of canvas, stretched over a wooden frame (i.e.,"flats"), rather than built out of actual, three-dimensional wood, brick, or the like. With the right lighting and positioning, these quickly painted flats can appear as very convincing replicas of their real, 3D counterparts. This is the exact idea behind texturing—using a 2D, detailed image placed upon a simple 3D geometry to create the illusion of a complex, detailed, fully 3D object.

An example of a good use of texturing is a rendering of a stucco wall; such a wall appears flat from any significant distance, but a closer look shows that it consists of many small bumps and sharp cracks. While each of these bumps could be modeled with geometry, this is likely to be expensive and unlikely to be necessary when the object is viewed from a distance. In a 3D computer graphics scene, such a stucco wall will most frequently be represented by a flat plane of triangles, covered with a detailed image of the bumpy features of lit stucco.

The fact that texture mapping can reduce the problem of generating and rendering complex 3D objects into the problem of generating and rendering simpler 3D objects covered with 2D paintings or photographs has made texture mapping very popular in real-time 3D. This, in turn, has led to the method being implemented in display hardware, making the method even less expensive computationally. The following sections will introduce and detail some of the concepts behind texture mapping, some mathematical bases underlying them, and basics of how texture mapping can be used in OpenGL applications.

6.7.2 Shading via Image Lookup

The real power of texturing lies in the fact that it uses a set of samples (an image) as its means of generating color. In a sense, texturing is simply a system of indirect coloring. Rather than directly interpolating colors that are stored in the vertices, the vertex values serve only to describe how an image is mapped to the triangle. By adding a level of indirection between the per-vertex values and the final colors, texturing can create the appearance of a very complex shading function that is actually no more than a lookup into a table of samples.

The process of texturing involves defining three basic mappings:

1. To map all points on a surface (smoothly in most neighborhoods) into a 2-dimensional (or in some cases, 1D or 3D) domain
2. To map points in this (possibly unbounded) domain into a unit square (or unit interval, cube, etc.)
3. To map points in this unit square to color values

The first stage will be done using a modification of the method we used for colors with Gouraud shading, an affine mapping. The second stage will involve methods such as min, max, and modulus. The final stage is the most unique to texturing and involves mapping points in the unit square into an image. We will begin our discussion with a definition of texture images.

6.7.3 Texture Images

The most common form of texture images (or *textures*, as they are generally known) are 2-dimensional, rectangular arrays of color values. Every texture has a width (the number of color samples in the horizontal direction) and a height (the number of samples in the vertical direction). Textures are similar to almost any other digital image, including the screen, which is also a 2D array of colors. Just as the screen has pixels (for picture elements), textures have *texels* (texture elements). While some graphics systems allow 1-dimensional textures (linear arrays of texels) and even 3-dimensional textures (cubes or rectangular parallelopipeds of texels), by far the most common and most useful are 2-dimensional, image-based textures. Our discussion of texturing will focus entirely on 2-dimensional textures.

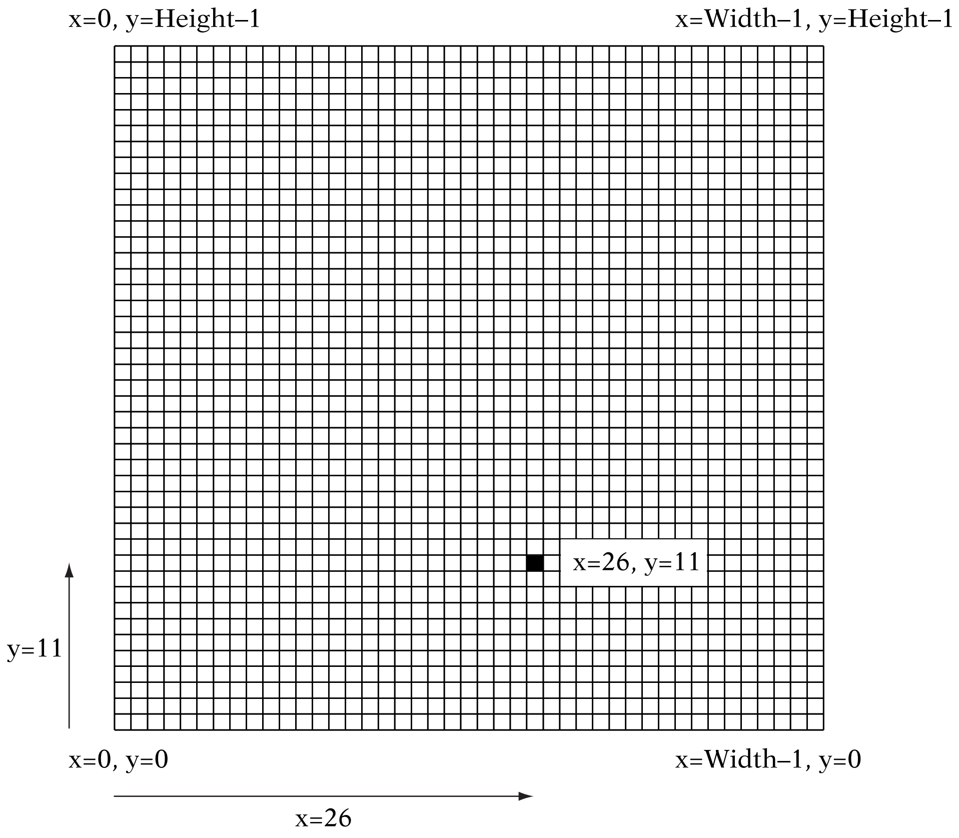


FIGURE 6.7 Texel-space coordinates in an image.

We can refer to the position of a given texel via a 2D value (x, y), in texel units — note that these coordinates are (column, row), the reverse of how we generally refer to matrix elements. Figure 6.7 shows an example of a common mapping of texel coordinates into a texture. Note that while the left to right increasing mapping of x is universal in graphics systems, the bottom to top increasing mapping of y is not (bottom to top *is* used in OpenGL).

SOURCE CODE DEMO BasicTexturing

Two-dimensional texturing is enabled in OpenGL at the highest level with the call

```
glEnable(GL_TEXTURE_2D);
```

and disabled with the function call

```
glDisable(GL_TEXTURE_2D);
```

A 2-dimensional texture image is specified in OpenGL via the function

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height,
   0, GL_RGBA, GL_UNSIGNED_BYTE, texels);
```

We will avoid explaining all of the possible values for the arguments (see the OpenGL Programming Guide [83] for details) because most are not relevant to our discussion and can be left as is in other cases. We will confine our discussion to the following:

- The first parameter, `GL_TEXTURE_2D`, specifies that the 2-dimensional texturing settings (the only form we will discuss in detail) are to be changed by this call.
- Parameter three, `GL_RGBA`, defines the requested "internal" format. In this case we are requesting only that the system store the given texture as full-color image with alpha per texel. This parameter does *not* define anything about the data we are passing in, only how we would like it to be stored in the system.
- The next two parameters, `width` and `height` (integers), specify the width and height of the texture in texels. OpenGL requires that textures have power-of-two dimensions (i.e., `width` $= 2^m$ and `height` $= 2^n$, where m and n are integers).
- The seventh parameter, `GL_RGBA`, specifies that the texel data we are sending defines each texel as a red, green, blue, and alpha value in sequence.
- The eighth parameter, `GL_UNSIGNED_BYTE`, defines that each of the components of each texel is stored as an unsigned 8-bit byte. Together, parameters seven and eight define that the texel data we are submitting has 32-bit texels, stored as RGBA quads, each component of which is between 0 and 255.
- The final parameter is a pointer to `width` $\times$ `height` texels of the given format, stored in row-major, left-to-right, bottom-to-top format. In the previous case, the pointer will point to a block of `width` $\times$ `height` $\times$ 4 bytes of texture data.

Efficient Texture Images in OpenGL

SOURCE CODE DEMO BasicTexturing

Note that `glTexImage2D` specifies the data for only the "current texture," the one active for the next set of rendered geometry. If a texture will be used many times (perhaps once or more per frame), this interface is a slow and cumbersome way to have to specify textures each time they are used. Instead, OpenGL allows applications to "bind" a texture to a positive integer "name" or "identifier." This is done in two steps. First, one or more free texture identifiers are generated via a call to `glGenTextures`, as in the following example, which generates identifiers for four subsequent textures:

```
GLuint textures[4];
glGenTextures(4, textures);
```

Upon return, the array will contain four nonzero texture names that can be bound to textures. Textures are both bound to names and accessed from names by the same call. A call to

```
glBindTexture(GL_TEXTURE_2D, textures[0]);
```

will bind the texture name passed as the second parameter. The exact behavior of `glBindTexture` is dependent upon whether or not the given identifier has already been "bound." On the first call to `glBindTexture` with a given nonzero identifier, this function will link the given identifier to the current 2D texture that was set with `glTexImage2D`. Subsequent calls with the same identifier will access the texture that was linked to the identifier and replace the current texture image. Once all textures are bound to different identifiers, calls to `glBindTexture` are all that are needed to quickly switch between all textures. In addition, the unsigned integer values are all that the application must store to reference their textures. When a texture is no longer needed, it should be deleted and its identifier freed for later use with `glDeleteTextures`, which takes the same arguments as `glGenTextures` as in the following:

```
GLuint textures[4];
// ...
glDeleteTextures(4, textures);
```

While convenience of texture specification is a useful benefit, there is a much more important reason for using texture binding in OpenGL, owing to the design realities of 3D rendering hardware. Image data that is to be used as a texture must be stored in special memory that is a part of the graphics subsystem. Texture images created in main system memory must first be formatted for consumption by the texturing hardware (sometimes requiring the bitwise format of the pixels to be converted to something supported by the hardware). Then, the texture must be transferred to the texture memory of the 3D hardware. Both of these steps are time-consuming, and if they must be redone each time the texture is used (often more than once per frame), the result can be greatly reduced performance of the application. Binding the texture in OpenGL allows the OpenGL implementation to take these steps once, during the first call to `glBindTexture` for each texture. Subsequent calls to bind a defined texture will simply require the OpenGL implementation to set the hardware to use the existing version of the texture in the device's texture memory. This is a much faster operation than converting and reloading a texture into texture memory. Note that if the contents of the texture must be changed (e.g., changing the color of one or more texels) once it is bound, the texture must be processed and transferred to texture memory again. In fact, OpenGL must be told *explicitly* to reload these changes — changing the contents of the source array of pixels that was passed into the original call to `glTexImage2D` will have no effect on the copy of the texture that is in texture memory. As a result, it is best to avoid changing the pixel colors of textures once they are bound or else decreased performance can result.

Sometimes it may not be possible for the OpenGL hardware to fit all currently bound textures into the device's texture memory at once. In such cases, the OpenGL implementation must move textures in and out of texture memory as they are needed. A texture that is currently stored in the device's texture memory is referred to as *resident*, while a texture that is not currently in texture memory is *nonresident*. If an OpenGL implementation must move textures about every frame, performance of an application will be degraded. It is important to delete bound textures that are no longer required, because this frees texture memory for actively used textures. OpenGL includes numerous functions that can be used by advanced programmers to fine-tune the use of bound textures, including functions for texture prioritization. See [83] for details on texture memory management.

6.8 Texture Coordinates

While textures can be indexed by 2D vectors of nonnegative integers on a per-texel basis (texel coordinates), textures are normally addressed in a more general, texel-independent manner. The texels in a texture are most often addressed via height- and width-independent "U" and "V" values. These 2D real-valued coordinates are mapped in the same way as texel coordinates,

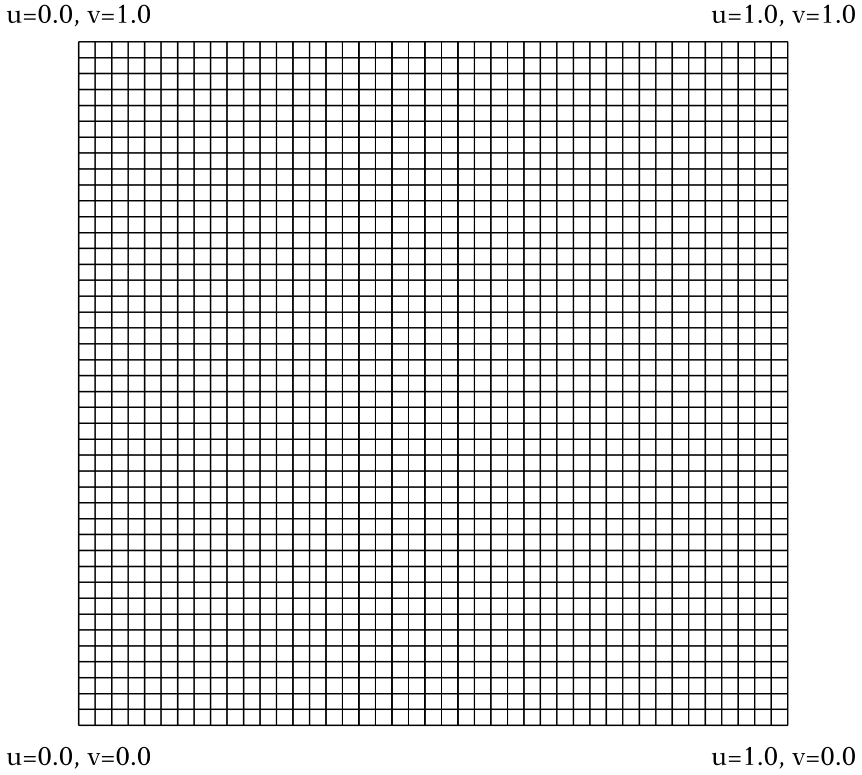


FIGURE 6.8 Mapping UV coordinates into an image.

except for the fact that U and V are multiplied by the width and height of the texture, respectively. Figure 6.8 depicts the common mapping of UV coordinates into a texture. These normalized UV coordinates have the advantage that they are completely independent of the height and width of the texture. Almost all texturing systems use these normalized UV coordinates, and as a result, they are often referred to as *texture coordinates*, or *texture* UVs.

The real-valued texture coordinates would *seem* to add a continuity that does not actually exist across the domain of an image, which is a discrete set of color values. For example, in C or C++ one does not access an array with a `float`—the index must first be rounded to an integer value. For the purposes of the initial discussion of texturing, we will leave the details of how real-valued texture coordinates map to texture colors somewhat vague. This is actually a rather broad topic and will be discussed in detail in Chapter 8. Initially, it is easiest to think of the texture coordinate as referring to the color of the closest texel. For example, given our assumption, a texture coordinate of (0.5, 0.5) in a texture with width and height equal to 128 texels would map to texel (64, 64). This is referred to as *nearest-neighbor* texture mapping. While this is the simplest method of mapping real-valued texture coordinates into a texture, it is not necessarily the most commonly used in modern applications. We shall discuss more powerful and complex techniques in Chapter 8, but nearest-neighbor mapping is sufficient for the purposes of the initial discussion of texturing.

Source Code Demo BasicTexturing

In vertex-by-vertex mode, per-vertex texture coordinates may be assigned to a vertex in OpenGL by setting the current texture coordinate value prior to creating a vertex. Recall that vertices are actually created only when `glVertex*` is called to specify the vertex position. The *u* and *v* coordinates (*s* and *t* in OpenGL) can be specified with

```
float u,v;
// ...
glTexCoord2f(u, v);

// OR

float uv[2];
// ...
glTexCoord2fv(uv);
```

When using vertex arrays and shared geometry, texture coordinates are enabled using

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

and the texture coordinate array itself is passed in using

```
static float uvs[2 * kNumVerts];
// ...
glTexCoordPointer(2, GL_FLOAT, 0, uvs);
```

where the arguments to `glTexCoordPointer` are equivalent to those of `glVertexPointer`.

6.8.1 Mapping Texture Coordinates

The texture coordinates defined at the three vertices of a triangle define an affine mapping from barycentric coordinates to UV space. Given the barycentric coordinates of a point in a triangle, the texture coordinates may be computed as follows (Do not confuse the barycentric *s* and *t* with the OpenGL *s* and *t*; they are unrelated.):

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (u_{V1} - u_{V3}) & (u_{V2} - u_{V3}) & u_{V3} \\ (v_{V1} - v_{V3}) & (v_{V2} - v_{V3}) & v_{V3} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

Although there is a wide range of methods used to map textures onto triangles (i.e., to assign texture coordinates to the vertices), a common goal is to avoid "distorting" the texture. In order to discuss texture distortion, we need to define the U and V basis vectors in UV space. If we think of the U and V vectors as 2-vectors rather than the "point-like" texture coordinates themselves, then we compute the basis vectors as

$$\mathbf{e}_u = (1, 0) - (0, 0)$$
$$\mathbf{e}_v = (0, 1) - (0, 0)$$

The $\mathbf{e}_u$ vector defines the mapping of the horizontal dimension of the texture (and its length defines the size of the mapped texture in that dimension), while the $\mathbf{e}_v$ vector does the same for the vertical dimension of the texture.

If we want to avoid distorting a texture when mapping it to a surface, we must ensure that the affine mapping of a texture onto a triangle involves rigid transforms only. In other words, we must ensure that these texture-space basis vectors map to vectors in object-space that are perpendicular and of equal length. We define *ObjectSpace*() as the mapping of a vector in texture space to the surface of the geometry object. In order to avoid distorting the texture on the surface, *ObjectSpace*() should obey the following guidelines:

$$ObjectSpace(\mathbf{e}_u) \cdot ObjectSpace(\mathbf{e}_v) = 0$$
$$|ObjectSpace(\mathbf{e}_u)| = |ObjectSpace(\mathbf{e}_v)|$$

In terms of an affine transformation, the first constraint ensures that the texture is not sheared on the triangle (i.e., perpendicular lines in the texture image will map to perpendicular lines in the plane of the triangle), while the second constraint ensures that the texture is scaled in a uniform manner (i.e., squares in the texture will map to squares, not rectangles, in the plane of the triangle). Figure 6.9 shows examples of texture-to-triangle mappings that do not satisfy these constraints.

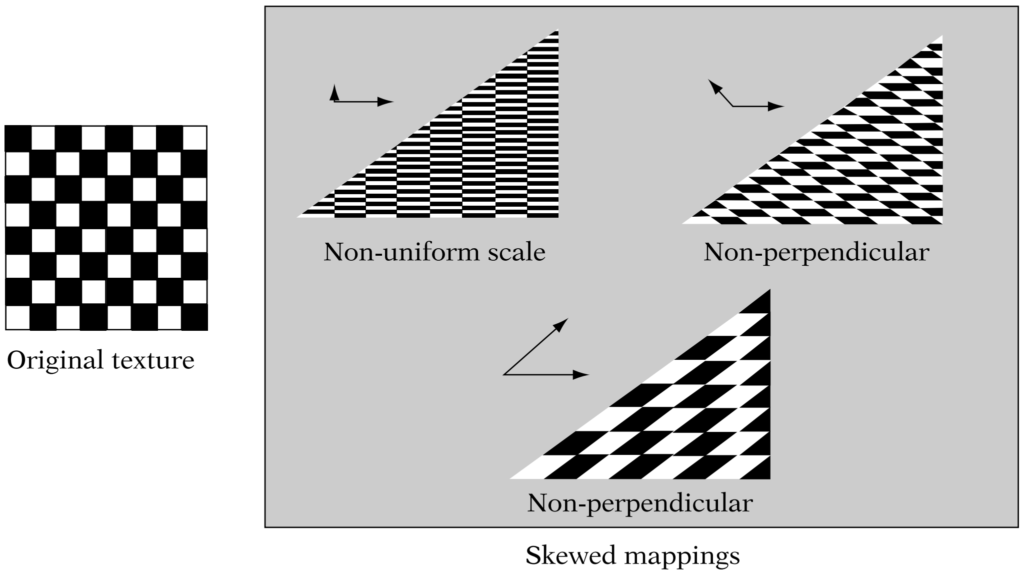


FIGURE 6.9 Examples of “skewed” texture coordinates.

Note that these constraints are by *no* means a requirement — many cases of texturing will stray from them, through either artistic desire or the simple mathematical inability to satisfy them in a given situation. However, the degree that these constraints do hold true for the texture coordinates on a triangle give some measure of how closely the texturing across the triangle will reflect the original planar form of the texture image.

6.8.2 Generating Texture Coordinates

Texture coordinates are often generated upon an object by some form of projection of the object-space vertex positions in $\mathbb{R}^3$ into the per-vertex texture coordinates in $\mathbb{R}^2$. All texture coordinate generation — in fact, all 2D texturing — is a type of projection. For example, imagine the cartographic problem of drawing a flat map of the earth. This problem is directly analogous to mapping a 2D texture onto a spherical object. The process cannot be done without distortion of the texture image. Any 2D texturing of a sphere is an exercise in matching a projection/“unwrapping” of the sphere onto a rectangular image (or several images) and the creation of 2D images that take this mapping into account. For example, a common, simple mapping of a texture onto a sphere is to use U and V as longitude and latitude in the texture image, respectively. This leads to discontinuities at the pole, where more and more texels are mapped over smaller and smaller surface areas as we approach the poles.

The artist must take this into account when creating the texture image. Except for purely planar mappings (such as the wall of a building), most texturing work done by an artist is an artistic cycle between generating texture coordinates upon the object and painting textures that are distorted correctly to map in the desired way to those coordinates.

6.8.3 Texture Coordinate Discontinuities

As was the case with per-vertex colors, there are situations that require shared, colocated vertices to be duplicated in order to allow the vertices to have different texture coordinates. These situations are less common than in the case of per-vertex colors, due to the indirection that texturing allows. Pieces of geometry with smoothly mapped texture coordinates can still allow color discontinuities on a per-sample level by painting the color discontinuities into the texture. Normally, the reason for duplicating colocated vertices in order to split the texture coordinates has to do with topology.

For example, imagine applying a texture as the label for a model of a tin can. For simplicity, we shall ignore the top and bottom of the can and simply wrap the texture as one would a physical label. The issue occurs at the

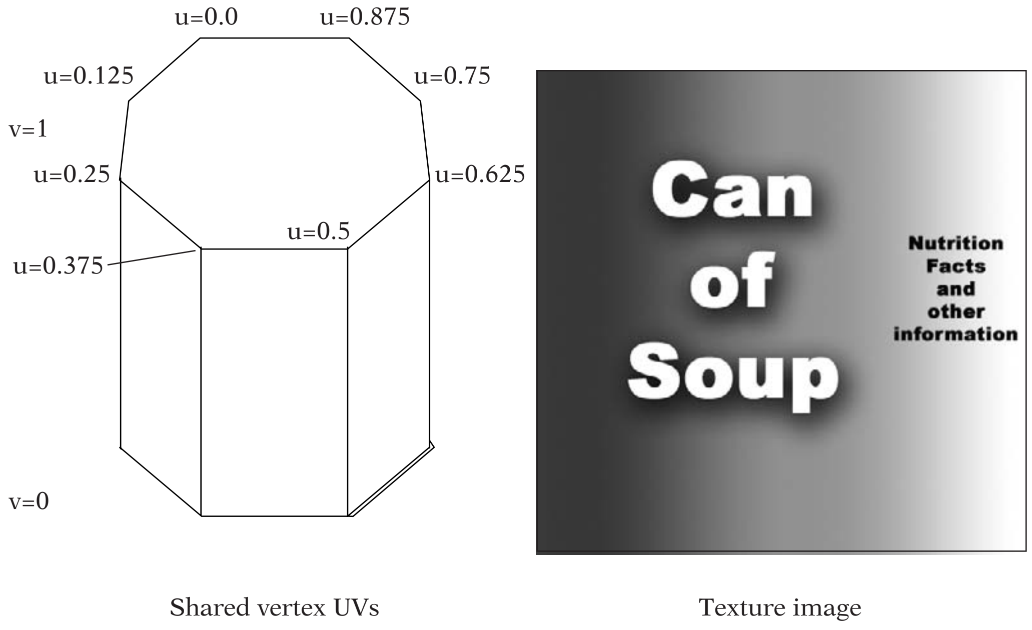


Figure 6.10 Texturing a can with completely shared vertices.

texture's seam. Figure 6.10 shows a tin can modeled as an 8-sided cylinder containing 16 shared vertices, 8 on the top and 8 on the bottom. The mapping in the vertical direction of the can (and the label) is simple, as shown in the figure. The bottom 8 vertices set $V = 0.0$ and the top 8 vertices set $V = 1.0$. So far, there is no problem. However, problems arise in the assignment of U. Figure 6.10 shows an obvious mapping of U to both the top and bottom vertices — U starts at 0.0 and increases linearly around the can until the eighth vertex, where it is 0.875, or $1.0 - 0.125$.

The problem is between the eighth vertex and the first vertex. The first vertex was originally assigned a U value of 0.0, but at the end of our circuit around the can, we would also like to assign it a texture coordinate of 1.0, which is not possible for a single vertex. If we leave the can as is, most of it will look perfectly correct, as we see in the front view of Figure 6.11. However, looking at the back view in Figure 6.11, we can see that the face between the eighth and first vertex will contain a squashed version of almost *the entire texture, in reverse*! Clearly, this in not what we want (unless we can always hide the seam). The answer is to duplicate the first vertex, assigning the copy associated with the first face $U = 0.0$ and the copy associated with the eighth face $U = 1.0$. This is shown in Figure 6.12 and looks correct from all angles.

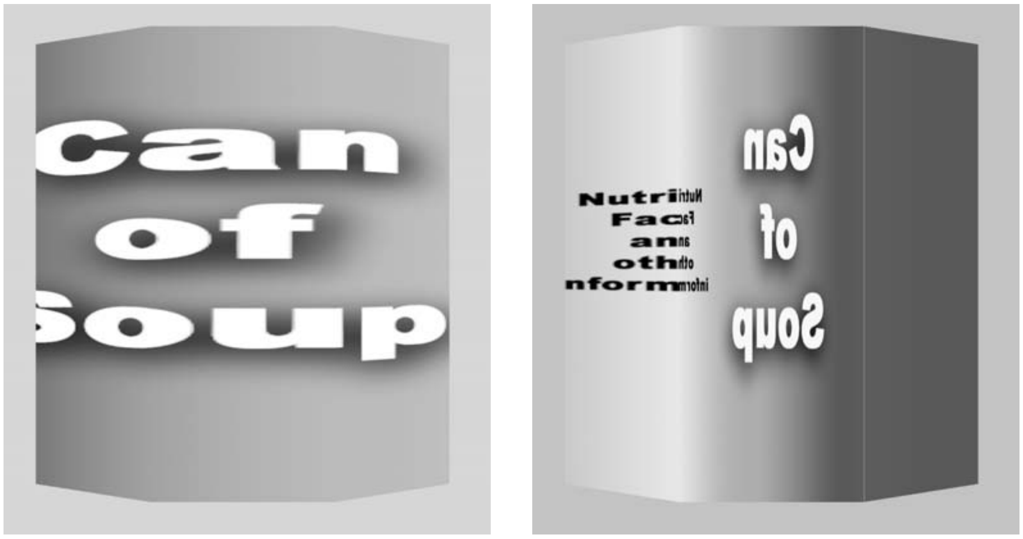

Can of Soup
Front side
(Appears to be correctly mapped)
Back side
(Incorrect, due to shared vertices along the label "seam")

FIGURE 6.11 Shared vertices can cause texture coordinate problems.

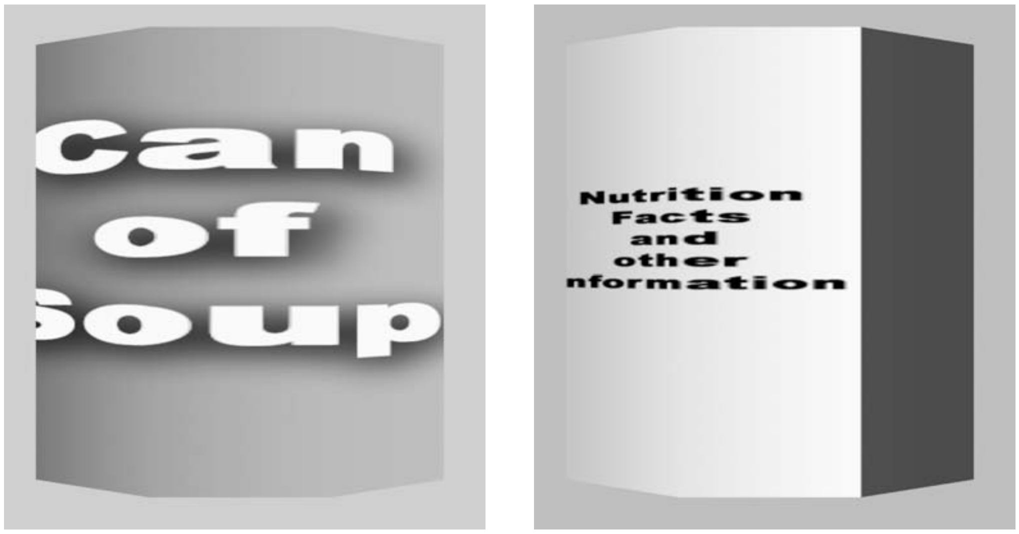


Figure 6.12 Duplicated vertices used to solve texturing issues.

6.8.4 Mapping Outside the Unit Square

So far, our discussion has been limited to texture coordinates within the unit square, $0.0 \le u, v \le 1.0$. However, there are interesting options available if we allow texture coordinates to fall outside of this range. In order for this to work, we need to define how texture coordinates map to texels in the texture when the coordinates are less than 0.0 or greater than 1.0. These operations are per-sample, not per-vertex, as we shall discuss.

The most common method of mapping unbounded texture coordinates into the texture is known as *texture wrapping*, *texture repeating*, or *texture tiling*. The wrapping of a component u of a texture coordinate is defined as

$$wrap(u) = u - \lfloor u \rfloor$$

The result of this mapping is that multiple "copies" of the texture "tile" the surface. Wrapping must be computed per-sample, not per-vertex. Figure 6.13 shows a square whose vertex texture coordinates are all outside of the unit square, with a texture applied via per-sample wrapping. Clearly, this is a very

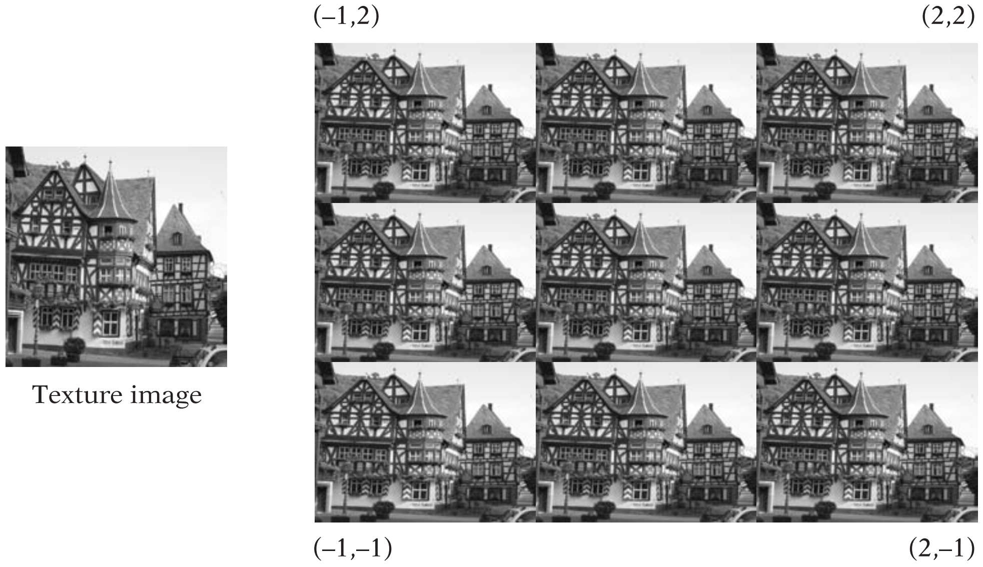


FIGURE 6.13 An example of texture wrapping.

different result than if we had simply applied the wrapping function to each of the vertices (which can be seen in Figure 6.14). In most cases, per-vertex wrapping produces incorrect results.

Wrapping is often used to create the effect of a tile floor, paneled walls, and many other effects where obvious repetition of a texture is required. However, in other cases wrapping is used to create a more subtle effect, where the edges of each copy of the texture are not quite as obvious. In order to make the edges of the wrapping less apparent, texture images must be created in such a way that the matching edges of the texture image are equal.

Wrapping creates a toroidal mapping of the texture, as tiling matches the bottom edge of the texture with the top edge of the neighboring copy (and vice versa), and the left edge of the texture with the right edge of the neighboring copy (and vice versa). This is equivalent to rolling the texture into a tube (matching the top and bottom edges), and then bringing together the ends of the tube, matching the seams. Figure 6.15 shows this toroidal matching of texture edges. In order to avoid the sharp discontinuities at the texture repetition boundaries, the texture must be painted or captured in such a way that it has "toroidal topology"; that is, the neighborhood of its top edge is equal to the neighborhood of its bottom edge, and the neighborhood of its left edge must match the neighborhood of its right edge. Also, the neighborhood of the four corners must all be equal, as they come together in a point in the mapping. This can be a tricky process for complex textures, and various algorithms have

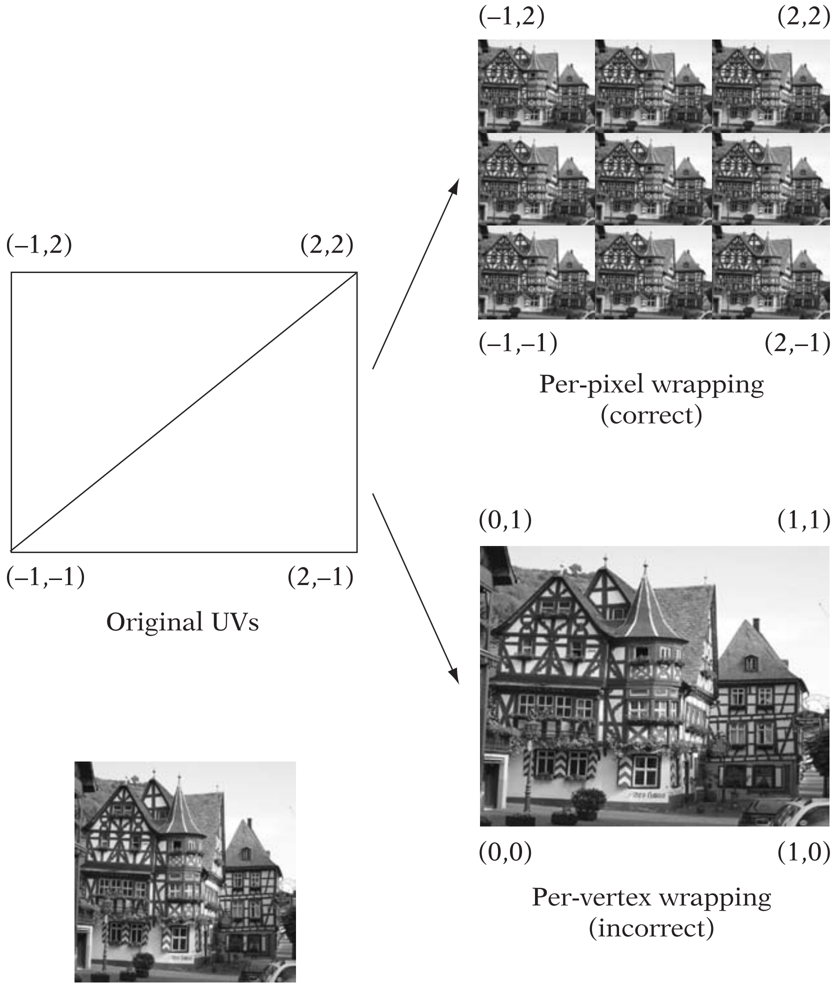


FIGURE 6.14 Computing texture wrapping.

been built to try to create toroidal textures automatically. However, the most common method is still to have an experienced artist create the texture by hand to be toroidal.

The other common method used to map unbounded texture coordinates is called *texture clamping*, and is defined as

$$clamp(u) = max(min(u, 1.0), 0.0)$$

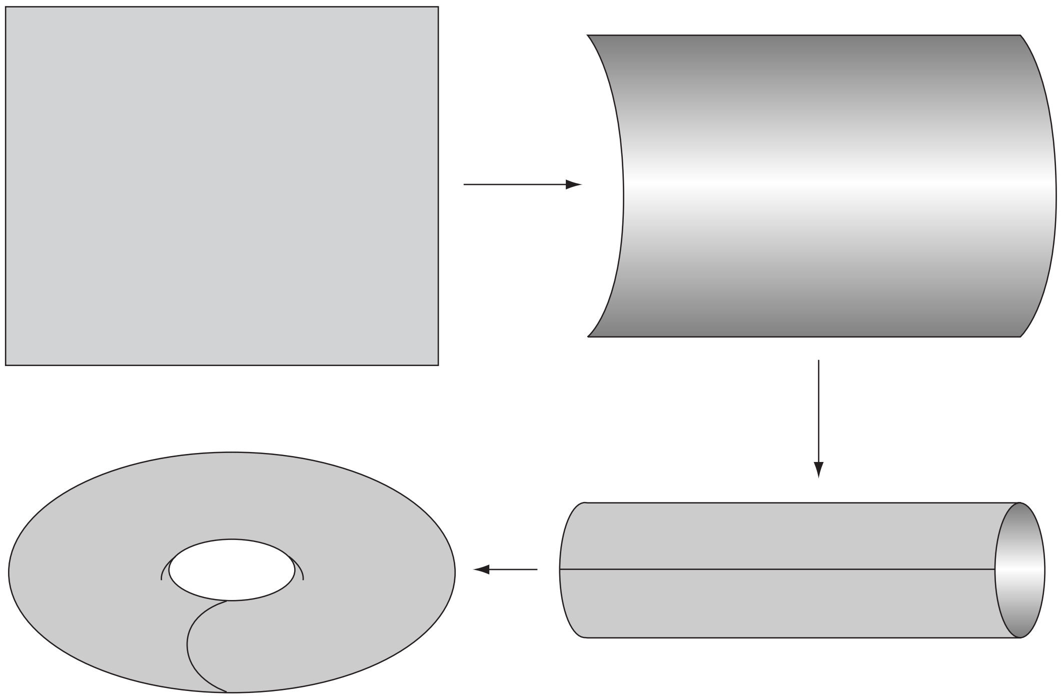


FIGURE 6.15 Toroidal matching of texture edges when wrapping.

Clamping has the effect of simply stretching the border texels (left, right, top, and bottom edge texels) out across the entire section of the triangle that falls outside of the unit square. An example of the same square we've discussed, but with texture clamping instead of wrapping, is shown in Figure 6.16. Note that clamping the vertex texture coordinates is very different from texture clamping. An example of the difference between these two operations is shown in Figure 6.17. Texture clamping must be computed per-sample and has no effect on any sample that would be in the unit square. Per-vertex coordinate clamping, on the other hand, affects the entire mapping to the triangle, as seen in Figure 6.17.

Clamping is useful when the texture image consists of a section of detail on a solid-colored background. Rather than wasting large expanses of texels and placing a small copy of the detailed section in the center of the texture, the detail can be spread over the entire texture but leaving the edges of the texture as the background color.

On many systems clamping and wrapping can be set independently for the two dimensions of the texture. For example, say we wanted to create the effect of a road; black asphalt with a thin set of lines down the center

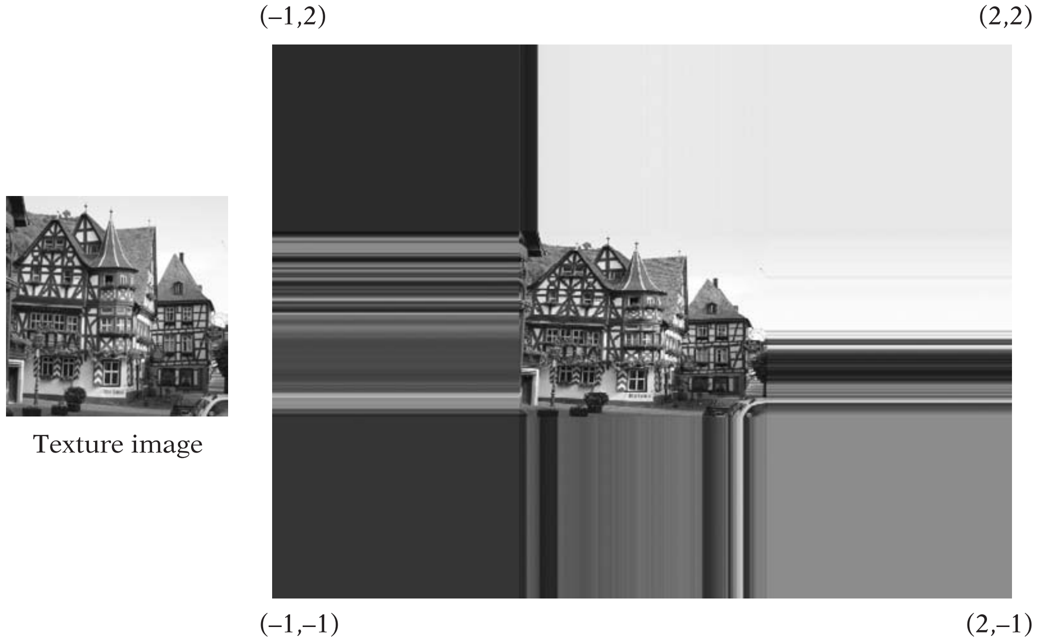


FIGURE 6.16 An example of texture clamping.

of the road. Figure 6.18 shows how this effect can be created with a very small texture by clamping the U dimension of the texture (to allow the lines to stay in the middle of the road with black expanses on either side) and wrapping in the V dimension (to allow the road to repeat off into the distance).

SOURCE CODE DEMO TextureWrapping

OpenGL supports both clamping and wrapping independently in U (which it calls "S") and V (which it calls "T"). The function `glTexParameteri` is used to set these values. The first argument specifies which type of texturing is to be affected (1-, 2-, or 3D), the second the mode and coordinate axis (`GL_TEXTURE_WRAP_S` or `GL_TEXTURE_WRAP_T` in 2D texturing), and the final argument sets the mode. The possible modes are `GL_REPEAT` (wrapping), `GL_CLAMP_TO_EDGE` (clamping), or `GL_CLAMP` (a modified version of clamping that uses a single "edge color" instead of the texture edge; see the OpenGL Programming Guide [83] for details of the behavior of this mode). To create our road example, we would call

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,GL_REPEAT);
```

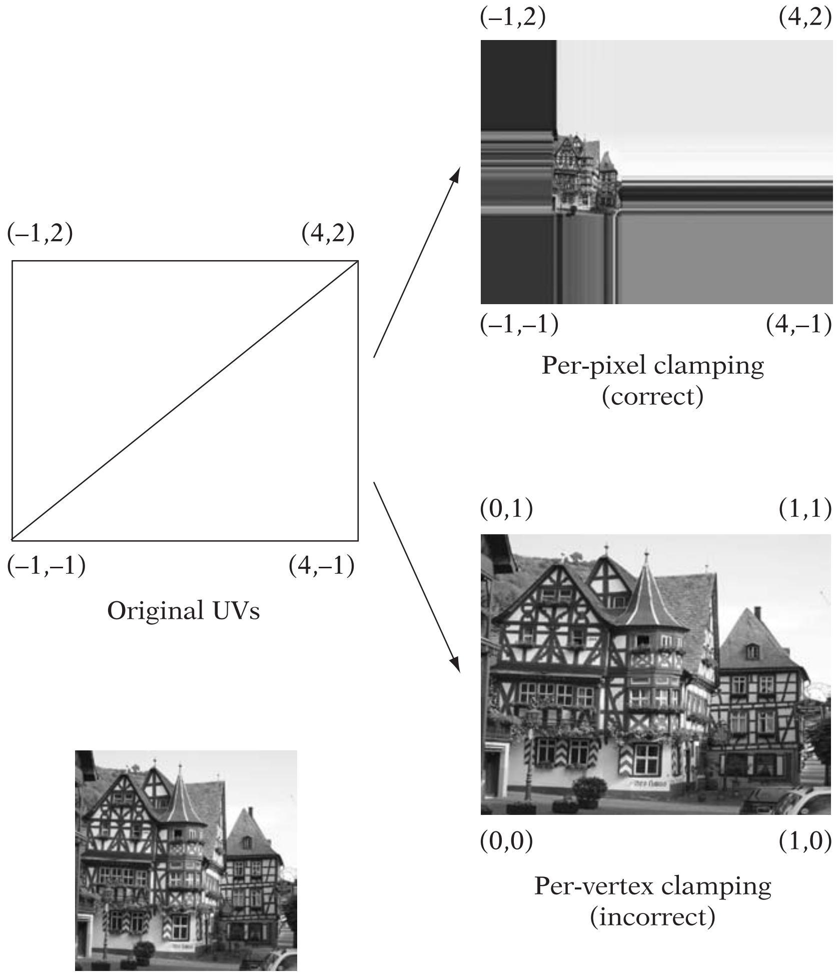


Figure 6.17 Computing texture clamping.

6.9 Reviewing the Steps of Texturing

Unlike basic Gouraud shading (which interpolates the per-vertex values directly as the final sample colors), texturing adds several levels of indirection between the values defined at the vertices (the UV values) and the final sample colors. This is at once the very power of the method and its most confusing aspect. This indirection means that the colors applied to a triangle by texturing can approximate an extremely complex function, far more complex and detailed than the planar function implied by Gouraud shading. However,

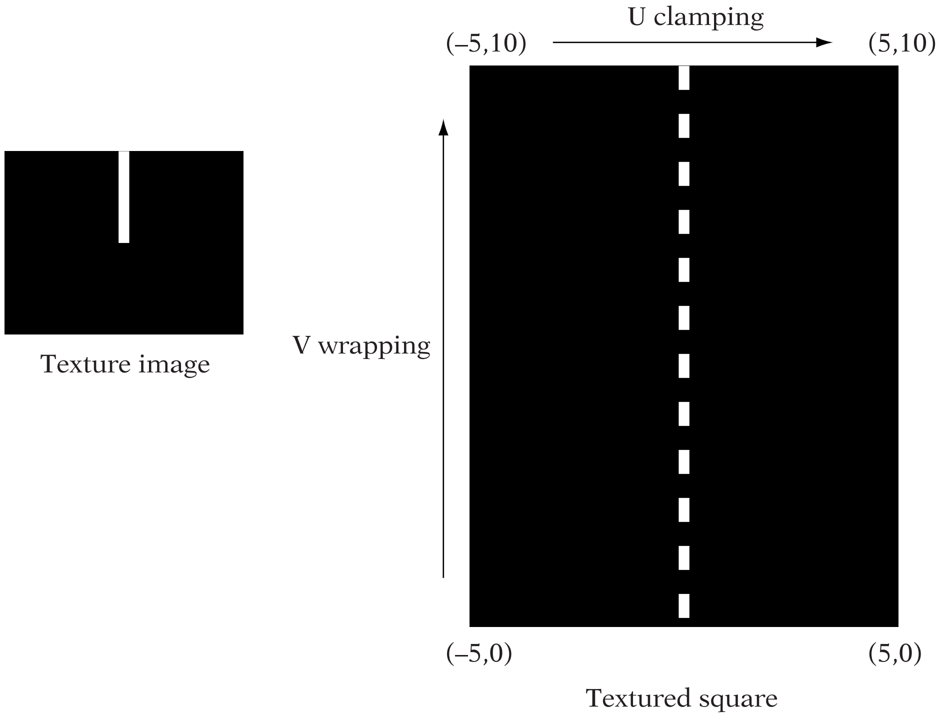


Figure 6.18 Mixing clamping and wrapping in a useful manner.

it also means that there are far more stages in the method whereupon things can go awry. This section aims to pull together all of the previous texturing discussion into a simple, step by step pipeline. Understanding this basic pipeline is key to developing and debugging texturing use in any application.

Texturing is a function that maps per-vertex 2-vectors (the texture coordinates), a texture image, and a group of settings into a per-sample color. The top-level stages are as follows:

1. Map the barycentric s and t values into u and v values, using the affine mapping defined by the three triangle-vertex texture coordinates: (u_1, v_1), (u_2, v_2), and (u_3, v_3). These input s and t values are the barycentric coordinates of the point in the triangle, and should not be confused with OpenGl's similar renaming of u and v:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (u_1 - u_3) & (u_2 - u_3) & u_3 \\ (v_1 - v_3) & (v_2 - v_3) & v_3 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

2. Using the texture coordinate mapping mode (either clamping or wrapping), map the U and V values into the unit square:

$$u_{unit}, v_{unit} = wrap(u), wrap(v)$$

or,

$$u_{unit}, v_{unit} = clamp(u), clamp(v)$$

3. Using the width and height of the texture image in texels, map the U and V values into integral texel coordinates via simple scaling:

$$u_{texel}, v_{texel} = \lfloor u_{unit} \times width \rfloor, \lfloor v_{unit} \times height \rfloor$$

4. Using the texture image, map the texel coordinates into colors using image lookup:

$$C_T = Image(u_{texel}, v_{texel})$$

These steps compose to create the mapping from a point on a given triangle to a color value. The following inputs must be configured, regardless of the specific graphics system:

- The per-vertex texture coordinates
- The texture image to be applied
- The coordinate mapping mode

6.10 Limitations of Texturing

For all of the flexibility that texturing affords the real-time 3D application developer, it still shares several limitations with its simpler cousins, flat and Gouraud shading. All of the methods described thus far assign colors that do not change for any given sample point at runtime. In other words, no matter what occurs in the scene, a fixed point on a given surface will always return the same color.

Real-world scenes are dynamic, with colors that change in reaction to changes in lighting, changes in position, and even changes to the surfaces themselves. Any shading method that relies entirely on values that are fixed over time and scene conditions will be unable to create truly convincing, dynamic worlds. Methods that can represent real-world lighting and the dynamic nature of moving objects are needed.

A very popular method of achieving these goals is to use a simple, fast approximation of real-world lighting. The next chapter will discuss in detail many aspects of how lighting is approximated in real-time 3D systems. Another method of generating dynamic shading of geometry is so-called procedural shading. While procedural shading has long been popular in off-line renderings for high-quality computer-generated images (such as those for feature films), it has more recently become popular in a simpler form, even in consumer-level 3D hardware. These simpler versions of fully general procedural shading are known as *pixel shaders* and *vertex shaders* and are discussed in the next section.

6.11 Procedural Colors and Shaders

At the highest level, the most powerful method of assigning colors to geometry would be to allow a completely generic, arbitrarily complex function to specify the color of a triangle at any given point. Such a method is often called *procedural texturing*, or *procedural shading*, so-called because the colors are generated by a small program or procedure, rather than directly from existing per-vertex or per-triangle colors. Such general procedural shaders are the accepted norm in non–real-time, photorealistic rendering, because they offer almost unlimited flexibility to the programmer or artist. However, by their very nature, these complex procedural shaders can require large amounts of computation per sample. While such a system is well-suited for the film industry, where single frames can be allowed hours to render on a high-end workstation, they are not as well-suited for real-time rendering, where an entire frame (often over a million samples) must be rendered in under one-thirtieth of a second on a consumer PC with a 3D graphics accelerator.

Consumer 3D hardware has advanced at an incredible rate, and most consumer 3D hardware built today supports a limited version of this general method via so-called vertex shaders and pixel shaders (also known as vertex programs and fragment programs in OpenGL), very simple programs that are run either per vertex (vertex shaders) or per sample (pixel shaders) to determine the color of a triangle at a point. These shaders can create incredible dynamic effects. The vertex and pixel shader standards, such as those set by DirectX, impose some basic limits to ensure that hardware can implement the range of possible shaders efficiently and consistently.

Pixel and vertex shader standards also avoid the considerable pain that preshader PC 3D graphics programmers spent testing and coping with the hundreds of "capability flags" that each piece of 3D hardware returned to describe their feature set. These preshader capability flags led to enormous amounts of renderer code to handle all of the various cases for different hardware cards. In fact, developers writing PC-based 3D renderers prior to the shader standards sometimes ended up having to actually query the name of the hardware card to enable or disable a block of code in their renderer. This was a fragile technique that caused no end of game-compatibility issues for end users.

The DirectX shader standard includes version numbers, which have allowed the standard to be upgraded over time, allowing for new features with backwards compatibility. However, this places a greater burden on a programmer who wishes to take advantage of these new features while still enabling their application to run on older hardware. If a shader is written to use instructions or limitations that were expanded for pixel shader version 2.0, for example, that shader cannot be used on a piece of 3D hardware that only supports pixel shaders up to version 1.0. The shader author will need to include another, more limited shader that is 1.0-compliant in order to work on the older hardware. Limitations that existed in some of the older shader versions include:

- A fixed limit to the number of instructions in the shader (program length)
- No looping or limited flow control (branching)
- Limitations on the number and type of possible arguments to the shading function (inputs)
- A limited number of temporary variables available during computation (“scratch space”)
- Limited instruction set compared to general-purpose processors

Most of these limitations have been avoided in recent versions of the pixel shader standard, but until the current version of shaders has been available in 3D hardware for one or two years, shader authors will need to include limited (generally less interesting) versions of their more complex shaders.

One original hurdle to the acceptance of shaders was the fact that rendering APIs expose shaders to the programmer at a very low level, one that resembles the assembly language of a very simple CPU. For today’s programmers, most of whom are well-versed and experienced in high-level languages, these low-level shading languages are at best cumbersome and at worst foreign. Worse yet, the limitations of these languages made it very difficult to write reusable code, meaning that using shaders in applications could end up involving dozens of pieces of shader code, each written for a different case.

This hurdle is been addressed agressively by hardware and API vendors, who have been working to create and expand high-level languages such as nVIDIA's Cg [75] and Microsoft's HLSL (High-Level Shading Language) for writing shaders. Both of these systems offer high-level languages to specify and compile shaders. These languages are not a complete solution for second-generation shader hardware however, as these shader compilers are still limited by the underlying limitations of current and previous generation shader hardware. Nevertheless, Cg and HLSL have gained acceptance quite rapidly, and there is every reason to believe that low-level shader programming will become less and less common as the shader hardware expands and the shader compilers continue to improve.

Another consideration with pixel and vertex shaders is that they are currently "all or nothing" prospects. In other words, if a developer intends to use a vertex shader for an object in an application, he or she may wish to do so to change only one aspect of the lighting pipeline. However, a shader is responsible for transformation, lighting, and projection of the vertices sent to it. As a result, the shader author wishing to change one aspect of lighting must write the transform and projection code into the shader as well. While this code can often be copied between the shaders in an application (and shaders for the common cases are available on the Internet from hardware vendors), it is still a burden that can turn some developers away from using shaders.

The greatest remaining limitation in current vertex shaders as set by the standards is that a vertex shader is a "one vertex in, one vertex out" pipeline to ensure general parallelism and simplicity. For example, this means that vertex shaders cannot subdivide geometry to add more detail, nor can they use the connectivity information to find or use adjacent vertices. Each vertex is shaded as if it were the only vertex in the model. However, flexibility is added by allowing a wide, customizable set of per-vertex data. In addition to the information normally associated with vertices, such as normals and diffuse colors, pixel shaders can include application- or shader-specific data with each vertex that can be used as inputs to the shader programs. These extra data slots can be used to implement vertex position animation in hardware, special lighting models involving complex surface properties, or almost anything the shader author wishes.

Pixel shaders must also work on a single pixel at a time, but the ability to sample several textures per pixel allows for pixel shaders to create incredibly wide-ranging effects, with textures used almost as general function-lookup tables.

Many of the most popular shaders that have been written and distributed are simply advanced mixtures of the texturing effects described earlier in this chapter along with some lighting tricks, such as those that will be discussed in the next chapter. As a result, rather than completely replacing the currently known techniques, shaders often build upon them, making knowledge of basic shading and lighting a prerequisite for building truly effective shaders. As shaders and hardware advance, new techniques are surfacing that push shaders beyond these basic methods, including even simple ray tracing and other complex reflection, optical, and atmospheric effects. These shaders benefit not only from their authors' understanding of computer graphics and shaders but also from their knowledge of physically-based lighting models and optical models. It is here where the true promise of shaders is starting to be seen.

The topics discussed throughout this book are reflected in the instruction-set architecture of shaders. The instructions tend either to implement fast versions of notoriously expensive operations or else fold somewhat complex but common computations into a single instruction. Examples include vertex shader instructions such as

- `dp3`, Computes the 3D dot product between two vectors in a single instruction.
- `m4x3`, Computes the multiplication of a 4×3 matrix with a 4-vector and is useful for computing scale-rotate-translate, model-to-camera transforms. Note that this (and some other vertex shader instructions) is a macro instruction and evaluates to a sequence of three actual instructions. On vertex shader hardware that imposes tight limits on instruction count, this can cause trouble if counted as a single instruction.
- `rsq`, Computes a single reciprocal square root (i.e., $1/\sqrt{x}$). This is commonly used as part of a sequence of instructions to normalize a vector.

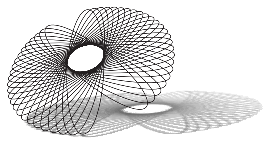
A full discussion of the options available via shaders is outside the scope of this text. For a more detailed introduction to pixel shaders, see [6] and [82]. For a more detailed discussion of cutting-edge shaders, see one of the growing number of shader books, such as [33]. Shader capabilities of consumer 3D hardware is advancing rapidly, well beyond the rate of most book publication, making the Internet an excellent source of up-to-date shader information.

6.12 Chapter Summary

In this chapter, we have discussed a wide range of methods used to map colors onto geometry. These techniques and concepts lay the foundation for the next two chapters, which will discuss a popular method of generating source colors (dynamic lighting), as well as a detailed discussion of the main consumer of these colors (rasterization). While we have already discussed many details regarding the extremely popular shading method known as texturing, this chapter is not the last time we shall mention it. Both of the following two chapters will discuss the ways that texturing affects other stages in the rendering pipeline.

For further reading, popular graphics texts such as Foley, van Dam, Feiner, and Hughes [36] detail other aspects of shading, including methods used for high-end off-line rendering, which are exactly the kinds of methods that are now starting to be implemented as pixel and vertex shaders in real-time hardware. Shader books such as [33] also discuss and provide examples of specific programmable shaders that implement these high-end shading methods and can serve as springboards for further experimentation.

Chapter 7 Lighting



7.1 Introduction

Much of the way we perceive the world visually, especially in terms of depth perception, is based on the way objects in the world react to lighting. This is especially true when the lighting in the visible scene is changing or the lights or objects are moving. While parallax (the apparent motion of objects with respect to one another as the viewpoint changes) is the strongest perceptual signal of depth and relative object position, changes in lighting also make a strong impact.

The coloring methods we have discussed so far, while powerful, use colors that are statically assigned at content creation time (by the artist) or at load-time (by the application). These colors do not change on a frame-to-frame basis. At best, the colors represent a "snapshot" of the scene lighting at a given moment for a given configuration of objects. For example, imagine a simple room scene, containing three lights at fixed positions. Assume further that we cannot move any of the objects in the room. Given these (very limiting) assumptions, to model all possible light-switch positions, we would still need to generate $2^3 = 8$ completely independent sets of textures (or vertex colors) for the room. Even if we did create these eight texture sets, any shiny objects in the room still would not look realistic as the camera moved around the room. Clearly, we need a dynamic method of rendering lighting in real time. The following sections will discuss the details of a popular set of methods for approximating lighting for real-time rendering, as well as examples of how these methods are exposed via OpenGL. Another popular rendering API, Direct3D, uses a slightly different lighting model, but the two are not completely divergent by any means. In order to avoid confusion, we will discuss only the OpenGL model.

7.2 Basics of Light Approximation

The physical properties of light are incredibly complex. Even relatively simple scenes could never be rendered realistically without "cheating." In a sense, all of computer graphics is little more than cheating — finding the cheapest-to-compute approximation for a given situation that will still result in a realistic image. Even non-real-time, photorealistic renderings are only approximations of reality, trading off accuracy for ease and speed of computation.

Real-time renderings are even more superficial approximations. Light in the real world reflects, scatters, diffracts, and bounces around the environment. Real-time 3D lighting generally models only direct lighting, the light that comes along an unobstructed path from light source to surface. Worse yet, many basic real-time lighting systems (including OpenGL) do not support automatic shadowing — objects located between the object being lit and the light source are ignored in the name of efficiency. However, despite these limitations, basic lighting can have *tremendous* effects on the overall impression of a rendered 3D scene.

Lighting in real-time 3D generally involves data from at least three different sources: the surface configuration (vertex position, normal), the surface material (how the surface reacts to light), and the light source properties (the way the light sources emit light).

7.2.1 Measuring Light

In order to understand the mathematics of lighting, even the simplified, non-physical approximation used by OpenGL, it is helpful to understand a little bit about how light is actually measured. The simplest way to understand how we measure light is in terms of an idealized light bulb and an idealized surface being lit by that bulb. To understand both the *brightness* and *luminance* (these are actually two different concepts; we will define them in the following section) of a lit surface, we need to measure and track the following path from end to end:

- The amount of light generated by the bulb
- The amount of light reaching the surface from the bulb
- The amount of light reaching the viewer from the surface

Each of these is measured and quantified differently. First, we need a way of measuring the amount of light being generated by the light bulb. Light bulbs are generally rated according to several different criteria. The number most people think of with respect to light bulbs is wattage; for example, we think of a 100-watt light bulb as being much brighter than a 25-watt light bulb, and generally, this is true. Wattage in this case is a measure of the electrical power consumed by the bulb in order to create light. It is *not* a direct measure of the amount of light actually *generated* by the bulb. In other words, two light bulbs may consume the same wattage (say, 100 watts) but produce different amounts of light — one type of bulb may simply be more efficient at converting electricity to light. So what is the measure of light output from the bulb?

Overall light output from a light source is a measure of power: light energy per unit time. This quantity is called *luminous flux*. The unit of luminous flux is the *lumen*. The luminous flux from a light bulb is measured in lumens, a quantity that is generally listed on boxes of commercially available light bulbs, near the wattage rating. However, lumens are not how we measure the amount of light that is incident upon a surface.

There are several different ways of measuring the light incident upon a surface. The one that will be of greatest interest to us is *illuminance*. Illuminance is a measure of the amount of luminous flux falling on a given area of surface. Illuminance is also called *luminous flux density*, as it is the amount of luminous flux per unit area. It is measured in units of *lux*, which are defined as lumens per meter squared. Illuminance is an important quantity because it measures not only the light power (in lumens), but also the area over which this power is distributed (in square meters). Given a fixed amount of luminous flux, increasing the surface area over which it is distributed will decrease the illuminance proportionally. We will see this property again later, when we discuss the illuminance from a point light source. Illuminance in this case is only the light incident upon a surface — *not* the amount reflected from the surface.

Light reflection from a surface depends on a lot of properties of the surface and the geometric configuration. We will cover approximations of reflection later in this chapter. However, the final step in our list of lighting measurements is to define how we measure the reflected light reaching the viewer from the surface. The quantity used to measure this is *luminance*, which is defined as illuminance per unit solid angle. The unit of luminance is the *nit*, and this value is the closest of those we have discussed to representing "brightness." However, brightness is a perceived value and is not linear with respect to luminance, due to the response curve of the human visual system. For details of the relationship between brightness and luminance, see [20].

The preceding quantities are photometric; that is, they are weighted by the human eye's response to different wavelengths of light. The field of radiometry studies the measurement of analogous quantities that do not include this physiological weighting. The radiometric equivalent of illuminance is *irradiance* (measured in watts per meter squared), and the equivalent of luminance is *radiance*. These radiometric units and quantities are relevant to anyone working with computer graphics, as they are commonly seen in the field of non-real-time rendering, especially in techniques known collectively as *global illumination* (see [19]).

7.2.2 Light as a Ray

Our discussion of light sources will treat light from a light source as a collection of rays, or in some cases simply as vectors. These rays represent infinitely narrow "shafts" of light. This representation of light will make it much simpler to approximate light-surface interaction. Our light rays will often have RGB colors or scalars associated with them that represent the "intensity" (and in the case of RGB values, the color) of the light incident upon a surface. While this value is often described in OpenGL literature as "brightness" or even "luminance," these terms are descriptive rather than physically based. In fact, these intensity values are more closely related to and roughly approximate the *illuminance* incident upon the given surface from the light source.

As we shall see, many low-level graphics systems (such as OpenGL) light an object without considering any other objects in the scene. As a result, no shadowing is computed. Computing even basic light occlusion can be extremely complex, since it involves determining if any object in the scene blocks the path between the current light and the point being lit. In fact, at its most basic, the operation is one of picking: generating a ray between the light position and the point being lit and checking to see if this ray intersects any objects. A technique known as *ray tracing* (see [40]) uses ray-object intersection to track the way light bounces around a scene. Very convincing shadows (and reflections) could be computed using ray tracing, and the technique was very popular in the 1980s and 1990s for non-real-time rendering. Owing to its computational complexity, this method is not generally used in real-time lighting, but shadows are sometimes approximated using other tricks (see [6], [81], [82], or Chapter 13 of Eberly [27]).

7.3 Lighting Approximation (OpenGL)

For the purposes of introducing a real-time lighting equation, we will discuss an approximation that is based on OpenGL's lighting model (or "pipeline"), specifically mentioning when our discussion strays from the model laid out in the OpenGL standard. OpenGL's lighting model is both standard and similar to those in other major graphics APIs. Initially, we will speak in terms of lighting "a sample": a generic point in space that may or may not represent a triangle or a vertex in a tessellation. We will attempt to avoid the concepts of vertices and triangles in this discussion, preferring to refer to a general point on a surface, along with a local surface normal and a surface material. (As will be detailed later, a surface material contains all of the information needed to determine how an object's surface reacts to lighting.) As we've discussed, OpenGL's lighting model does not represent the "real world" — there are many simplifications required for real-time lighting performance.

By default, OpenGL uses the supplied vertex colors directly. In order to switch from direct use of the static vertex colors to real-time lighting computations, use

```
glEnable(GL_LIGHTING);
```

To switch lighting off and return to static coloring, use

```
glDisable(GL_LIGHTING);
```

7.4 Types of Light Sources

The next few sections will discuss the common types of light sources that appear in real-time 3D systems. Each section will open with a general discussion of a given light source, followed by coverage in mathematical terms, and close with the specifics of implementation in OpenGL, along with any interesting results of or reasons for OpenGL's design decisions. The discussion will progress (roughly) from the simplest (and least computationally expensive) light sources to the most complex.

For each type of light source, we will be computing two important values: the unit-vector $\hat{\mathbf{L}}$ (here, we break with our notational convention of lower-case vectors in order to make the equations more readable) and the scalar i_L. The vector $\hat{\mathbf{L}}$ is the *light direction vector* — it points from the current surface sample point P_V, *toward* the source of the light.

The scalar i_L is the light *intensity* value, which is a rough approximation of the illuminance from the light source at the given surface location P_V. With some types of lights, there will be per-light *tuning* values that adjust the function that defines i_L. In addition, in each of the final lighting term equations, we will also modulate in an RGB color light intensity value that scales i_L. These color terms are of the form L_A, L_D, and so on. They will be defined per light and per lighting component and will (in a sense) approximate a scale factor upon the overall luminous flux from the light source.

The values $\hat{\mathbf{L}}$ and i_L do not take any information about the surface itself into account, only the relative geometry between the light source and the sample point in space. Discussion of the contribution of surface orientation (i.e., the surface normal) will be taken up individually, as each type of light and component of the lighting equation will be handled differently.

7.4.1 Directional Lights

A directional light source (also known as an "infinite" light source) is similar to the light of the Sun as seen from Earth. Relative to the size of the Earth, the Sun seems almost infinitely far away, meaning that the rays of light reaching the Earth from the Sun are basically parallel to one another, independent of position on the earth. Consider the source and the light it produces as a single vector. A directional light is defined by a *point at infinity*, P_L. The light source direction is produced by turning the point into a unit vector (by subtracting the position of the origin and normalizing the result):

$$\hat{\mathbf{L}} = \frac{P_L - 0}{|P_L - 0|}$$

Figure 7.1 shows the basic geometry of a directional light. Note that the light rays are the negative (reverse) of the light direction vector $\hat{\mathbf{L}}$, since it points from the surface to the light source.

The value i_L for a directional light is constant for all sample positions:

$$i_L = 1$$

Since both i_L and light vector $\hat{\mathbf{L}}$ are constant for a given light (and independent of the sample point P_V), directional lights are the least computationally

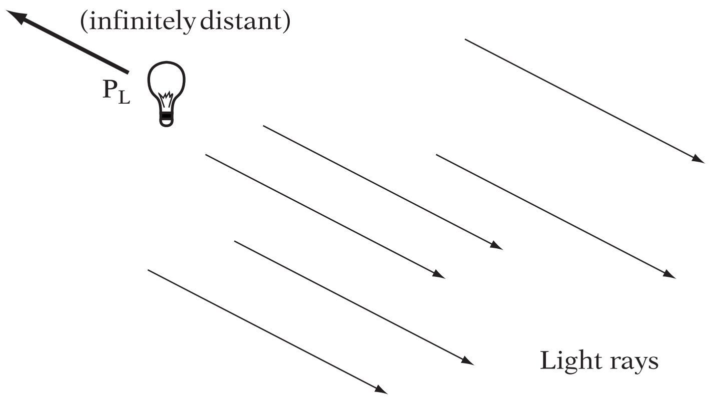


Figure 7.1 The basic geometry of a directional light.

expensive type of light source. Neither $\hat{\mathbf{L}}$ nor i_L needs to be recomputed for each sample.

In OpenGL a directional light is signified by setting the w-coordinate of the desired light's position to zero, causing it to be treated as an affine vector, rather than a point. The x, y, and z components of the light position should be set to the corresponding components of P_L.

OpenGL refers to lights by integer indices. The light at a given index may be of any type and is enabled via the function call:

```
int index;
// ...
glEnable(GL_LIGHT0 + index);
```

where `index` is the desired light (zero-based). The function `glDisable` may be used to turn off a light in the same way. The following code sets light 0 to be a directional light that is located infinitely far away in the direction of the given vector `dir` as follows:

```
GLfloat dir[4];
// ...
dir[3] = 0.0f; // w coord
glLightfv(GL_LIGHT0, GL_POSITION, dir);
```

7.4.2 Point Lights

A point or positional light source (also known as a "local" light source to differentiate it from an infinite source) is similar to a bare light bulb, hanging in space. It illuminates equally in all directions. A point light source is defined by its location, the point P_L. The light source direction produced is

$$\hat{\mathbf{L}} = \frac{P_L - P_V}{|P_L - P_V|}$$

This is the normalized vector that is the difference from the sample position to the light source position. It is not constant per-sample, but rather forms a vector field that points toward P_L from all points in space. This normalization operation is one factor that often makes point lights more computationally expensive than directional lights. While this is not a prohibitively expensive operation to compute *once per light*, we must compute the subtraction of two points and normalize the result to compute this light vector for *each lighting sample* (generally per-vertex for each light) for every frame. Figure 7.2 shows the basic geometry of a point light.

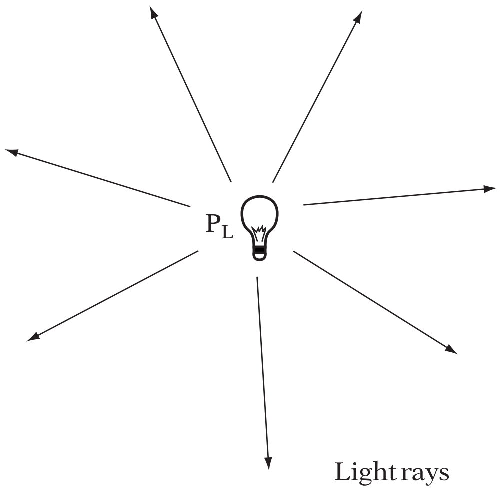


FIGURE 7.2 The basic geometry of a point light.

In OpenGL, point lights are specified with a nonzero *w*-coordinate. The following code sets light 0 to be a positional light that is located at the given position pos.

```
GLfloat pos[4];
// ...
pos[3] = 1.0f; // w coord
glLightfv(GL_LIGHT0, GL_POSITION, pos);
```

Unlike the directional light, a positional light has a nonconstant function defining i_L. This nonconstant intensity function approximates a basic physical property of light known as the *inverse-square law* (which we will detail shortly). Our idealized point light source radiates a constant amount of luminous flux, which we call I, at all times. In addition, this light power is evenly distributed in all directions from the point source's location. Thus, any cone-shaped subset (*a solid angle*) of the light coming from the point source represents a constant fraction of this luminous flux (we will call this I_{cone}). An example of this conical subset of the sphere is shown in Figure 7.3.

Illuminance (the photometric value most closely related to our i_L) is measured as luminous flux per unit area. If we intersect the cone of light with a plane perpendicular to the cone, the intersection forms a disc (see Figure 7.3). This disc is the surface area illuminated by the cone of light. If we assume that this plane is at a distance *dist* from the light center and the radius of

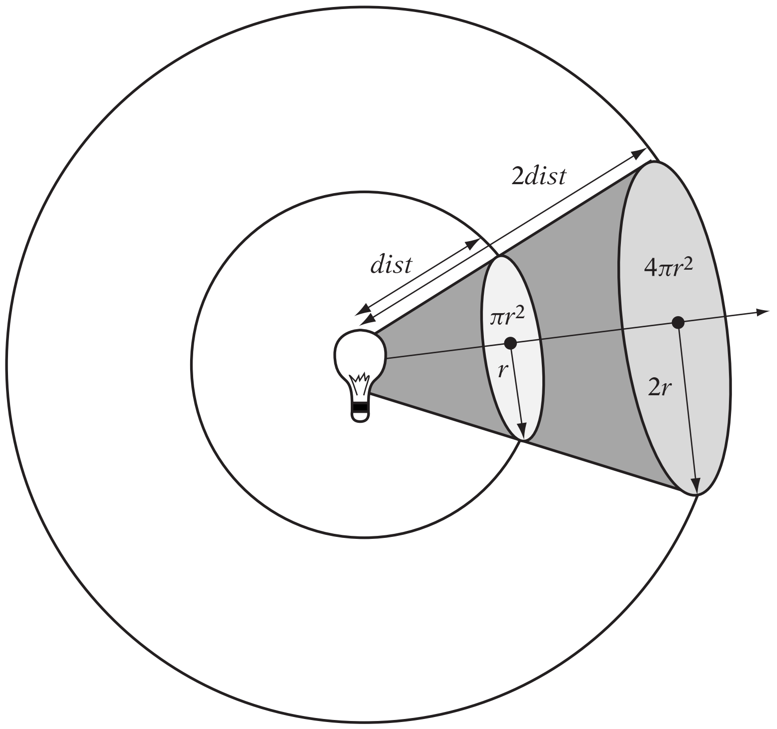


FIGURE 7.3 The inverse-square law.

the resulting disc is r, then the area of the disc is πr^2. The illuminance E_{dist} (in the literature, illuminance is generally represented with the letter E) is proportional to

$$E_{dist} = \frac{power}{area} \propto \frac{I_{cone}}{\pi r^2}$$

However, at a distance of $2dist$, then the radius of the disc is $2r$ (see Figure 7.3). The resulting radius is $\pi(2r)^2$, giving an illuminance E_{2dist} proportional to

$$E_{2dist} \approx \frac{I_{cone}}{\pi(2r)^2} = \frac{I_{cone}}{4\pi r^2} = \frac{E_{dist}}{4}$$

Doubling the distance divides the illuminance by a factor of four, because the same amount of light energy is spread over four times the surface area. This is known as the inverse-square law, and it states that for a point source, the illuminance decreases with the square of the distance from the source. As an example of a practical application, the inverse-square law is the reason why a candle can illuminate a small room that is otherwise completely unlit but will *not* illuminate an entire stadium. In both cases, the candle provides the same amount of luminous flux. However, the actual surface areas that must be illuminated in the two cases are vastly different due to distance.

The inverse-square law results in a basic i_L for a point light equal to

$$i_L = \frac{1}{dist^2}$$

where

$$dist = |P_L - P_V|$$

which is the distance between the light position and the sample position.

Source Code Demo
Distance Attenuation

While exact inverse-square law attenuation is physically correct, it does not always work well artistically or perceptually. As a result, OpenGL and most other modern graphics APIs support a more general distance attenuation function for positional lights; a general quadratic. Under such a system, the function i_L for a point light is

$$i_L = \frac{1}{k_c + k_l dist + k_q dist^2}$$

The distance attenuation constants k_c, k_l, and k_q are defined per light and determine the shape of that light's attenuation curve. Figure 7.4 is a visual example of constant, linear, and quadratic attenuation curves. The spheres in each row increase in distance linearly from left to right.

The OpenGL light values that map to k_c, k_l, and k_q are `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`, respectively, and are set using `glLight*`. OpenGL defines that *dist* be computed in "eye" or camera coordinates; this specification of the space used is important, as there may be scaling differences between model space, world space, and camera space, which would change the scale of the attenuation.

The attenuation of a point light's intensity by this quadratic can be computationally expensive, as it must be recomputed per-sample. In order to increase performance on some systems, OpenGL applications can leave the attenuation values at their OpenGL defaults, which are $k_c = 1$ and $k_l = k_q = 0$. This disables distance attenuation and can increase performance in some cases.

7.4.3 Spotlights

A spotlight is like a point light source with the ability to limit its light to a cone-shaped region of the world. The behavior is similar to a theatrical spotlight with the ability to focus its light upon a specific part of the scene.

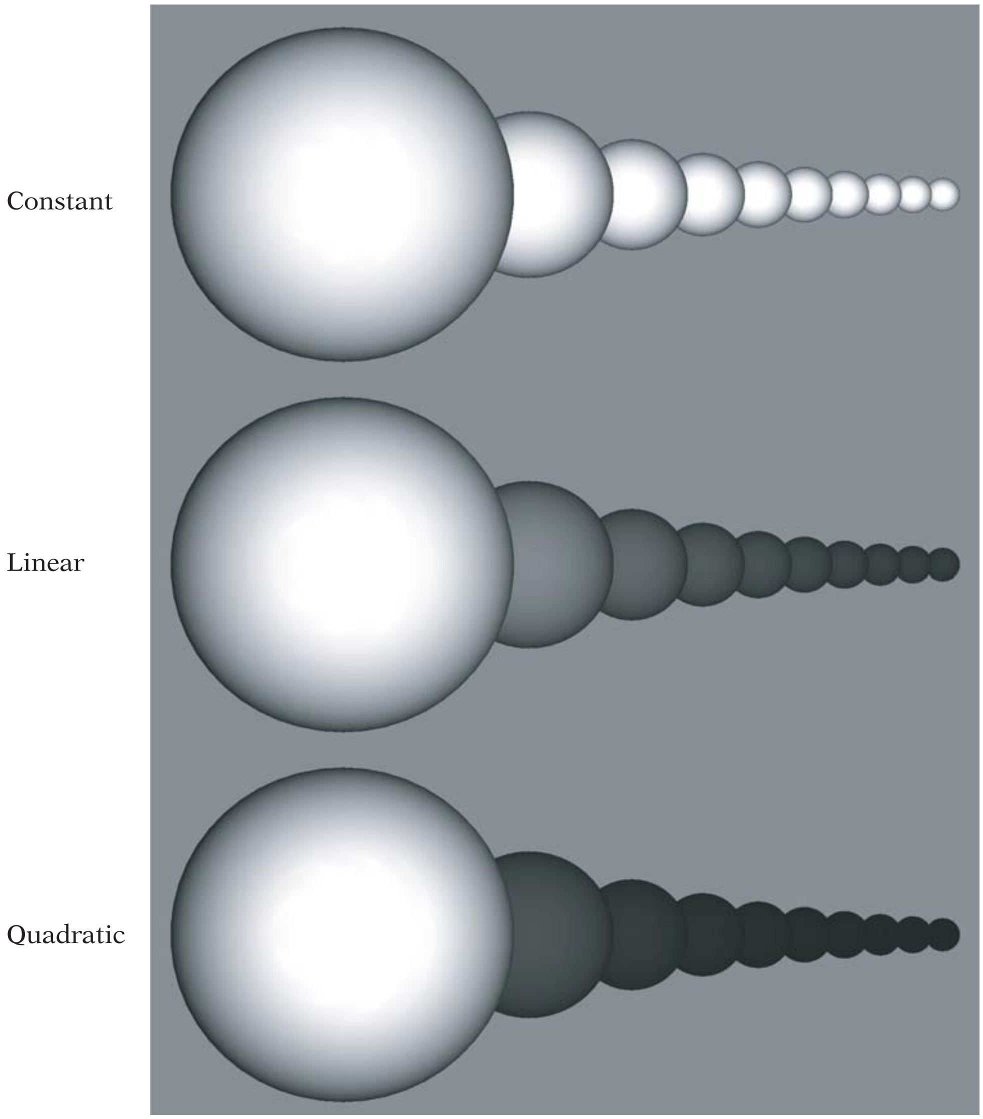


Figure 7.4 Distance attenuation.

In addition to the position P_L that defined a point light source, a spotlight is defined by a direction vector $\mathbf{d}$, a scalar cone angle θ, and a scalar exponent s. These additional values define the direction of the cone and the behavior of the light source as the sample point moves away from the central axis of the cone. The infinite cone of light generated by the spotlight

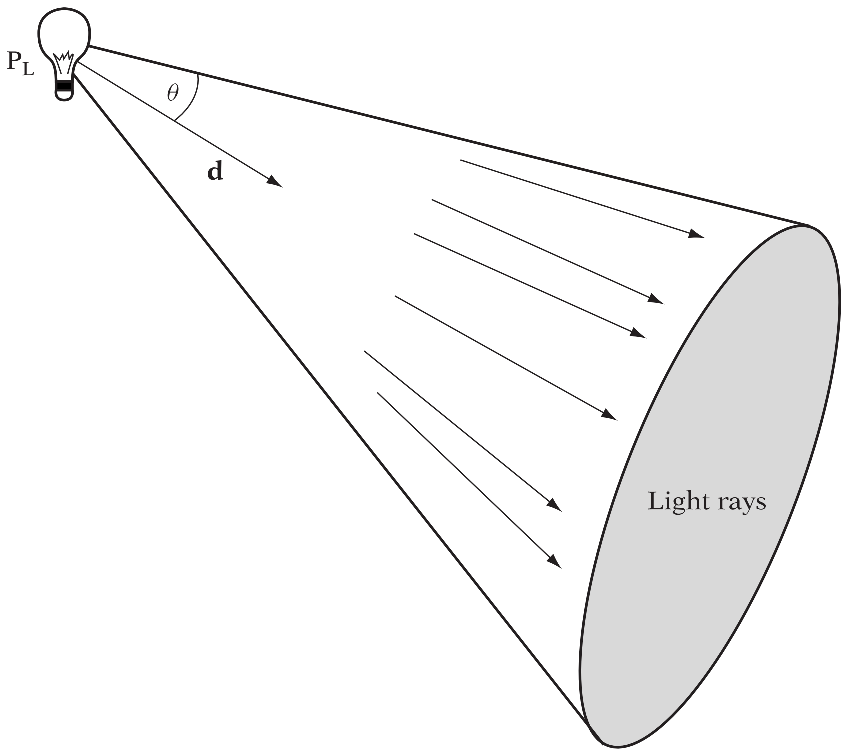


FIGURE 7.5 The basic geometry of a spotlight.

has its apex at the light center P_L, an axis **d** (pointing *toward the base* of the cone), and a half angle of θ. Figure 7.5 illustrates this configuration. The exponent s is not a part of the geometric cone; as will be seen shortly, it is used to attenuate the light within the cone itself.

The light vector is equivalent to that of a point light source:

$$\hat{\mathbf{L}} = \frac{P_L - P_V}{|P_L - P_V|}$$

For a spotlight, i_L is based on the point light function but adds an additional term to represent the focused, conical nature of the light emitted by a spotlight:

$$i_L = \frac{spot}{k_c + k_l dist + k_q dist^2}$$

where

$$spot = \begin{cases} (-\hat{\mathbf{L}} \cdot \mathbf{d})^s, & \text{if } (-\hat{\mathbf{L}} \cdot \mathbf{d}) \geq \cos\theta \\ 0, & \text{otherwise} \end{cases}$$

As can be seen, the *spot* term is 0 when the sample point is outside of the cone. The *spot* term makes use of the fact that the light vector and the cone vector are normalized, causing $(-\hat{\mathbf{L}} \cdot \mathbf{d})$ to be equal to the cosine of the angle between the vectors. We must negate $\hat{\mathbf{L}}$ because it points toward the light, while the cone direction vector **d** points away from the light. Computing the cone term first can allow for performance improvements by skipping the rest of the light calculations if the sample point is outside of the cone. In fact, some graphics systems even check the bounding volume of an object against the light cone, avoiding any spotlight computation on a per-sample basis if the object is entirely outside of the light cone.

Inside of the cone, the light is attenuated via a function that does not represent any physical property but is designed to allow artistic adjustment. The light's i_L function reaches its maximum inside the cone when the vertex is along the ray formed by the light location P_L and the direction **d**, and decreases as the vertex moves toward the edge of the cone. The dot product is used again, meaning that i_L falls off proportionally to

$$\cos^s \omega$$

where ω is the angle between the cone direction vector and the vector between the sample position and the light location $(P_V - P_L)$. As a result, the light need not attenuate smoothly to the cone edge — there may be a sharp drop to $i_L = 0$ right at the cone edge. Adjusting the s value will change the rate at which i_L falls to 0 inside the cone as the sample position moves off axis.

The multiplication of the *spot* term with the distance attenuation term means that the spotlight will attenuate over distance within the cone. In this way, it acts exactly like a point light with an added conic focus. The fact that both of these expensive attenuation terms must be recomputed per-sample makes the spotlight the most computationally expensive type of standard light in most systems. When possible, applications attempt to minimize the number of simultaneous spotlights (or even avoid their use altogether).

Spotlights with circular attenuation patterns are not universal. Another popular type of spotlight (see Warn [111]) models the so-called barn door spotlights that are used in theater, film, and television. Such lights have four metal "doors" around the edge of the light, forming a square. Each of the doors may swing in or out to tighten the light pattern in that direction. Barn door lights allow for much finer-grained control than cone-based spotlights. However, more information is required to model them, as the positions of the four barn doors must be stored and used. Also, the orientation of the "ring" of barn doors must be known, since the light is no longer rotationally symmetrical around its direction vector as it was in a cone-shaped spotlight. Because of these additional computational expenses, conical spotlights are by far the more common form in real-time graphics systems.

Source Code Demo Spotlight

In OpenGL, a spotlight is defined as a point light source with a spotlight cone angle (called the *cutoff angle* in OpenGL) that is $\neq$ 180 degrees. The default spotlight cone angle for a light is 180 degrees, meaning that unless the angle is changed, a positional light will illuminate objects in all directions (i.e., it will not be a spotlight). This default was chosen to ensure both performance and ease of use. Since spotlights are so computationally expensive and can be hard to use (it is easy to select a direction vector that causes the light to point off in the wrong direction, leaving the scene with no light), it is best to require an application to specifically enable them.

The spot cutoff angle is specified *in degrees* using `glLightf`, passing the enumeration `GL_SPOT_CUTOFF` and the angle as a floating-point scalar. Remember that this is actually the half-angle of the cone — the overall *field of view* of the spotlight will be twice this value. Similarly, the spotlight attenuation exponent is set using `glLightf` with an enumeration of `GL_SPOT_EXPONENT`. The spotlight direction vector is set using `glLightfv`, passing the enumeration `GL_SPOT_DIRECTION` and a floating-point 3-vector containing the direction. An example of setting light 0 to a spotlight at the origin, pointing along the x-axis, with a 30-degree cutoff angle and quadratic attenuation follows:

```
GLfloat pos[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_LIGHT_POSITION, pos);

GLfloat dir[3] = { 1.0f, 0.0f, 0.0f };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);

glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 30.0f);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2.0f);
```

7.4.4 Other Types of Light Sources

One type of light source that is not generally supported in low-level, real-time 3D graphics SDKs (including OpenGL) are area light sources, similar to the fluorescent light fixtures seen in most office buildings. The main interest in area light sources are the soft-edged shadows that they produce. These soft-edged shadows occur at shadow boundaries, where the point in partial shadow is illuminated by part of the area light source but not all of it. The shadow becomes progressively darker as the given point can "see" less and less of the area light source. This soft shadow region (called the *penumbra*, as opposed to the fully-shadowed region, called the *umbra*) is highly prized in non–real-time, photorealistic renderings for the realistic quality it lends to the results.

Real-time 3D lighting generally avoids testing per-sample light-object visibility. In fact, soft shadows are even more complicated than hard shadows, as the fraction of the area light that is visible from the given point must be computed and is not just a Boolean visible/not visible switch. Since it is very expensive to compute these soft shadows in a general way in real time, the great benefit of area light sources is lost, and most real-time systems do not support them.

7.5 Surface Materials and Light Interaction

Having discussed the various ways in which the light sources in our model generate light incident upon a surface, we must complete the model by discussing how this incoming light (our approximation of illuminance) is converted (or reflected) into outgoing light (our approximation of luminance) as seen by the viewer or camera. This section will discuss a common real-time model of light/surface reflection.

In the presence of lighting, there is more to surface appearance than a single color. Surfaces respond differently to light, depending upon their composition; for example, unfinished wood, versus plastic, versus metal. Gold-colored plastic, gold-stained wood, and actual gold all respond differently to light, even if they are all the same basic color. Most real-time 3D lighting models take these differences into account with the concept of a material.

Source Code
Demo
Components

A material describes the behavior of an object with respect to light. In our real-time rendering model, a material describes the way a surface generates or responds to four different categories of light: emitted light, ambient light, diffuse light, and specular light. Each of these forms of light is an approximation of real-world light and, put together, they can serve well at differentiating not only the colors of surfaces but also the apparent compositions (shiny versus matte, plastic versus metal, etc.). Each of the four categories of approximated light will be individually discussed.

7.5.1 OpenGL Materials

As with the rest of the chapter, the focus will be on the lighting model that is used by OpenGL. Most of these concepts carry over to other common low-level, real-time 3D SDKs as well, even if the methods of declaring these values and the exact interaction semantics might differ slightly from API to API.

OpenGL uses a single function set to apply all manner of material properties, `glMaterial*`. In order to understand the use of `glMaterial*`, we shall examine an example:

```
GLfloat color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_EMISSION, color);
```

This code illustrates several basic concepts of how OpenGL will handle materials. OpenGL works on the concept of a single "current" material. All calls to `glMaterial*` change the values of the current material. The first argument, `GL_FRONT`, specifies that the value being set is to be applied to the material for the front "side" of the surface. OpenGL actually has *two* current materials, one for the front side of the triangles that form the surface and one for the back side. This makes it easy to render a thin, double-sided surface as a single set of triangles, without separate triangles for the front and back surfaces.

The second and third arguments take the form of many of the other functions that were introduced for light sources. The second parameter specifies the property to be set; in this case, the emissive color of the surface (which will be covered in detail in the following section); and the third parameter specifies the value to which the property is to be set.

Note that when lighting is enabled, alpha values are handled differently than they are in the case of static vertex colors. In the lit case, the alpha value of the surface is the alpha component of one of the surface's material colors (actually, the diffuse material color as will become apparent). The alpha components of all other material and light colors are ignored. No other calculations are performed on alpha values during lighting. The diffuse material color's alpha value is passed on directly as the "lit" alpha value, since lighting is considered to have no effect on the inherent opacity of the surface.

7.6 Categories of Light

7.6.1 Emission

Emission, or emissive light, is the light produced by the surface itself, in the absence of any light sources. Put simply, it is the color and intensity with which the object "glows." Because this is purely a surface-based property, only surface materials (not lights) contain emissive colors. The emissive color of a material is written as M_E. One approximation that is made in real-time systems is the (sometimes confusing) fact that this "emitted" light does not illuminate the surfaces of any other objects. In fact, another common (and perhaps mo re descriptive) term used for emission is *self-illumination*. The fact that emissive objects do not illuminate one another avoids the need for the graphics systems to take other objects into account when computing the light at a given point.

OpenGL allows the emission color of a surface material to be set using `glMaterialfv` and the constant `GL_EMISSION`. The default value is black (i.e., no emission), since the vast majority of objects in most scenes do not glow.

```
GLfloat color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_EMISSION, color);
```

The alpha component of the emission color is ignored.

7.6.2 **Ambient**

Ambient light is the term used in real-time lighting as an "umbrella" under which all forms of indirect lighting are grouped and approximated. Indirect lighting is light that is incident upon a surface not via a direct ray from light to surface, but rather via some other, more complex path. In the real world, light can be scattered by particles in the air, and light can "bounce" multiple times around a scene prior to reaching a given surface. Accounting for these multiple bounces and random scattering effects is very difficult if not impossible to do in a real-time rendering system, so most systems use a per-light, per-material constant for all ambient light.

A light's ambient color represents the color and intensity of the light from a given source that is to be scattered through the scene. The ambient material color represents how much of the overall ambient light the particular surface reflects.

Ambient light has no direction associated with it. However, most lighting models do attenuate the ambient light from each source based on the light's intensity function at the given point, i_L. As a result, point and spotlights do not produce equal amounts of ambient light throughout the scene. This tends to localize the ambient contribution of point and spotlights spatially and keeps ambient light from overwhelming a scene. The overall ambient term for a given light and material is thus

$$C_A = i_L L_A M_A$$

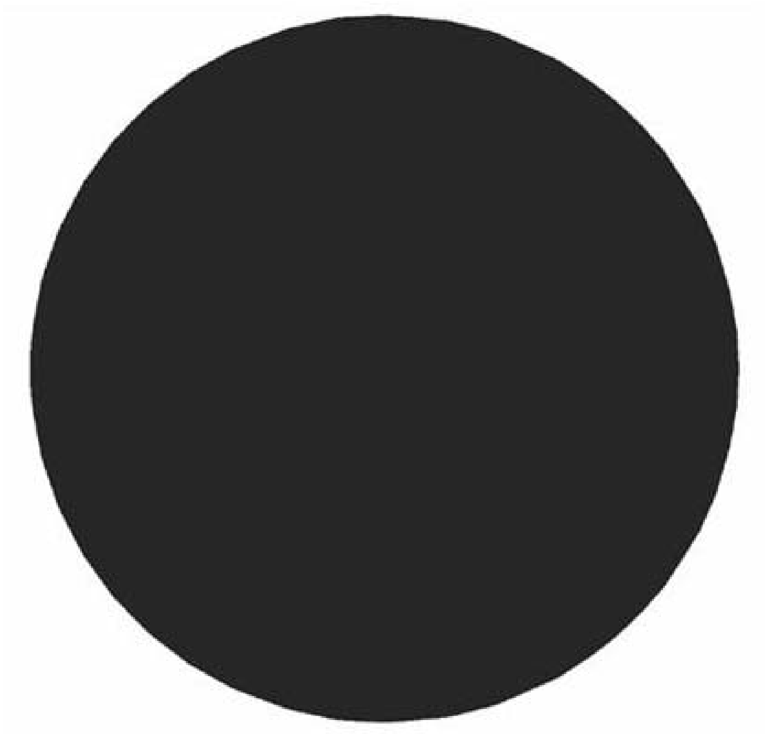


FIGURE 7.6 Sphere lit by ambient light.

where L_A is the light's ambient color, and M_A is the material's ambient color. Figure 7.6 provides a visual example of a sphere lit by purely ambient light. Without any ambient lighting, most scenes will require the addition of many lights to avoid dark areas, leading to decreased performance. Adding some ambient light allows specific light sources to be used more artistically, to highlight parts of the scene that can benefit from the added dimension of dynamic lighting. However, adding too much ambient light can lead to the scene looking "flat," as the ambient lighting dominates the coloring.

In OpenGL both materials and lights have independent ambient colors, each accessed using `GL_AMBIENT`. An example that sets each of these (for the current material and the light at index zero) follows:

```
GLfloat color[4] = { 0.25f, 0.25f, 0.25f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT, color);
```

```
GLfloat light[4] = { 0.0f, 0.0f, 0.5f, 1.0f };
glLightfv(GL_LIGHT0, GL_AMBIENT, light);
```

In addition, OpenGL supports the concept of an *overall* ambient lighting level. This ambient light is independent of any specific light source and represents the overall ambient lighting in the scene (and is written W_A, for world ambient). It is added to the contribution of the other lights and is set globally using

```
GLfloat light[4] = { 0.0f, 0.0f, 0.5f, 1.0f };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, light);
```

The alpha component of the ambient color is ignored.

7.6.3 Diffuse

Diffuse lighting, unlike the previously discussed emissive and ambient terms, represents direct lighting. The diffuse term is dependent on the lighting incident upon a point on a surface from each single light via the direct path. As such, diffuse lighting is dependent on material colors, light colors, i_L, *and* the vectors $\hat{\mathbf{L}}$ and $\hat{\mathbf{n}}$.

The diffuse lighting term treats the surface as a pure diffuse (or matte) surface, sometimes called a *Lambertian reflector*. These surfaces have the property that their luminance is independent of view direction. In other words, like our earlier approximation terms, emissive and ambient, the diffuse term is not view-dependent. The luminance is dependent on only the incident illuminance.

The illuminance incident upon a surface is proportional to the luminous flux incident upon the surface, divided by the surface area over which it is distributed. In our earlier discussion of illuminance, we assumed (implicitly) that the surface in question was perpendicular to the light direction. If we define an infinitesimally narrow ray of light with direction $\hat{\mathbf{L}}$ to have luminous flux I and cross-sectional area δa (Figure 7.7), then the illuminance E incident upon a surface whose normal $\hat{\mathbf{n}} = \hat{\mathbf{L}}$ is

$$E \propto \frac{I}{\delta a}$$

However, if $\hat{\mathbf{n}} \neq \hat{\mathbf{L}}$ (i.e., the surface is not perpendicular to the ray of light), then the configuration is as shown in Figure 7.8. The surface area intersected by the (now oblique) ray of light is represented by $\delta a'$.

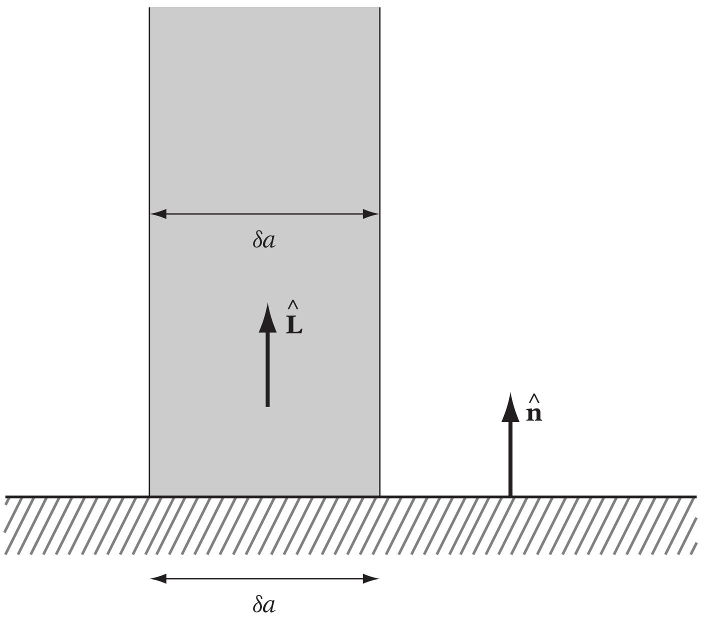


Figure 7.7 A shaft of light striking a perpendicular surface.

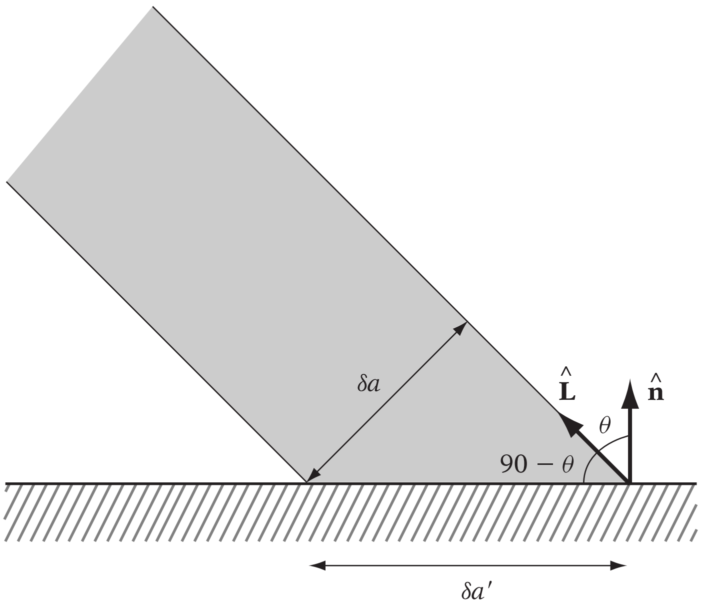


Figure 7.8 The same shaft of light at a glancing angle.

From basic trigonometry and our figure, we can see that

$$\begin{aligned}
\delta a' &= \frac{\delta a}{\sin\left(\frac{\pi}{2} - \theta\right)} \\
&= \frac{\delta a}{\cos\theta} \\
&= \frac{\delta a}{\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}}
\end{aligned}$$

And, we can compute the illuminance E' as follows:

$$\begin{aligned}
E' &\propto \frac{I}{\delta a'} \\
&\propto I\left(\frac{\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}}{\delta a}\right) \\
&\propto \left(\frac{I}{\delta a}\right)(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \\
&\propto E(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})
\end{aligned}$$

Note that if we evaluate for the original special case $\hat{\mathbf{n}} = \hat{\mathbf{L}}$, the result is $E' = E$, as expected. Thus, the reflected diffuse luminance is proportional to $(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})$. Figure 7.9 provides a visual example of a sphere lit by a single light source that involves only diffuse lighting.

Generally, both the material and the light include diffuse color values (M_D and L_D, respectively). The resulting diffuse color for a point on a surface and a light is then equal to

$$C_D = i_L max(0, \hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) L_D M_D$$

Note the $max()$ function that clamps the result to 0. If the light source is behind the surface (i.e., $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} < 0$), then we assume that the back side of the surface obscures the light (self-shadowing), and no diffuse lighting occurs.

In OpenGL, both materials and lights have independent diffuse colors, each accessed using `GL_DIFFUSE`. An example that sets each of these (for the current material and the light at index zero) is

```
GLfloat color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_DIFFUSE, color);

GLfloat light[4] = { 1.0f, 1.0f, 0.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_DIFFUSE, light);
```

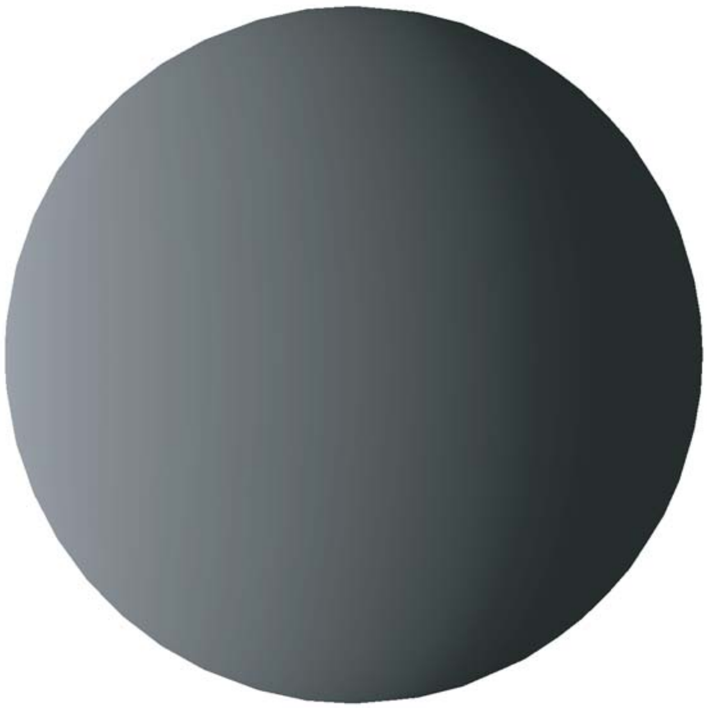


Figure 7.9 Sphere lit by diffuse light.

The alpha component of the diffuse material color defines the alpha value for the surface.

7.6.4 Specular

A perfectly smooth mirror reflects all of the light from a given direction $\hat{\mathbf{L}}$ out along a single direction, the reflection direction $\hat{\mathbf{r}}$. While few surfaces approach completely mirrorlike behavior, most surfaces have at least some mirrorlike component to their lighting behavior. As a surface becomes rougher (at a microscopic scale), it no longer reflects all light from $\hat{\mathbf{L}}$ out along a single direction $\hat{\mathbf{r}}$, but rather in a distribution of directions centered about $\hat{\mathbf{r}}$. This tight (but smoothly attenuating) distribution around $\hat{\mathbf{r}}$ is often called a *specular highlight*, and is often seen in the real world. A classic example is the bright, white "highlight" reflections seen on smooth, rounded plastic objects. The specular component of real-time lighting is an entirely empirical approximation of this reflection distribution, specifically designed to generate these highlights.

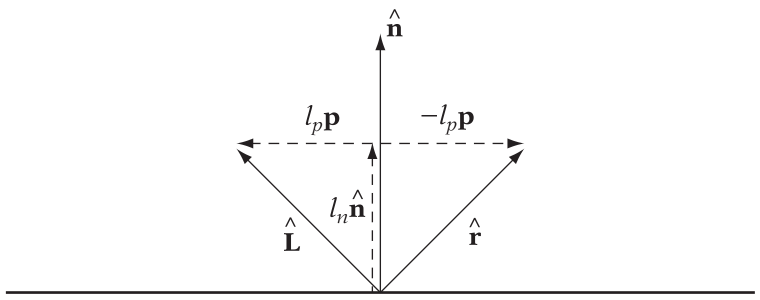


FIGURE 7.10 The relationship between the surface normal, light direction, and the reflection vector.

Because specular reflection represents mirrorlike behavior, the intensity of the term is dependent on the relative directions of the light ($\hat{\mathbf{L}}$), the surface normal ($\hat{\mathbf{n}}$), *and* the viewer ($\hat{\mathbf{v}}$). Prior to discussing the specular term itself, we must introduce the concept of the light reflection vector, $\hat{\mathbf{r}}$. Computing the reflection of a light vector $\hat{\mathbf{L}}$ about a plane normal $\hat{\mathbf{n}}$ involves negating the component of $\hat{\mathbf{L}}$ that is perpendicular to $\hat{\mathbf{n}}$. We do this by representing $\hat{\mathbf{L}}$ as the weighted sum of $\hat{\mathbf{n}}$ and a unit vector $\hat{\mathbf{p}}$ that is perpendicular to $\hat{\mathbf{n}}$ (but in the plane defined by $\hat{\mathbf{n}}$ and $\hat{\mathbf{L}}$), as follows and as depicted in Figure 7.10.

$$\hat{\mathbf{L}} = l_n\hat{\mathbf{n}} + l_p\hat{\mathbf{p}}$$

The reflection of $\hat{\mathbf{L}}$ about $\hat{\mathbf{n}}$ is then

$$\hat{\mathbf{r}} = l_n\hat{\mathbf{n}} - l_p\hat{\mathbf{p}}$$

We know that the component of $\hat{\mathbf{L}}$ in the direction of $\hat{\mathbf{n}}$ (l_n) is the projection of $\hat{\mathbf{L}}$ onto $\hat{\mathbf{n}}$, or

$$l_n = \hat{\mathbf{L}} \cdot \hat{\mathbf{n}}$$

Now we can compute $l_p\hat{\mathbf{p}}$ by substitution of our value for l_n:

$$\begin{aligned}
\hat{\mathbf{L}} &= l_n\hat{\mathbf{n}} + l_p\hat{\mathbf{p}} \\
\hat{\mathbf{L}} &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} + l_p\hat{\mathbf{p}} \\
l_p\hat{\mathbf{p}} &= \hat{\mathbf{L}} - (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}
\end{aligned}$$

So, the reflection vector $\hat{\mathbf{r}}$ equals

$$\begin{aligned}\hat{\mathbf{r}} &= l_n\hat{\mathbf{n}} - l_p\hat{\mathbf{p}} \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - l_p\hat{\mathbf{p}} \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - (\hat{\mathbf{L}} - (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}) \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - \hat{\mathbf{L}} + (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \\ &= 2(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - \hat{\mathbf{L}}\end{aligned}$$

The specular term itself is designed specifically to create an intensity distribution that reaches its maximum when the view vector $\hat{\mathbf{v}}$ is equal to $\hat{\mathbf{r}}$, that is, when the viewer is looking directly at the reflection of the light vector. The intensity distribution falls off toward zero rapidly as the angle between the two vectors increases, with a "shininess" control that adjusts how rapidly the intensity attenuates. The term is based on the following formula:

$$(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^{m_{shine}} = (\cos\theta)^{m_{shine}}$$

where θ is the angle between $\hat{\mathbf{r}}$ and $\hat{\mathbf{v}}$. The shininess factor m_{shine} controls the size of the highlight; a smaller value of m_{shine} leads to a larger, more diffuse highlight, which makes the surface appear more dull and matte; whereas, a larger value of m_{shine} leads to a smaller, more intense highlight, which makes the surface appear shiny. This shininess factor is considered a property of the surface material and represents how smooth the surface appears. Generally, the complete specular term includes specular colors defined on both the light and material (L_S and M_S), which allow the highlights to be tinted a given color. The specular light color is often set to the diffuse color of the light, since a colored light generally creates a colored highlight. In practice, however, the specular color of the material is more flexible. Plastic and clear-coated surfaces (such as those covered with clear varnish), whatever their diffuse color, tend to have white highlights, while metallic surfaces tend to have tinted highlights. For a more detailed discussion of this and several other (more advanced) specular reflection methods, see Chapter 16 of [36]. A visual example of a sphere lit from a single light source providing only specular light is shown in Figure 7.11. The complete specular lighting term is

$$C_S = \begin{cases} i_L max(0, (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}))^{m_{shine}} L_S M_S, & \text{if } \hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Note that, as with the diffuse term, a self-shadowing conditional is applied ($\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0$). However, unlike the diffuse case, we must make this term explicit, as the specular term is not directly dependent upon $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}$. Simply clamping the

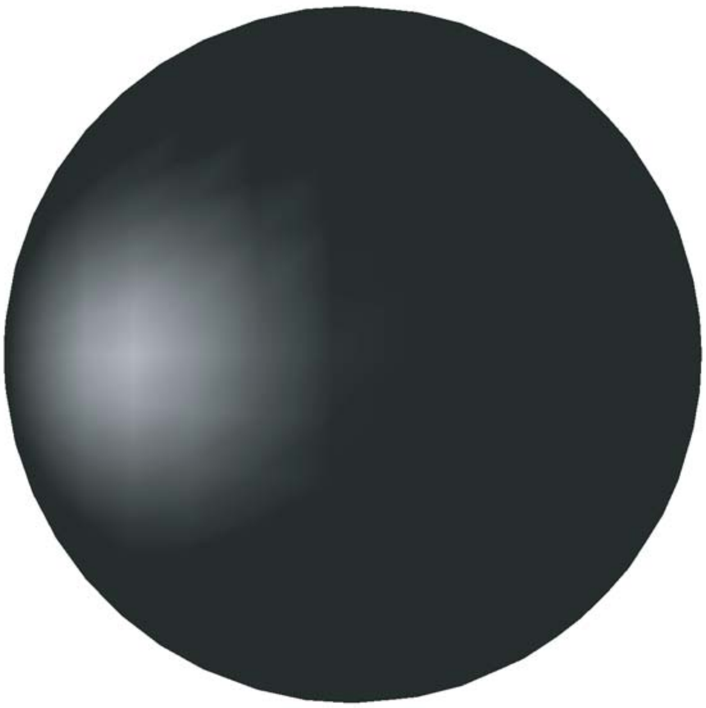


Figure 7.11 Sphere lit by specular light.

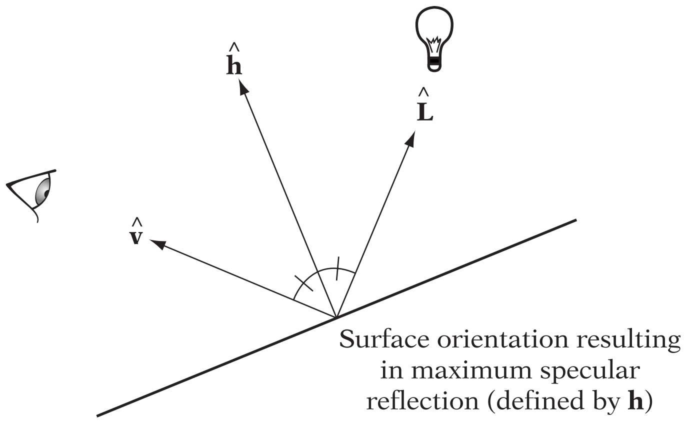
specular term to be greater than 0 could allow objects whose normals point away from the light to generate highlights, which is not correct. In other words, it is possible for $\hat{\mathbf{r}} \cdot \hat{\mathbf{v}} > 0$, even if $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} < 0$.

In OpenGL, both materials and lights have specular components but only materials have specular exponents, as the specular exponent represents the shininess of a particular surface:

```
GLfloat color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glMaterialfv(GL_FRONT, GL_SPECULAR, color);
glMaterialf(GL_FRONT, GL_SHININESS, 10.0f);

GLfloat light[4] = { 0.0f, 1.0f, 0.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_SPECULAR, light);
```

The alpha component of the specular color is ignored.



$\hat{\mathbf{h}}$
$\hat{\mathbf{L}}$
$\hat{\mathbf{v}}$
Surface orientation resulting in maximum specular reflection (defined by **h**)

Figure 7.12 The specular halfway vector.

Infinite Viewer Approximation

One of the primary reasons that the specular term is the most expensive component of lighting is the fact that a normalized view and reflection vector must be computed for each sample, requiring at least one normalization per sample, per light. However, there is another method of approximating specular reflection that can avoid this expense in common cases. This method is based on a slightly different approximation to the specular highlight geometry, along with an assumption that the viewer is "at infinity" (at least for the purposes of specular lighting).

Rather than computing $\hat{\mathbf{r}}$ directly, the OpenGL method uses what is known as a "halfway" vector. The halfway vector is the vector that is the normalized sum of $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$:

$$\hat{\mathbf{h}} = \frac{\hat{\mathbf{L}} + \hat{\mathbf{v}}}{|\hat{\mathbf{L}} + \hat{\mathbf{v}}|}$$

The resulting vector bisects the angle between $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$. This halfway vector is equivalent to the surface normal $\hat{\mathbf{n}}$ that would generate $\hat{\mathbf{r}}$ such that $\hat{\mathbf{r}} = \hat{\mathbf{v}}$. In other words, given fixed light and view directions, $\hat{\mathbf{h}}$ is the surface normal that would produce the maximum specular intensity. So, the highlight is brightest when $\hat{\mathbf{n}} = \hat{\mathbf{h}}$. Figure 7.12 is a visual representation of the configuration, including the surface orientation of maximum specular reflection. The resulting (modified) specular term is

$$C_S = \begin{cases} i_L max(0, (\hat{\mathbf{h}} \cdot \hat{\mathbf{n}}))^{m_{shine}} L_S M_S, & \text{if } \hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

By itself, this new method of computing the specular highlight would not appear to be any better than the reflection vector system. However, if we assume that the viewer is at infinity, then we can use a constant view vector for all vertices, generally the camera's view direction. This is analogous to the difference between a point light and a directional (infinite) light. Thanks to the fact that the halfway vector is based only on the view vector and the light vector, the infinite viewer assumption can reap great benefits when used with directional lights. Note that in this case, both $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$ are constant across all samples, meaning that the halfway vector $\hat{\mathbf{h}}$ is also constant. Used together, these facts mean that specular lighting can be computed very quickly if directional lights are used exclusively and the infinite viewer assumption is enabled.

By default, OpenGL uses this infinite viewpoint for lighting. While this is technically "less accurate" than using a noninfinite viewpoint (real-world specular highlights move as the viewer translates), the performance benefits are significant, making it a worthwhile default. To cause OpenGL to use the more accurate, "local" viewpoint when computing lighting, call

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

7.7 Combined Lighting Equation

Having covered materials, lighting components, and light sources, we now have enough information to evaluate our full lighting model at a given point. In order to do so, we must take all of the above terms into account, including

- The material properties of the object
- The emissive, ambient, diffuse, and specular components of lighting
- The contributions of multiple, independent lights

For a visual example of all of these components combined, see the lit sphere in Figure 7.13.

When lighting a given point, the contributions from each component of each active light L are summed to form the final lighting equation, which is detailed as follows:

$$\begin{aligned} C_V = {} & \text{Emissive} + \text{World Ambient} \\ & + \sum_{L}^{lights} (\text{Per-light Ambient} + \text{Per-light Diffuse} + \text{Per-light Specular}) \end{aligned}$$

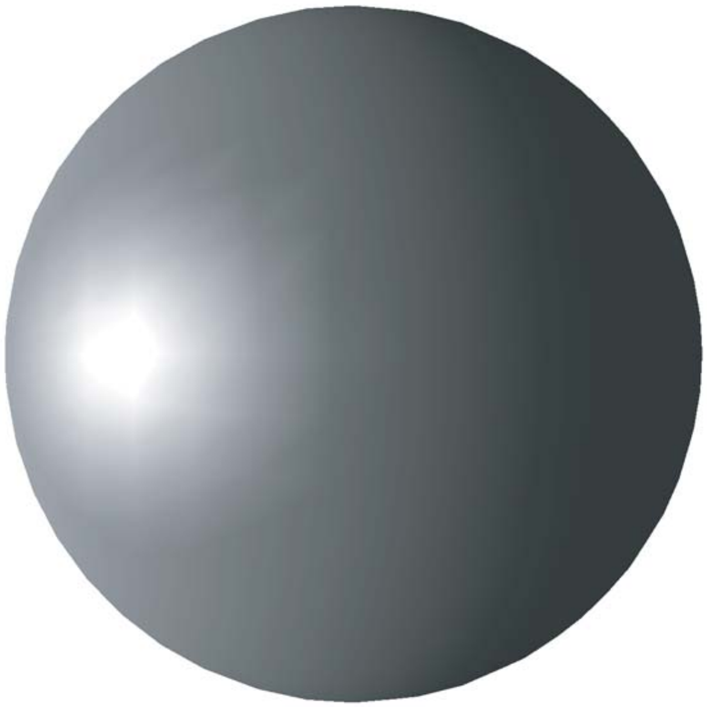


FIGURE 7.13 Sphere lit by a combination of ambient, diffuse, and specular lighting.

$$\begin{aligned} &= M_E + M_A W_A + \sum_{L}^{lights} (C_A + C_D + C_S) \\ A_V &= M_{Alpha} \end{aligned} \tag{7.1}$$

where the results are

1. C_V, the computed, lit RGB color of the sample
2. A_V, the alpha component of the RGBA color of the sample

The intermediate, per-light values used to compute the results are

3. C_A, the per-light ambient term, which is equal to

$$C_A = i_L M_A L_A$$

4. C_D, the per-light diffuse term, which is equal to

$$C_D = i_L M_D L_D (max(0, \hat{\mathbf{L}}_L \cdot \hat{\mathbf{n}}))$$

5. C_S, the per-light specular term, which is equal to

$$C_S = i_L M_S L_S \begin{cases} max(0, (\hat{\mathbf{h}}_L \cdot \hat{\mathbf{n}}))^{m_{shine}}, & \text{if } \hat{\mathbf{L}}_L \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Finally, these intermediate values are computed from the following source data items. Not all of these source values appear in the equations we've covered in this section since some are used indirectly to compute i_L for a given type of light, as detailed for each category of light source:

1. M_{Alpha}, the material's alpha value (generally, the alpha component of the diffuse material color)
2. M_E, the emissive color of the material
3. M_A, the ambient color/reflectance of the material
4. M_D, the diffuse color/reflectance of the material
5. M_S, the specular color/reflectance of the material
6. m_{shine}, the specular shininess of the material
7. L_A, the ambient color of the light L
8. L_D, the diffuse color of the light L
9. L_S, the specular color of the light L
10. $\hat{\mathbf{h}}_L$, the specular halfway vector for the light L and the current sample
11. $\hat{\mathbf{L}}_L$, the light direction vector for the light L and the current sample
12. W_A, the overall world ambient light color
13. i_L, the light intensity value of the light L, which is dependent upon the type of light and the values that follow
14. k_c, k_l, and k_q, the constant, linear, and quadratic distance attenuation factors of the light L
15. θ, the spotlight cone angle of the light L
16. P_L, the position of the light L
17. $\hat{\mathbf{n}}$, the surface normal at the sample
18. P_V, the position of the sample

The combined lighting equation 7.1 brings together all of the properties discussed in the previous sections. Clearly, many different values and components must come together to light even a single sample. This fact can make lighting complicated and difficult to use at first. A completely black rendered image can be the result of many possible errors. However, an understanding of the lighting pipeline can make it much easier to determine which features to disable or change in order to debug lighting issues.

7.8 Lighting and Shading

Thus far, our lighting discussion has focused on computing color at a generic point on a surface, given a location, a surface normal, and a surface material. Another aspect of lighting that is just as important as the basic lighting equation is the question of when and how to evaluate that equation to completely light a surface and how to assign colors to points on the surface for which the lighting equation is not specifically evaluated. This aspect of *dynamic lighting* will involve the use of the shading methods discussed in Chapter 6. However, when selecting between these shading methods, we must take into account the fact that the colors we will supply to our shading functions represent something far more specific than the generic colors discussed in Chapter 6 — namely, dynamic lighting.

Ultimately, a triangle in view is drawn to the screen by coloring the screen pixels covered by that triangle (as will be discussed in more detail in Chapter 8). Any lighting system must be teamed with a shading method that can quickly compute colors for each and every pixel covered by the triangle. The sheer number of pixels that must be drawn per frame (e.g., a sphere that covers 50 percent of a 1024×768 screen will require the shading system to compute colors for $\approx 400,000$ pixels, regardless of the tessellation) requires that many low- to mid-end graphics systems forgo computing the lighting equation for each pixel in favor of another method. Next, we will discuss some of the more popular methods. Some of these methods will be familiar, as they are simply the shading methods discussed in the previous chapter, using results of the lighting equation as source colors.

7.8.1 Flat-shaded Lighting

The simplest shading method applied to lighting is per-triangle, flat shading. This method involves evaluating the lighting equation once per triangle and using the resulting color as C_F, the constant triangle color. The color is assigned to every pixel covered by the triangle. This is the highest-performance

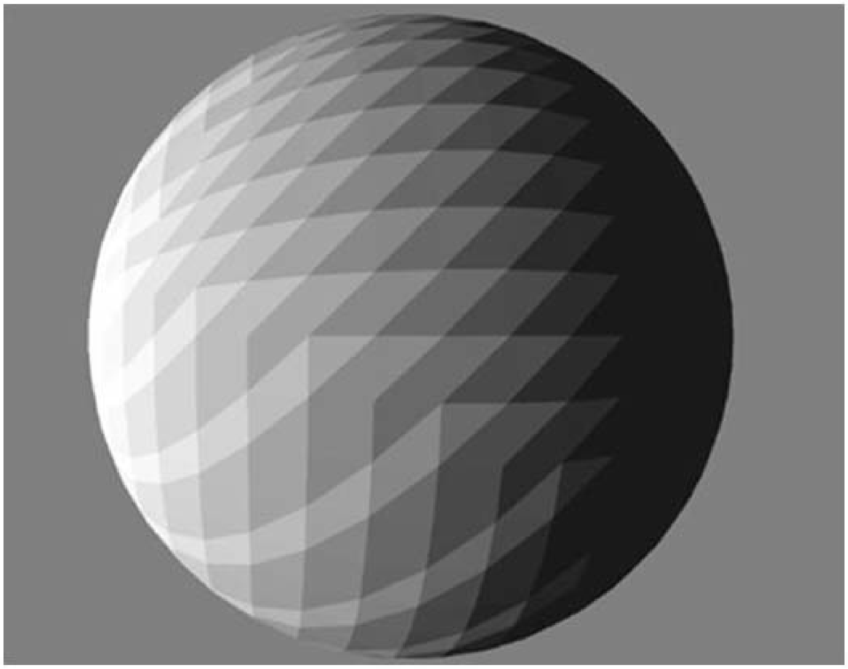


Figure 7.14 Flat-shaded lighting.

lighting/shading combination, owing to two facts: the expensive lighting equation need only be evaluated once per triangle, and a single color can be used for all pixels in the triangle. Figure 7.14 shows an example of a sphere lit and shaded using per-triangle lighting and flat shading.

To evaluate the lighting equation for a triangle, we need a sample location and surface normal. The surface normal used is generally the face normal (discussed in Chapter 1), as it accurately represents the plane of the triangle. However, the issue of sample position is more problematic. No single point can accurately represent the lighting across an entire triangle (except in special cases); for example, in the presence of a point light, different points on the triangle should be attenuated differently, according to their distance from the light. While the centroid of the triangle is a reasonable choice, the fact that it must be computed specifically for lighting makes it less desirable. For reasons of efficiency (and often to match with the graphics system, as will be discussed presently for OpenGL), the most common sample point for flat shading is one of the triangle vertices, as the vertices already exist in the desired space. This can lead to artifacts, since a triangle's vertices are (by definition) at the edge of the area of the triangle.

Flat-shaded Lighting in OpenGL

As with per-triangle coloring, in OpenGL per-triangle lighting is actually done quite simply. The final, lit color of one of the triangle's vertices is used directly as the color of the entire triangle. The OpenGL specification details which vertex is used in each mode, but for `GL_TRIANGLES` the vertex used is the last (third) vertex in the triangle. As a result, OpenGL does not have a notion of a polygon normal for lighting. The face normal must be associated with the final vertex that is used to generate the triangle. This can be problematic in the case of indexed geometry, where some vertices may have to be used as the third vertex for more than one triangle (it is very easy to generate indexed geometry that has more triangles than vertices). In such cases, it may be necessary to duplicate vertices in order to be able to specify triangle-specific normals. Recall that flat shading is enabled in OpenGL with the function call

```
glShadeModel(GL_FLAT);
```

7.8.2 Per-Vertex Lighting

Flat-shaded lighting suffers from the basic flaws and limitations of flat shading itself; the faceted appearance of the resulting geometry tends to highlight rather than hide the piecewise triangular approximation. In the presence of specular lighting, the tessellation is even more pronounced, causing entire triangles to be lit with bright highlights. With moving lights or geometry, this can cause gemstonelike "flashing" of the facets. For smooth surfaces such as the sphere in Figure 7.14 this faceting is often unacceptable.

The next logical step is to use vertex lighting with Gouraud shading. The lighting equation is evaluated per-vertex, and the results are interpolated across the triangles using Gouraud shading. Generating a single lit color that is shared by all co-located vertices leads to smooth lighting across surface boundaries. Even if co-located vertices are not shared (i.e., each triangle has its own copy of its three vertices), simply setting the normals to be the same in all copies of a vertex will cause all copies to be lit the same way. Figure 7.15 shows an example of a sphere lit and shaded using per-vertex lighting and Gouraud shading.

Per-vertex lighting only requires evaluating the lighting equation once per vertex. In the presence of well-optimized vertex sharing (where there are more triangles than vertices), per-vertex lighting requires fewer lighting equation evaluations than does flat shading. However, the shading interpolation method used (Gouraud) is more expensive computationally, since it must interpolate between the three vertex colors on a per-pixel basis.

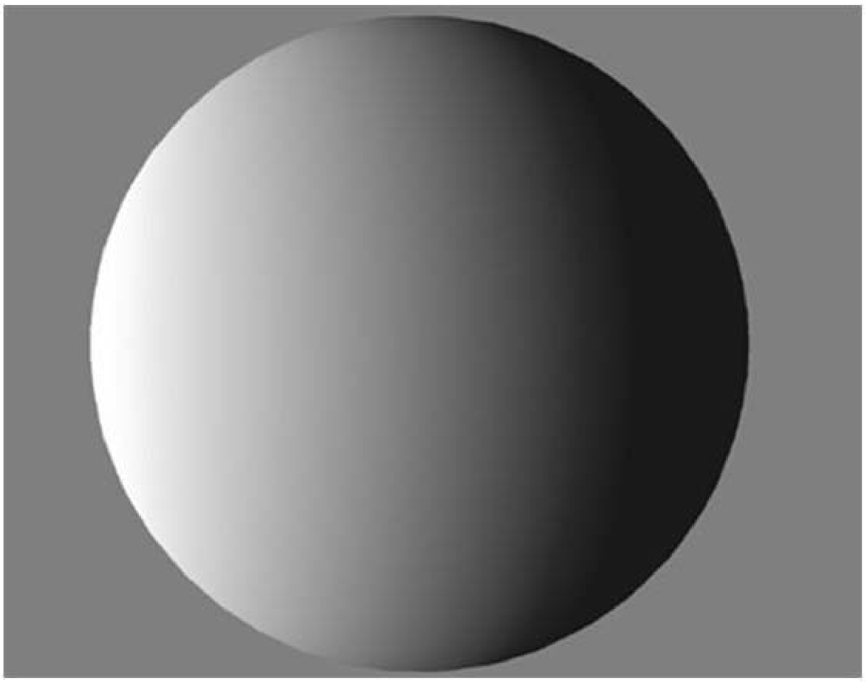


Figure 7.15 Gouraud-shaded lighting.

Per-vertex lighting is the standard in OpenGL, and Gouraud shading of the results is enabled via

```
glShadeModel(GL_SMOOTH);
```

Gouraud-shaded lighting is a vertex-centric method—the surface positions and normals are used only at the vertices, with the triangles serving only as areas for interpolation. This shift to vertices as localized surface representations means that we will need surface normals at each vertex. The next section will discuss several methods for generating these vertex normals.

Generating Vertex Normals

In order to generate smooth lighting that represents a surface at each vertex, we need to generate a single normal that represents the surface at each vertex, not at each triangle. There are several common methods used to generate these per-vertex surface normals at content creation time or at loadtime, depending upon the source of the geometry data.

When possible, the best way to generate smooth normals during the creation of a tessellation is to use analytically computed normals based on the surface being approximated by triangles. For example, if the set of triangles represent a sphere centered at the origin, then for any vertex at location P_V, the surface normal is simply

$$\hat{\mathbf{n}} = \frac{P_V - 0}{|P_V - 0|}$$

This is the vertex position, treated as a vector (thus the subtraction of the zero point) and normalized. Analytical normals can create very realistic impressions of the original surface, as the surface normals are pivotal to the overall lighting impression. Examples of surfaces for which analytical normals are available include most of the types of surface representations mentioned earlier in this chapter; implicit surfaces and parametric surface representations generally include analytically defined normal vectors at every point in their domain.

In the more common case the mesh of triangles exists by itself, with no available method of computing exact surface normals for the surface being approximated. In this case the normals must be generated from the triangles themselves. While this is unlikely to produce optimal results in all cases, simple methods can generate normals that tend to create the impression of a smooth surface and remove the appearance of faceting.

One of the most popular algorithms for generating normals from triangles takes the mean of all of the face normals for the triangles that use the given vertex. Figure 7.16 demonstrates a two-dimensional example of averaging triangle normal vectors. The algorithm may be pseudo-coded as follows:

```
for each vertex V
{
    vector V.N = (0,0,0);
    for each triangle T that uses V
    {
      vector F = TriangleNormal(T);
      V.N += F;
    }

    V.N.Normalize();
}
```

Basically, the algorithm sums the normals of all of the faces that are incident upon the current vertex and then renormalizes the resulting summed vector. Since this algorithm is (in a sense) a mean-based algorithm, it can

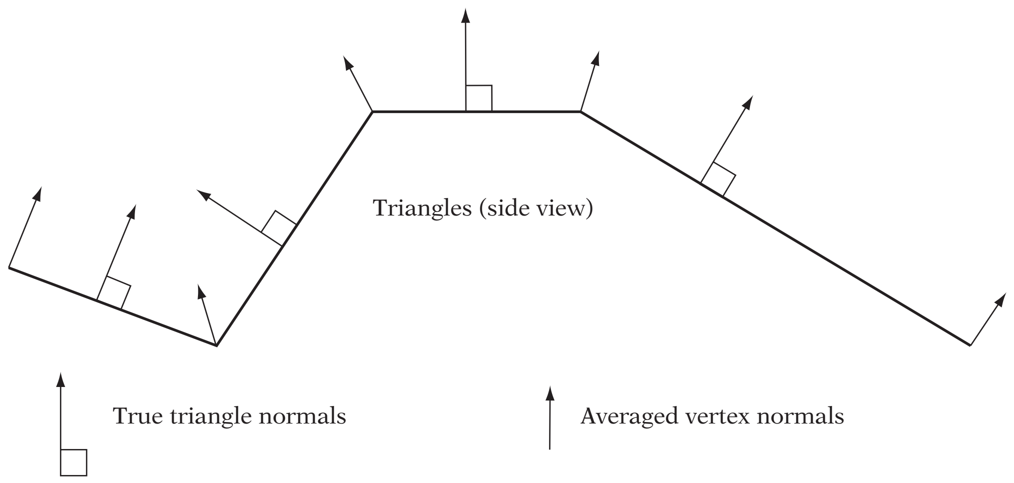


FIGURE 7.16 Averaging triangle normal vectors.

be affected by tessellation. Triangles are not weighted by area or other such factors, meaning that the face normal of each triangle incident upon the vertex has an equal "vote" in the makeup of the final vertex normal. While the method is far from perfect, any vertex normal generated from triangles will by its nature be an approximation. In most cases the averaging algorithm generates convincing normals. Note that in cases where there is no fast (i.e., constant-time) method of retrieving the set of triangles that use a given vertex (e.g., if only the OpenGL-style index lists are available), the algorithm may be turned "inside out" as follows:

```
for each vertex V
{
    V.N = (0,0,0);
}

for each triangle T
{
    // V1, V2, V3 are the vertices used by the triangle
    vector F = TriangleNormal(T);
    V1.N += F;
    V2.N += F;
    V3.N += F;
}
```

```
for each vertex V
{
    V.N.Normalize();
}
```

Basically, this version of the algorithm uses the vertex normals as "accumulators," looping over the triangles, adding each triangle's face normal to the vertex normals of the three vertices in that triangle. Finally, having accumulated the input from all triangles, the algorithm goes back and normalizes each final vertex normal. Both algorithms will result in the same vertex normals, but each works well with different vertex/triangle data structure organizations.

Sharp Edges

SOURCE CODE DEMO Edges

As with Gouraud shading based on fixed colors, Gouraud-shaded lighting generates smooth triangle boundaries by default. In order to represent a sharp edge, vertices along a physical crease in the geometry must be duplicated so that the vertices can represent the surface normals on either side of the crease. By having different surface normals in copies of co-located vertices, the triangles on either side of an edge can be lit according to the correct local surface orientation. For example, at each vertex of a cube, there will be three vertices, each one with a normal of a different face orientation as we see in Figure 7.17.

7.8.3 Per-Pixel Lighting (Phong Shading)

There are significant limitations to Gouraud shading. Specifically, the fact that the lighting equation is evaluated only at the vertices can lead to artifacts. Even a cursory evaluation of the lighting equation shows that it is highly nonlinear. However, Gouraud shading interpolates linearly across polygons. Any nonlinearities in the lighting across the interior of the triangle will be lost completely. These artifacts are not as noticeable with diffuse and ambient lighting as they are with specular lighting, because diffuse and ambient lighting are closer to linear functions than is specular lighting (owing at least partially to the nonlinearity of the specular exponent term and to the rapid changes in the specular halfway vector $\hat{\mathbf{h}}$ with changes in viewer location).

For example, let us examine the specular lighting term for the surface shown in Figure 7.18. We draw the two-dimensional case, in which the triangle is represented by a line segment. In this situation the vertex normals all point outward from the center of the triangle, meaning that the triangle is representing a somewhat domed surface. The point light source and the

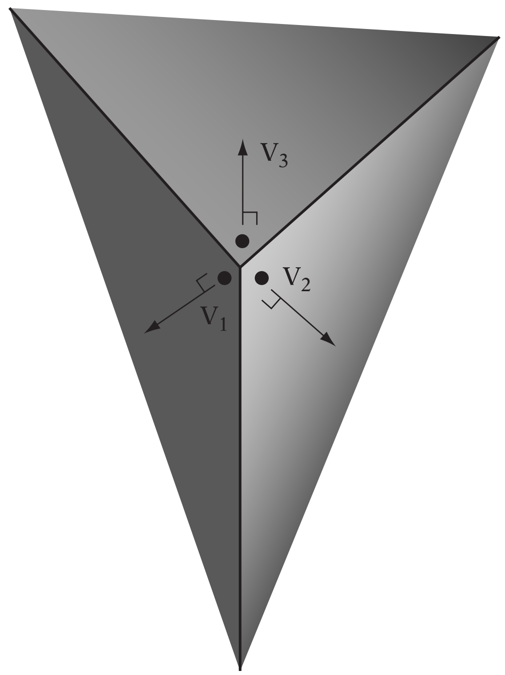


FIGURE 7.17 One corner of a faceted cube.

viewer are located at the same position in space, meaning that the view vector $\hat{\mathbf{v}}$, the light vector $\hat{\mathbf{L}}$, and the resulting halfway vector $\hat{\mathbf{h}}$ will all be equal for all points in space. The light and viewer are directly above the center of the triangle. Because of this, the specular components computed at the two vertices will be quite dark (note the specular halfway vectors shown in Figure 7.18 are almost perpendicular to the normals at the vertices). Linearly interpolating between these two dark specular vertex colors will result in a polygon that is relatively dark.

However, if we look at the geometry that is being approximated by these normals (a domed surface as in Figure 7.19), we can see that in this configuration the interpolated normal at the center of the triangle would point straight up at the viewer and light. If we were to evaluate the lighting equation at a point near the center of the triangle in this case, we would find an extremely bright specular highlight there. The specular lighting across the surface of this triangle is highly nonlinear, and the maximum is internal to the triangle. Even more problematic is the case in which the surface is moving over time. In rendered images where the highlight happens to line up with a vertex, there will be a bright, linearly interpolated highlight at the vertex. However, as the surface moves so that the highlight falls between

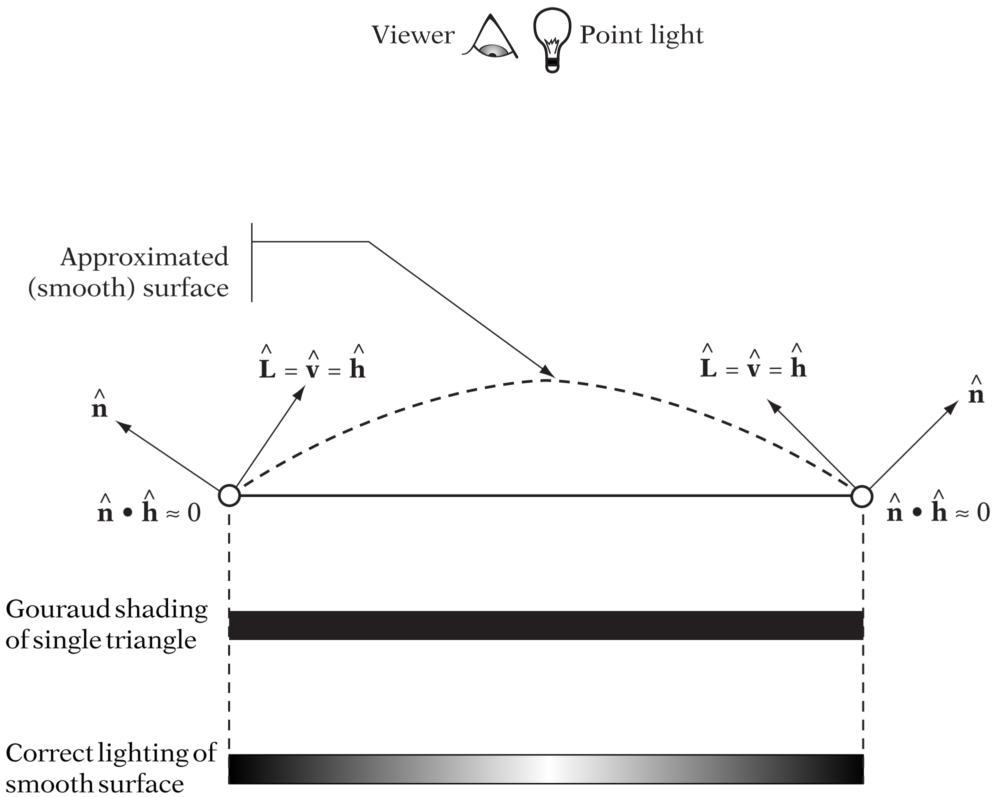

Viewer
Point light
Approximated (smooth) surface
$\hat{\mathbf{L}} = \hat{\mathbf{v}} = \hat{\mathbf{h}}$
$\hat{\mathbf{n}}$
$\hat{\mathbf{L}} = \hat{\mathbf{v}} = \hat{\mathbf{h}}$
$\hat{\mathbf{n}}$
$\hat{\mathbf{n}} \bullet \hat{\mathbf{h}} \approx 0$
$\hat{\mathbf{n}} \bullet \hat{\mathbf{h}} \approx 0$
Gouraud shading of single triangle
Correct lighting of smooth surface

FIGURE 7.18 Gouraud shading can miss specular highlights.

vertices, the highlight will disappear completely. This is a very fundamental problem with approximating a complex function with a piecewise-linear representation. The accuracy of the result is dependent upon the number of linear segments used to approximate the function. In our case this is equivalent to the density of the tessellation.

If we want to increase the accuracy of lighting on a general Gouraud-shaded surface, we must subdivide the surface to increase the density of vertices (and thus lighting samples). However, this is an expensive process, and we may not know a priori which sections of the surface will require significant tessellation. Dependent upon the particular view at runtime, almost any tessellation may be either overly dense or too coarse. In order to create a more general, high-quality lighting method, we must find another way around this problem.

So far, the methods we have discussed for lighting have all evaluated the lighting equation once per basic geometric object, such as per-vertex or per-triangle. *Phong shading* (named after its inventor, Phong Bui-Tuong [89])

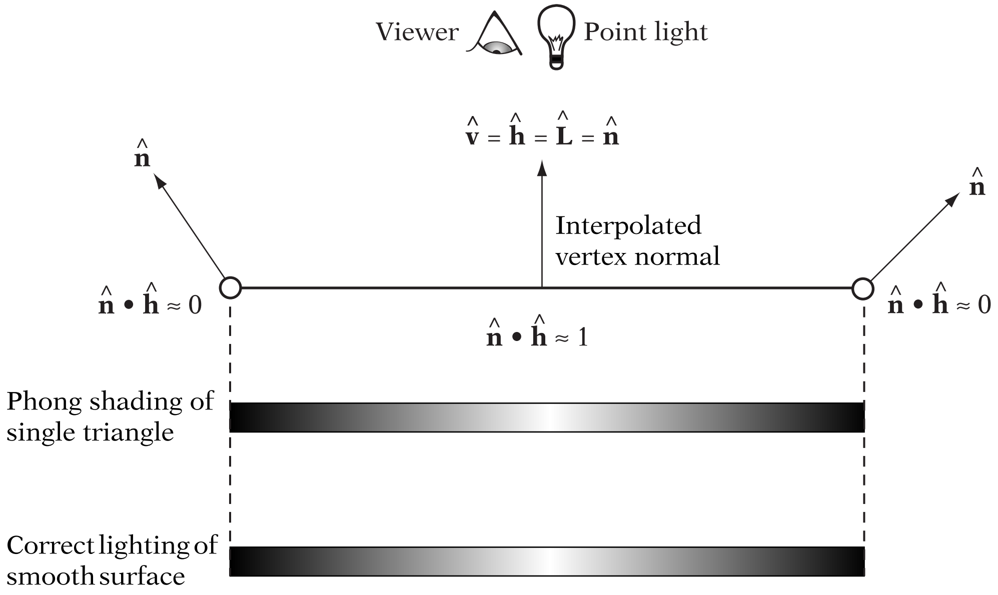

Viewer
Point light
$\hat{\mathbf{v}} = \hat{\mathbf{h}} = \hat{\mathbf{L}} = \hat{\mathbf{n}}$
$\hat{\mathbf{n}}$
$\hat{\mathbf{n}}$
Interpolated vertex normal
$\hat{\mathbf{n}} \bullet \hat{\mathbf{h}} \approx 0$
$\hat{\mathbf{n}} \bullet \hat{\mathbf{h}} \approx 1$
$\hat{\mathbf{n}} \bullet \hat{\mathbf{h}} \approx 0$
Phong shading of single triangle
Correct lighting of smooth surface

FIGURE 7.19 Phong shading of the same configuration.

works by evaluating the lighting equation once for each pixel covered by the triangle. The difference between Gouraud and Phong shading may be seen in Figures 7.18 and 7.19. For each sample across the surface of a triangle, the vertex normals, positions, reflection, and view vectors are interpolated, and the interpolated values are used to evaluate the lighting equation. However, since triangles tend to cover more than 1–3 pixels, such a lighting method will result in far more lighting computations per triangle than do per-triangle or per-vertex methods.

There are several issues that make Phong shading expensive to implement in a high-performance, real-time system. The first of these is the actual normal vector interpolation, since basic barycentric interpolation of the three vertex normals will almost never result in a normalized vector. As a result, the normal vector will have to be interpolated and renormalized *per sample*, which is much more frequent than per vertex.

Per sample, once the interpolated normal is computed and renormalized, the full lighting equation must be evaluated. Not only is this operation expensive, it is not a fixed amount of computation. The complexity of the lighting equation is dependent on the number of lights and numerous graphics engine settings. This resulted in Phong shading being rather unpopular in game-centric consumer 3D hardware prior to the advent of pixel and vertex shaders. There is no standard method in OpenGL to enable Phong shading, although an implementation of OpenGL could implement it and expose it via an extension. It should be noted that with the availability of pixel shader hardware (as discussed in Chapter 6), it is possible to implement per-pixel lighting methods, including Phong shading and methods based upon per-pixel interpolation of normals and lighting evaluation.

7.9 Merging Textures and Lighting

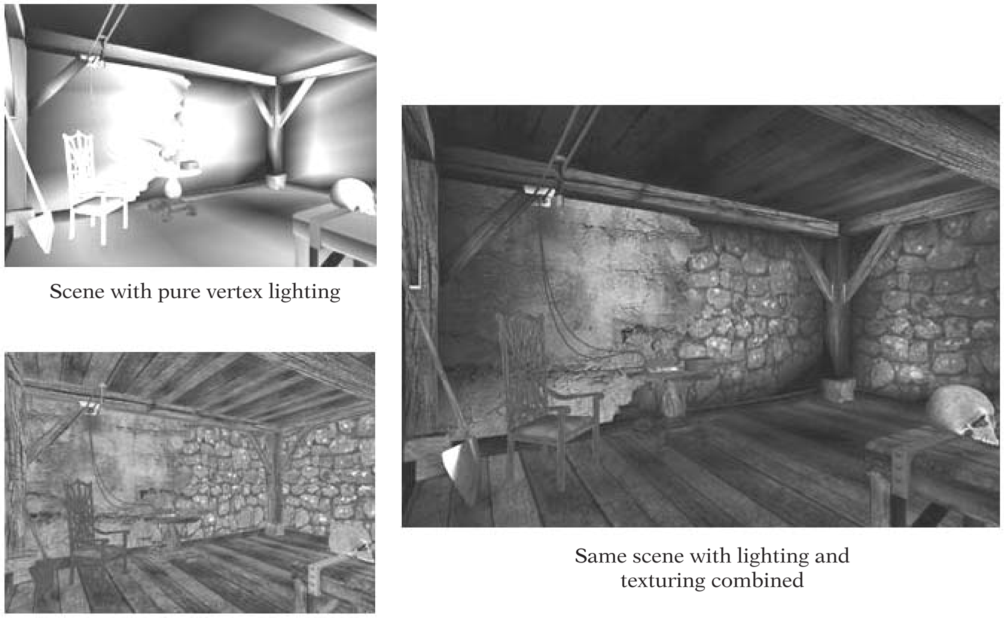
Source Code Demo Textures

Of the methods we have discussed for coloring geometry, the two most powerful are texturing and dynamic lighting. However, they each have drawbacks when used by themselves. Texturing is normally a static method and looks flat and painted when used by itself in a dynamic scene. Lighting can generate very dynamic effects, but it is limited to face- or vertex-level detail unless special pixel shaders are used. It is only natural that graphics systems would want to use the results of both techniques together on a single surface. This is possible, but the issue of how to combine the two methods must be addressed.

Each of the two methods is capable of generating a color per-sample. With texturing, this is done directly via texture sampling; with lighting, it is done by interpolating values computed at each vertex (generally using one of the shading methods discussed in Chapter 6). These two colors must be combined in a way that makes visual sense. The method of combining textures with face or vertex colors is called the *texture application mode*. The most common way of combining textures and vertex colors is via multiplication, also known as *modulate mode texturing*. In modulate texture combination, the texture color at the given sample C_T and the final (generally lit) interpolated vertex color C_V are combined by per-component multiplication:

$$C = C_T C_V$$
$$A = A_T A_V$$

The visual effect here is that the vertex colors darken the texture (or vice versa). As a result, texture images designed to be used with modulate mode texture combination are normally painted as if they were fully lit. The vertex colors, representing the lighting in the scene, darken these fully lit textures to make them look more realistic in the given environment. As Figure 7.20 demonstrates, the result of modulation can be very convincing, even though the lighting is rather simple and the textures are static paintings. In the presence of moving or otherwise animated lights, the result can be even more immersive, as the human perceptual system is very reliant upon lighting cues in the real world.



Scene with pure vertex lighting

Same scene with lighting and texturing combined

Scene with pure texturing

FIGURE 7.20 Textures and lighting combined via modulation.

By default, OpenGL uses modulate mode for combining textures and vertex colors. However, it does support other modes via

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, …);
```

including `GL_REPLACE` (which ignores vertex colors) and `GL_DECAL` (an alpha-blended mode that applies the texture as a transparent "decal" to the surface), among others. You may wish to refer to the OpenGL Programming Manual [83] for details.

7.9.1 Specular Lighting and Textures

If the full lighting equation 7.1 is combined with the texture via multiplication, then lighting can only *darken* the texture, since lit vertex colors C_V are clamped to the range [0, 1]. While this looks correct for diffuse or matte objects, for shiny objects with bright specular highlights, it can look very dull. It is often useful to have the specular highlights "wash out" the texture. We cannot simply add the full set of lighting because the texture will almost always wash out and can *never* get darker. To be able to see the full range of effects requires that the diffuse colors darken the texture while the specular components of color add highlights. This is only possible if we split the lighting components.

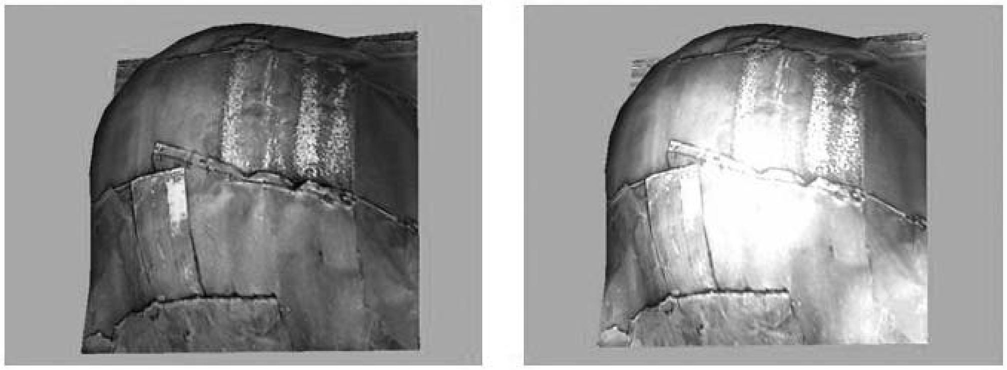
OpenGL includes a mode that allows this. The mode is enabled using

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

and it works by "splitting" the results of the lighting model equation 7.1 into two pieces. The first piece, C_D, contains the emissive, ambient, and diffuse terms of the color summation. The second piece, C_S, contains the specular terms. The two colors are combined with the texture color as follows:

$$C = C_T C_D + C_S$$

Because the specular term is added after the texture is multiplied, this mode (sometimes called *modulate with late add*) causes the diffuse terms to attenuate the texture color, while the specular terms wash out the result. The differences between the separate and combined specular modes can be very striking as Figure 7.21 makes clear. Unfortunately, the default mode in OpenGL is to disable this feature and use combined diffuse and



Specular vertex color added to diffuse vertex color, then modulated with the texture

Diffuse vertex color modulated with the texture, then specular vertex color added

FIGURE 7.21 Combining textures and lighting.

specular colors. Once separate specular colors is enabled, it can be disabled with

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
```

However, this "late add" effect must be supported in the rasterizer level, as it requires two colors to be interpolated per-pixel rather than one (i.e., the specular and the combined ambient-diffuse-emissive). Some graphics hardware emulates this effect using a simpler trick: allowing only a single component (i.e., white) specular value C_S that is applied as a late add. In fact, hardware implementing this trick often uses the vertex color's alpha channel to hold this specular value, meaning that the feature is mutually exclusive with respect to per-vertex alpha blending. This limitation has caused the popularity of the alpha-channel specular trick to wane in current consumer 3D hardware.

7.10 Lighting and Programmable Shaders

Today, procedural shading using vertex and pixel shaders is rapidly gaining popularity, requiring application developers (in many cases) to leave behind existing lighting pipelines, such as those supplied in an OpenGL implementation, and write their own. However, while the exact methods of enabling, disabling, and controlling lighting differ between a "fixed-function" lighting pipeline and hard-coded, shader-based lighting pipelines, all of the concepts and formulas given in this chapter may be used in the creation of lighting-based shaders, as well. In fact, to effectively use shaders, a developer must have a true understanding of the concepts behind dynamic lighting, in order to know which parts of these equations they must add as code to their shaders and which they can ignore.

Even with the growing power of vertex and pixel shader hardware, developers must be able to actively trade off parts of the lighting pipeline if they are to fit all of their desired effects into their "performance budget." A full understanding of the components of the lighting pipeline, as well as the way they fit together into the overall lighting equation, is an important part of this challenge. Interested readers should investigate any of the multitude of shader tutorials available on the Internet at 3D hardware developers' sites, as well as in books such as [33].

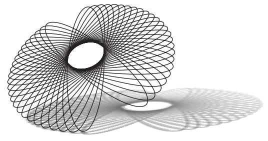
7.11 Chapter Summary

In this chapter we have discussed the basics of dynamic lighting, both in terms of geometric concepts and implementation in OpenGL's standard pipeline.

Per-vertex (and in some cases today, per-pixel) lighting is a very powerful addition to any 3D application. Correct use of lighting can create compelling 3D environments at limited computational expense. As we have discussed, judicious use of lighting is important in order to maximize visual impact while minimizing additional computation.

For further information, there are numerous paths available to the interested reader. More and more, developers are leaving behind the inflexible lighting pipelines that exist in DirectX and OpenGL and are writing their own via vertex and pixel shaders. The growing wealth of shader resources includes web sites ([6], [82]) and even book series [33]. Many of these new shaders are based on far more detailed and complex lighting models, such as those presented in computer graphics conference papers and journal articles like those of ACM SIGGRAPH or in books such as [113].

Chapter 8 Rasterization



8.1 Introduction

The final stage in the rendering pipeline is called *rasterization*. Rasterization is the operation that takes screen-space geometry, a shading method such as those described in the previous chapters, and the inputs to those shading methods and actually draws the geometry to the low-level 2D display device. Once again, we will focus on drawing sets of triangles, as these are the most common primitive in 3D graphics systems. In fact, for much of this chapter, we will focus on drawing an individual triangle. For almost all modern display devices, this low-level "drawing" operation involves assigning color values to each and every dot, or *pixel*, on the display device.

At the conceptual level, the entire topic of rasterization is simply an "implementation detail." Rasterization is required because the display devices we use today are based on a dense rectangular grid of light-emitting elements, or pixels (a short version of "picture elements"), each of whose colors and intensities are individually adjustable in every frame.

Earlier displays (used prior to the mid-1970s) were not based on these grids of pixels, but were instead capable of drawing only lines or curves between points on the screen. Unlike the discrete grid of addressable points on a raster display, the entire surface of a so-called *vector display screen* is addressable continuously. The screen-space positions of each line's endpoints were fed to the display system, and it drew the line by directly tracing the path between the points onto the screen. These vector displays were very much like the screens on an engineer's oscilloscope (in fact, many of the early ones *were* oscilloscopes). An analogous, more modern example is the popular "laser show" seen at planetariums and live concert venues. Figure 8.1 is a basic drawing of how such vector displays worked.

These vector displays required no rasterization, as lines and curves could be drawn by directly tracing them onto the display. However, while vector displays could render perfectly smooth and sharp lines between any pair of vertices (and thus the outlines of objects), they were also limited to drawing wireframe geometry. Furthermore, they could not generally "fill" areas of the screen with light and were (for the most part) unable to display more than grayscale light or a few selected colors. Basically, they were not capable of drawing scenes with any photorealism. Examples of common vector displays include some early video games, specifically *Asteroids*, *Tempest*, and *Battlezone* (all by Atari, the latter two including rudimentary color). Another (somewhat different) example of a vectorlike display is the pen-plotter, which draws by moving a set of colored pens across the surface of a sheet of paper.

The limitations of vector displays led to a move in the mid-1970s toward using televisionlike *raster* displays, with their accompanying grids of individually colored pixels. Decades of television images (both monochrome and color) had proven that raster displays were very flexible and could support areas of color, complex images, and a full range of realistic color. However, raster displays required that the images displayed on them be discretized into a rectangular grid of color samples for each image. In order to achieve this, a computer graphics system must convert the projected, colored geometry representations into the required grid of colors. Moreover, in order to render

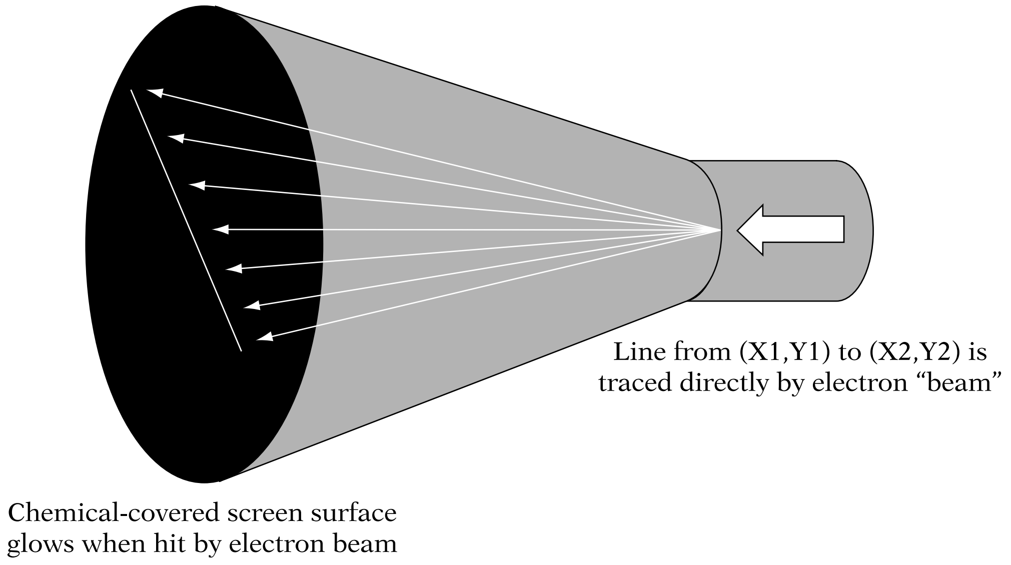

Line from (X1,Y1) to (X2,Y2) is traced directly by electron "beam"
Chemical-covered screen surface glows when hit by electron beam

FIGURE 8.1 Vector display hardware.

real-time animation, the computer graphics system must do so many times per second. This process of generating a grid of color samples from a projected scene is called *rasterization*.

By its very nature, rasterization is time-consuming when compared to the other stages in the rendering pipeline. Whereas the other stages of the pipeline generally require per-object, per-triangle, or per-vertex computation, rasterization inherently requires computation of some sort for every pixel. As of the early 2000s, displays 1600 pixels wide by 1200 pixels high — resulting in approximately *2 million* pixels on the screen — are popular. Add to this the fact that rasterization will in practice often require each pixel to be computed several times, and we come to the realization that the number of pixels that must be computed generally outpaces the number of triangles in a given frame by a factor of 10, 20, or more.

In fact, in purely software 3D pipelines, it is not uncommon to see as much as 80 to 90 percent of rendering time spent in rasterization. This level of computational demand has led to the fact that rasterization was the first stage of the graphics pipeline to be accelerated via purpose-built consumer hardware. In fact, most 3D computer games began to *require* some form of 3D hardware by the early 2000s. This chapter will not detail the methods and code required to write a software 3D rasterizer, since most game developers no longer have a need to write them. However, complete software pipelines including rasterization are still seen in low-power and low-cost devices, such as handheld computers and cellular telephones. Also, so-called mass market 3D games, which are designed to run on older computers will sometimes include a software rasterizer, generally rendering the game at a decreased frame rate or reduced visual quality. While we will not discuss the implementation details of software rasterizers, many of the high-level concepts required to create them *will* be covered in this chapter. For the details on how to write a set of rasterizers, see Hecker's excellent series of articles on perspective texture mapping in Game Developer Magazine [60].

8.2 Displays and Framebuffers

Every piece of display device hardware, whether it be a computer monitor, a television, or some other such device requires a source of image data. For computer graphics systems, this source of image data is called a *framebuffer* (so called because it is a buffer of data that holds the image information for a "frame," or a screen's worth of image). In basic terms, a framebuffer is a two-dimensional digital image: a block of memory that contains numerical values that represent colors at each point on the screen. Each color value represents the color of the screen at a given point — a picture element, or *pixel*. Each pixel has red, green, and blue components. Put together, this framebuffer represents the image that is to be drawn on the screen. The display hardware reads these colors from memory every time it needs to update the image on the screen, generally at least 30 times per second and often 60 or more times per second.

As we shall see, framebuffers often include more than just a color per pixel. While it is the per-pixel color that is actually used to set the color and intensity of light emitted by each point on the display, the other per-pixel values are used internally during the rasterization process. In a sense, these other values are analogous to per-vertex normals and per-triangle material colors; while they are never displayed directly, they have a significant effect on how the final color is computed.

8.2.1 Framebuffer Memory Organization

Cathode ray tube (CRT) displays, such as televisions and monitors, work by redrawing the screen from left to right, top to bottom, pixel by pixel (Figure 8.2). In order to set the color of each pixel, the display must be supplied

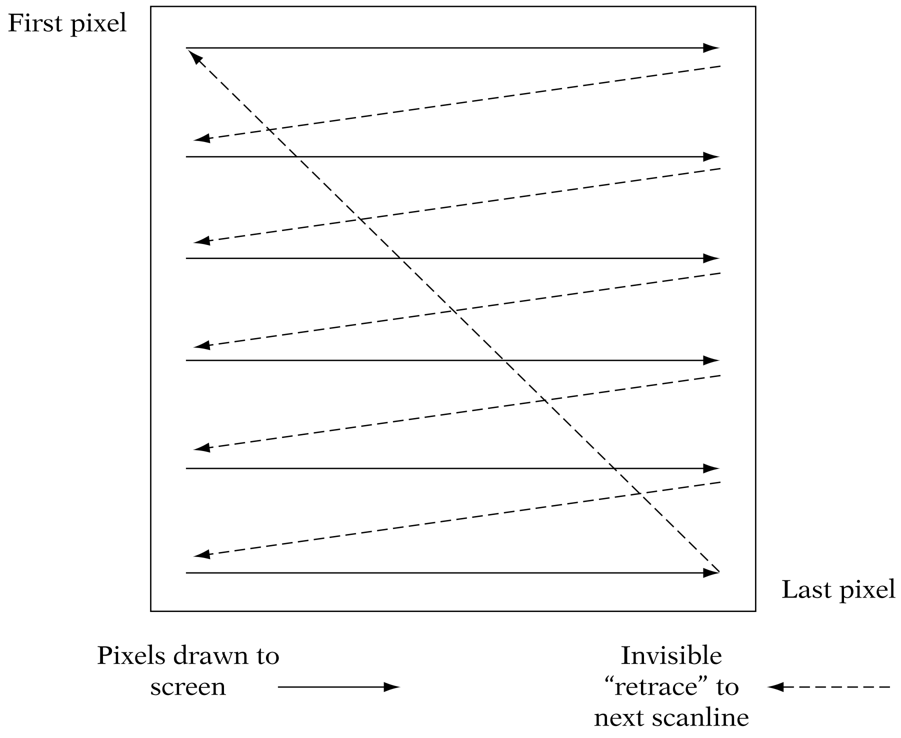


Figure 8.2 CRT redraw pattern.

with the correct color as it is needed during the redrawing process. In the case of televisions, this color information is supplied to the display device directly from the video source (a cable TV tuner, videotape player, DVD player, etc.) at the exact moment it is needed. As a result, most televisions do not have framebuffers — they display the data as it is supplied.

With computer displays, the device supplying the colors as needed is the framebuffer memory. The display system must read the color of each required pixel from the framebuffer memory when it is needed. In order to feed this scanning process with pixel data most efficiently, framebuffers are generally arranged such that the pixels are stored in the order they are scanned out to the screen. This is a row-major order, meaning that all pixels in each horizontal line on the display are stored together, in order of increasing x coordinate. These lines of pixels are called *scanlines* in the framebuffer, as they represent a single left-to-right pass (or "scan") across the screen. Each scanline in the framebuffer is followed by the next lower scanline until the bottom-right corner of the screen is reached (the memory layout matches the pixel-scanning sequence, which is shown in Figure 8.2). As mentioned in Chapter 5, the positive y dimension of the screen is downward to match the scanning order.

This organization of framebuffer memory means that we will (as often as possible) be drawing a given piece of geometry (normally a triangle) in scanline-by-scanline order, thus limiting the need to jump around rather randomly in the framebuffer. Such a method can reduce memory bandwidth to the framebuffer. As we shall see, it can also be an efficient way of computing per-pixel triangle colors.

8.2.2 Interlacing

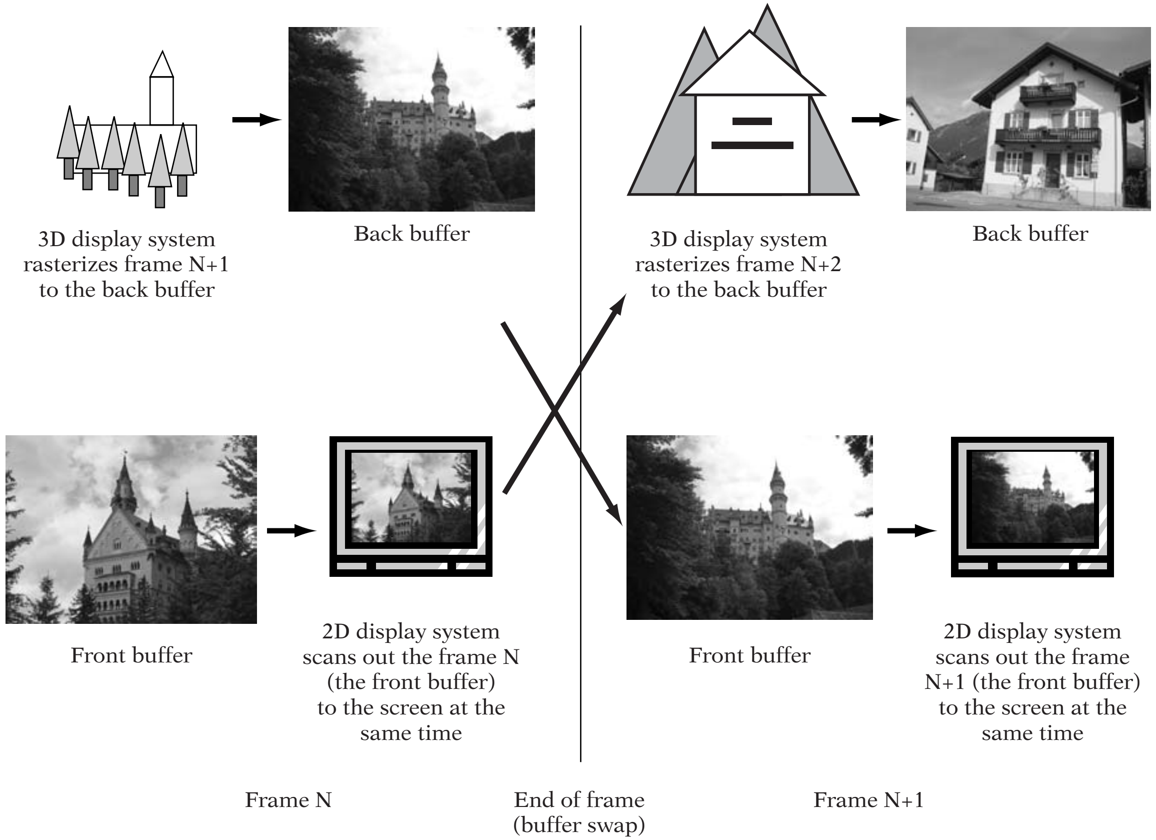
Note that most television systems actually use a slightly different scanning method, known as *interlacing*. Interlacing draws the *even* scanlines in top-to-bottom order, and then goes back and draws the *odd* scanlines in top-to-bottom order. Each of these sets of lines is known as a *field*, an even field and an odd field per frame. A television redraws one field every 60th of a second. In redrawing this way, a television *appears* to be refreshing the screen every 60th of a second when, actually, it is only refreshing half of the lines every 60th of a second. The entire screen is redrawn only every two passes, or every 30th of a second. However, the fact that the even and odd sets of lines "cover the screen" means that, effectively, a low-resolution version of the entire screen is drawn every 60th of a second. This trick reduces the amount of information that needs to be transferred from the source per second to draw television images (originally, this reduced the required radio bandwidth of television signals). However, interlacing causes thin horizontal lines (which only get redrawn every 30th of a second since they are only a part of one field in each frame) to flicker. This makes interlacing inappropriate for computer screens (although early home computers often used owners' existing interlaced televisions as their monitors to reduce costs). Because they feed televisions as their display devices, video game consoles must deal with interlacing, a fact that some architectures will expose at the framebuffer level.

8.2.3 Multiple Buffers

It is common to have two full-sized blocks (or "buffers") of framebuffer memory in a display system. At any given time, one of these copies is being read by the display hardware to update the display device itself, while the other is being written by the 3D graphics system. At the instant between the end of reading the data of one frame out to the display device and starting to read the next, the two buffers can be "swapped." This swapping allows the buffer that was just written (drawn) by the 3D system to be read out to the display device, while making the previous frame's buffer available to the 3D system to prepare as the next frame. Figure 8.3 shows this process schematically.

This system is known as *double buffering* because it involves two complete screen-sized images. At any given time, one buffer (the "front buffer") is being read pixel by pixel onto the display device by the 2D display system, while the other (the "back buffer") is being written to by the 3D graphics system with the next frame. Once the next frame is drawn to the back buffer, the next time the front buffer is finished being read out to the display (generally during the moment that the display is resetting itself for the next pass), the two buffers are swapped. The back buffer becomes the new front buffer and the front buffer that has just been read onto the screen becomes the new back buffer, ready to be redrawn with the next frame. Note that in most cases this "swapping" operation does *not* involve copying or moving the data in the buffers. It simply involves swapping the two *pointers* that point to the front and back buffers. On most display devices, this is a single instruction in the hardware. As a result, the swap operation is extremely fast.

Double buffering is a significant performance optimization, as it allows parallelism between the 3D rendering and the 2D display system. While one buffer (the current front buffer) is being read out to the screen, the 3D hardware can simultaneously write the scene to the other buffer (the current back buffer). Systems that cannot support fast buffer swapping can still render and display in parallel, but rather than swapping the two pointers quickly between frames, the back buffer's contents must be copied to the front buffer. This involves moving a lot of data from one memory block to another, often causing memory bus performance issues. As a result, double buffering with buffer swapping is extremely popular in modern display hardware.



3D display system rasterizes frame N+1 to the back buffer
Back buffer
3D display system rasterizes frame N+2 to the back buffer
Back buffer
Front buffer
2D display system scans out the frame N (the front buffer) to the screen at the same time
Front buffer
2D display system scans out the frame N+1 (the front buffer) to the screen at the same time
Frame N
End of frame (buffer swap)
Frame N+1

Figure 8.3 Double buffering.

8.3 Conceptual Rasterization Pipeline

Conceptually, there are several stages to even a simple rasterization pipeline. It should be noted that while these stages tend to exist in rasterization hardware implementations, hardware almost never follows the order (or even the structure) of the conceptual stages in the list that follows. This simple pipeline rasterizes a single triangle as follows:

1. Determine the visible pixels covered by the triangle.
2. Compute a color for the triangle at each such pixel.
3. Determine a final color for each pixel and write to the framebuffer.

The first stage further decomposes into two separate steps: (1) determining the pixels covered by a triangle and (2) determining which of those pixels are visible. The rest of this chapter will discuss each of these pipeline stages in detail.

8.4 Determining the Pixels Contained by a Triangle

Triangles are convex, no matter how they are projected (in some cases, triangles may appear as a line or a point, but these are still convex objects). This is a very useful property, because it means that any triangle intersects a scanline in at most one contiguous segment. Thus, for any scanline that intersects a triangle, we can represent the intersection with a single "span," a minimum x value and a maximum x value. Thus, the representation of a triangle during rasterization consists of a set of spans, one per scanline that the triangle intersects. Furthermore, the convexity of triangles also implies that the set of scanlines intersected by a triangle is contiguous in y; there is a minimum and maximum y for a given triangle, which contains all of the nonempty spans. An example of the set of spans for a triangle is shown in Figure 8.4. The dark bands overlaid on the triangle represent the pixel spans that will be used to draw the triangle.

The minimum y pixel coordinate for a triangle y_{min} is simply the minimum y value of the three triangle vertices. Similarly, the maximum y pixel coordinate y_{max} of the triangle is simply the maximum y value of the three vertices. Thus, a simple min/max computation among the three vertices defines the entire range of $(y_{max} - y_{min} + 1)$ spans that must be generated for a triangle.

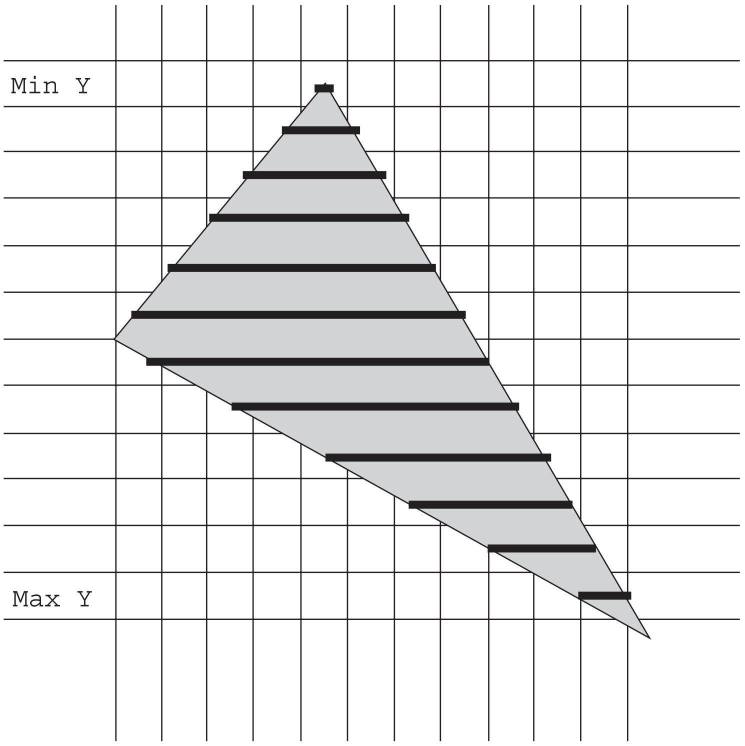


FIGURE 8.4 A triangle and its raster spans.

The behavior of a graphics system when a triangle vertex or edge falls *exactly* on a pixel center is determined by a system-dependent *fill convention*, which ensures that if two triangles share a vertex or an edge, only one triangle will draw to the pixel. This is very important, as without a well-defined fill convention, there may be "holes" (dropouts), or double-drawn pixels on the shared edges between triangles. Holes along a shared triangle edge allow the background color to show through what would otherwise be a continuous, opaque surface, making the surface appear to be "cracked." Double-drawn pixels along a shared edge result in more subtle artifacts, normally seen only when transparency or other forms of blending are used (see section 8.8 on pixel blending later in this chapter). For details on implementing fill conventions, see Hecker's *Game Developer* article series [60].

Generating the spans themselves simply involves intersecting the horizontal scanline with the edges of the triangle. Owing to the convexity of the triangle, unless the scanline intersects a vertex, that scanline will intersect exactly two of the edges of the triangle (one to cross from outside the triangle into it, and one to leave again). These two intersection points will define the minimum and maximum x values of the span.

Not all rasterizers generate a table of all spans in a triangle explicitly. In fact, the most common method is simply to start at the top of the triangle, computing the extents of the first span. Having generated the first span, all of the pixels in that span are completely rasterized. The system then generates the next span and rasterizes it completely and so on until all spans in the triangle are rasterized. This has the benefit of not having to store a table of spans, which could (in theory) require as many span entries as there are scanlines on the screen. Only the information for the current span need be stored. In fact, in purpose-built hardware, the next span information can even be computed by one piece of hardware while another piece of hardware rasterizes the current span, increasing performance via parallelism.

8.5 Determining Which Pixels are Visible

The overall goal in rendering geometry is to ensure that the final, rendered images convincingly represent the given scene. At the highest level, this means that objects must appear to be correctly obscured by closer objects and must not be obscured by more distant objects. This process is known as *visible surface determination*, and there are numerous, very different ways of accomplishing it. The methods all involve comparing the depth of surfaces at one level of granularity or another and rendering in such a way that the object of minimum depth (i.e. the closest object) at a given pixel is the one rendered to the screen.

8.5.1 Depth Sorting

One of the oldest visible surface determination algorithms predates computer graphics significantly and is called the *painter's algorithm*. It works by simulating a somewhat idealized version of the method used by artists when painting a scene. The painter starts by painting the background, then moves to painting closer and closer objects, often painting over parts of more distant objects that were already painted onto the canvas.

The computer graphics version of the painter's algorithm works by sorting all of the triangles in back-to-front (far to near) order, and drawing them in that order. This method is actually a geometric method rather than a rasterization method — triangles can be sorted at any time in the pipeline, as long as some notion of view direction is known, in order to assign camera-relative depth to every vertex. Most frequently, depth sorting is done after the view-space transform (either before or after the perspective division), since this stage generates camera-relative depth as a side-effect. Each triangle drawn will overwrite the more distant triangles that have already been drawn. At first glance, it would seem that when all triangles are drawn, the entire scene will be correctly displayed.

However, triangle sorting has several major problems. First, it is potentially very slow. Second, for some scenes, correct ordering of the given triangles may not be possible. We will briefly discuss some of the issues, but a more detailed review may be found in [36]. The first issue is one of performance. Sorting all of the triangles in a scene against one another is an expensive process. While a smart application can often decrease this expense by sorting large groups of triangles as a unit (say, all of the triangles in a single object) and then sorting the smaller groups among themselves (often called a "divide and conquer" method), this is not a fully general optimization.

Also, real-world painters do not (generally) paint a cityscape by first painting all of the people in all of the offices in an office building and then painting over them with the building's walls! This would be an immense waste of time and paint. However, the most basic form of the computer graphics painter's algorithm does just that. It draws all of the triangles in the scene, often drawing over the same section of the screen several times. This is known as *overdraw*, and it is a waste of computation that can lead (even on high-performance 3D hardware) to decreased performance. Avoiding this overdraw can be difficult and scene-dependent (see Zhang [121] for an example of an overdraw reduction method). An excellent overview of many depth-complexity reduction methods may be found in Chapter 12 of [27].

The larger issue with the triangle-level painter's algorithm is that there are many situations in which it is either difficult to compute a correct ordering of triangles or may even be impossible. The most important part of any sorting method (in terms of correctness) is determining the metric by which we sort the objects. While the concept of depth seems a simple metric, implementing it for triangles can be very tricky. Most triangles do not have all three vertices at the same depth—different parts of the triangle are at different depths. No single depth value can adequately represent an entire triangle. Figure 8.5 is an example of such a case. Each of the triangles (seen in side view) could (when represented by a single depth value) be considered "in front." It is only when they are compared pairwise to each other that we can compute which one of the pair is in front. Even then, a general method for doing so is complex.

Worse yet, some cases simply cannot be sorted. In Figure 8.6, we see such a case. Unless we split one of these triangles, no sorting method can draw these four triangles correctly. In fact, the most popular triangle sorting method requires that the scene be static and adds a preprocessing step to split triangles that could cause such sorting issues. The method is known as a BSP tree and is described in [38]. Basically, it involves creating a binary "decision tree" (once for a nonmoving scene), which allows the triangles to be sorted from back-to-front by testing the camera location versus a 3D plane at each node. The geometry resides in the leaves of the tree, and rendering is done by traversing the tree, following the left and right child of each node

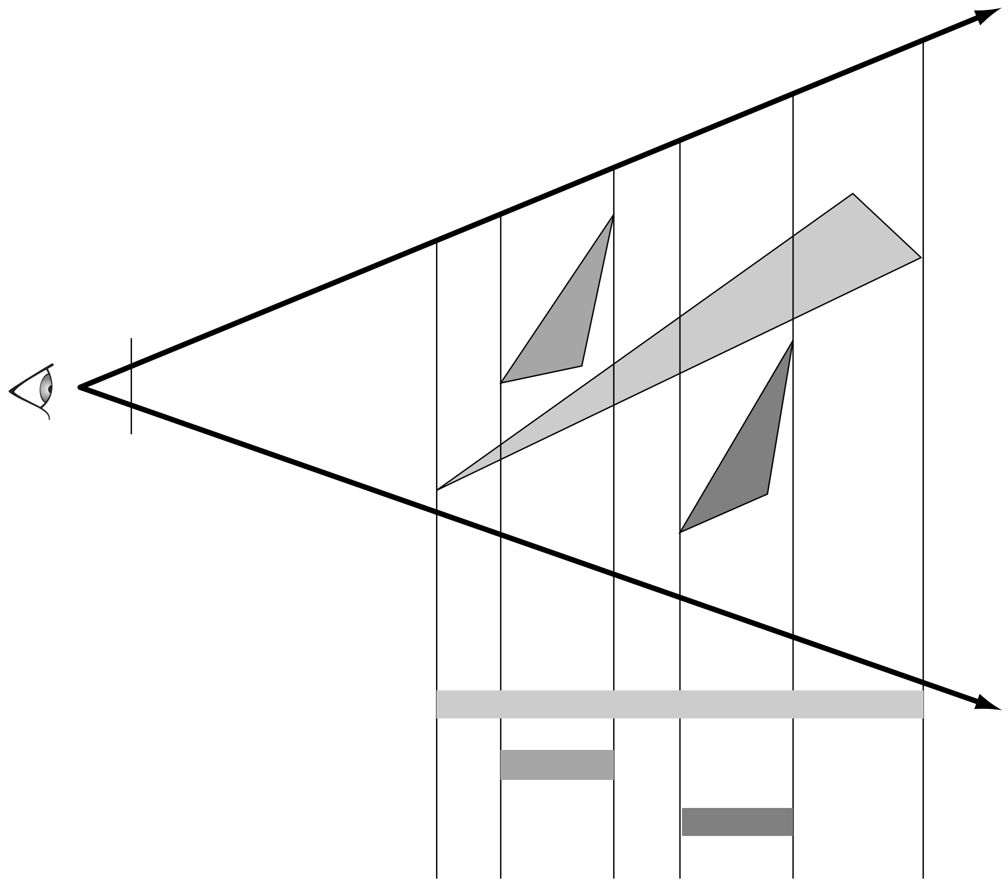


Figure 8.5 Triangles that overlap in depth (side view).

in one order or the other based on the camera location versus plane test. This method was quite popular in so-called first-person shooter games in the mid-to-late 1990s.

Depth sorting is an input-focused method. It ensures that the geometry going into the rasterization process is supplied in an order that will generate a correct image, as long as the order is preserved by the rasterization process. Depth sorting allows the rasterization system to be “dumb” in terms of visible surface determination. All it requires is that the rasterizer draw the geometry in the order supplied. For software rasterizers, this is often a useful feature, since entirely software-based 3D systems tend to do whatever possible to avoid putting more work into the (already overburdened) rasterization code.

However, rasterizers were some of the first parts of the raster graphics pipeline to be accelerated with purpose-built hardware, meaning that a

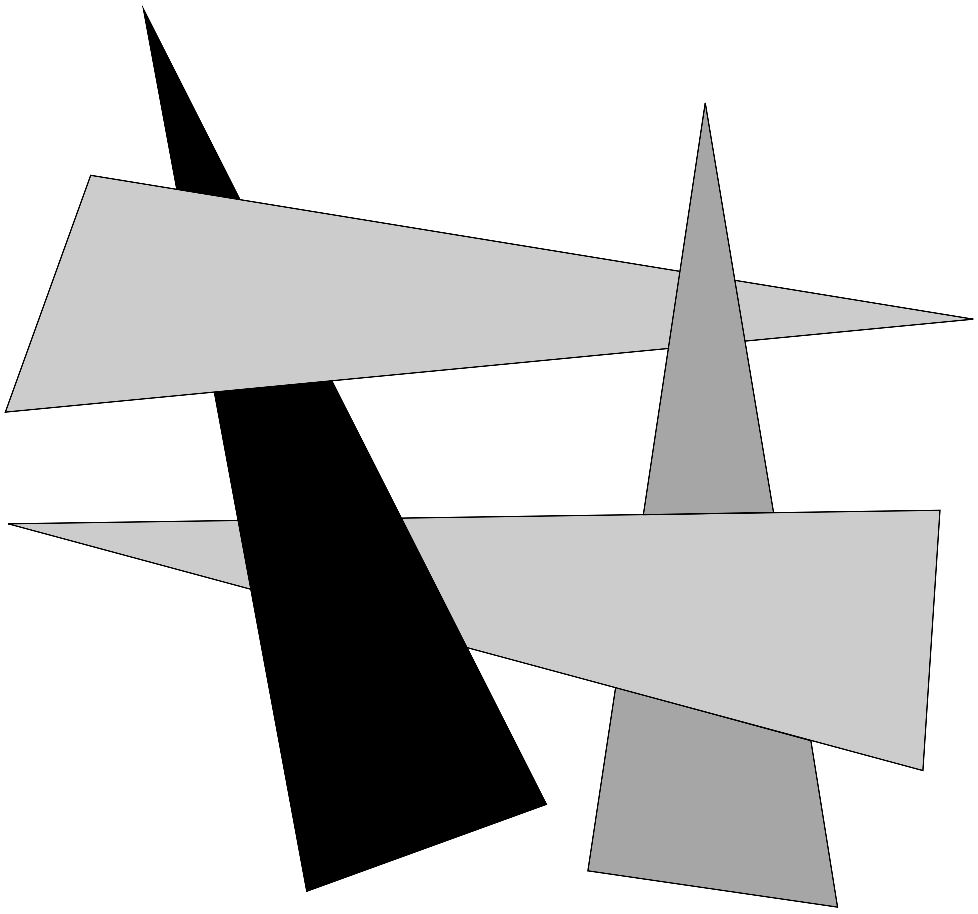


Figure 8.6 Triangle configuration that cannot be depth-sorted without splitting.

rasterizer-based visible surface determination system could achieve high performance. The depth buffer (also known as a "z-buffer," which is actually a special case of depth buffering) is such a rasterizer-based visibility system.

8.5.2 Depth Buffering

Depth buffering is based on the concept that visibility should be output-focused. In other words, since pixels are the final destination of our rendering pipeline, visibility should be computed on a per-pixel basis. If the final color seen at each pixel is the color of the surface with the minimum depth (of all surfaces drawn to that pixel), the scene will appear to be drawn correctly.

In other words, of all the surfaces drawn to a pixel, the surface with minimum depth should “win” the pixel and select that pixel’s color.

Since common rasterization methods tend to render a triangle at a time, a given pixel may be drawn several times over the course of a frame. If we wish to avoid sorting the triangles by depth (and we do), then the triangle that should win a given pixel may not be the last one drawn to that pixel. We must have some method of storing the depth of the current “nearest triangle” at each pixel, along with the color of that triangle.

Having stored this information, we can compute a simple test each time a pixel is drawn. If the new triangle’s depth is closer than the currently stored depth value at that pixel, then the new triangle writes its color to the pixel and its depth to the depth value for that pixel. If the new triangle has greater depth than that of the current triangle coloring the pixel, then the new triangle’s color and depth are ignored, as it represents a surface that is behind the closest known triangle at the current pixel. Figure 8.7 represents the rendering of two triangles to a small depth buffer. Note how the closer triangle always wins the pixel (the correct result), even if it is drawn first.

Because the method is per-pixel, there is no need to determine some single metric of “overall depth” that represents an entire triangle. The depth of each triangle is computed per-pixel, and this value is used in the comparison. As a result, the depth buffer automatically handles configurations that cannot be correctly displayed using triangle sorting. Geometry may be passed to the depth buffer in any order. The situation in which this random order can be problematic is when two surfaces have equal depth at a given pixel. In this case order will matter, depending on the exact comparison used to order depth (i.e., $<$ or $\leq$). However, such circumstances are problematic with almost any visible surface method.

There are several drawbacks to the depth buffer. One of the drawbacks of the depth buffering method is implied in the name of the method; it requires a *buffer* of depth values, one per pixel. This is a large block of memory, generally requiring as much memory as (or more than) the framebuffer itself. Also, just as the framebuffer must be cleared to the background color before each frame, the depth buffer must be cleared to the “background depth,” which is generally the maximum representable depth value. Finally, the depth buffer requires the following work for *each pixel covered by each triangle*:

- Computation of a depth value for the triangle
- Lookup of the existing pixel depth in the depth buffer
- Comparison of these two values
- (For new “winner” pixels only) Writing the new depth to the depth buffer

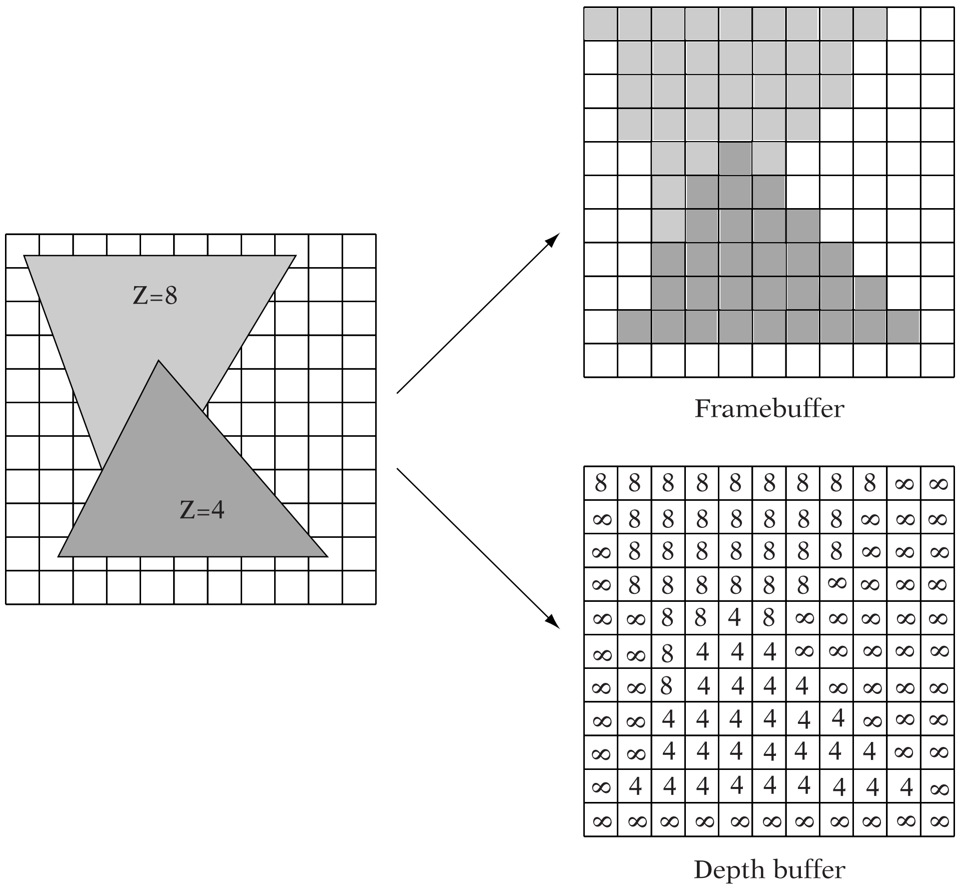


FIGURE 8.7 Two triangles rendered to a depth buffer.

This additional work per pixel covered by each triangle makes depth buffering unsuitable for constant use in most software rasterizers. Fully-software 3D systems tend to use depth sorting wherever possible, reserving depth buffering for the few objects that truly require it.

In addition, the depth buffer does not fix the problem of overdraw. We must still compute the depth of every triangle pixel and compare it to the buffer. However, it can make overdraw less of an issue in some cases, since it is not necessary to compute or write the color of any pixel that fails the depth test. In fact, some applications will try to render their depth-buffered scenes in roughly front-to-back ordering so that the later geometry is likely to fail the depth buffer test and not require color computations.

Depth buffering is extremely popular in 3D applications that run on hardware-accelerated platforms, as it is easy to use and requires little application code or host CPU computation and produces quality images at high performance.

Computing Per-Pixel Depth Values

The first step in computing the visibility of a pixel using a depth buffer is to compute the depth value of the current triangle at the given pixel. As we shall see, z_{ndc} (which appeared to be a rather strange choice for z back in Chapter 5) will work quite well. However, the reason why z_{ndc} works well and z_{view} does not is rather interesting.

In order to better understand the nature of how depth values change across a triangle in screen space, we must be able to map a point on the screen to the point in the triangle that projected to it. This is very similar to picking, and we will use several of the concepts we first discussed in Chapter 5. Owing to the nonlinear nature of perspective projection, we will find that our mapping from screen space pixels to view space points on a given triangle is somewhat complicated. We will follow this mapping through several smaller stages.

A triangle in view space is simply a convex subset of a plane in view space. As a result, we can define the plane of a triangle in view space by the values $\hat{\mathbf{n}}$ and c, such that the points $P = (x_p, y_p, z_p)$ in the plane are those that satisfy

$$\hat{\mathbf{n}} \cdot (x_p, y_p, z_p) + c = 0 \tag{8.1}$$

Looking back at picking, a point in 2D NDC coordinates (x_{ndc}, y_{ndc}) maps to the view space ray $t\mathbf{r}$ such that

$$t\mathbf{r} = (x_{ndc}, y_{ndc}, -d)t, \quad t \geq 0$$

where d is the projection distance (the distance from the view space origin to the projection plane). Any point in view space that projects to the pixel at (x_{ndc}, y_{ndc}) must intersect this ray. Normally, we cannot "invert" the projection matrix, since a point on the screen maps to a ray in view space. However, by knowing the plane of the triangle, we can intersect the triangle with the view ray as follows. All points P in view space that fall in the plane of the triangle are given by equation 8.1. In addition, we know that the point on the triangle that projects to (x_{ndc}, y_{ndc}) must be equal to $t\mathbf{r}$ for some t. Substituting the vector $t\mathbf{r}$ for the points (x_p, y_p, z_p) in equation 8.1 and solving for t,

$$\begin{aligned}
\hat{\mathbf{n}} \cdot (t\mathbf{r}) + c &= 0 \\
t(\hat{\mathbf{n}} \cdot \mathbf{r}) &= -c \\
t &= \frac{-c}{\hat{\mathbf{n}} \cdot \mathbf{r}}
\end{aligned}$$

From this value of t, we can compute the point along the projection ray $(x_{view}, y_{view}, z_{view}) = t\mathbf{r}$ that is the view space point on the triangle that projects to (x_{ndc}, y_{ndc}). This amounts to finding

$$\begin{aligned}(x_{view}, y_{view}, z_{view}) &= t\mathbf{r} \\ &= t(x_{ndc}, y_{ndc}, -d) \\ &= \frac{-c(x_{ndc}, y_{ndc}, -d)}{\hat{\mathbf{n}} \cdot \mathbf{r}} \\ &= \frac{-c(x_{ndc}, y_{ndc}, -d)}{\hat{\mathbf{n}} \cdot (x_{ndc}, y_{ndc}, -d)} \\ &= \frac{-c(x_{ndc}, y_{ndc}, -d)}{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z d} \qquad (8.2)\end{aligned}$$

However, we are only interested in z_{view} right now, since we are trying to compute a per-pixel value for depth buffering. The z_{view} component of equation 8.2 is

$$z_{view} = \frac{dc}{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z d} \qquad (8.3)$$

As a quick check of a known result, note that in the special case of a triangle of constant depth $z_{view} = z_{const}$, we can substitute

$$\hat{\mathbf{n}} = (0, 0, 1)$$

and

$$c = -z_{const}$$

Substituted into equation 8.3 evaluates to the expected constant $z_{view} = z_{const}$:

$$\begin{aligned}z_{view} &= \frac{d(-z_{const})}{0x_{ndc} + 0y_{ndc} - 1d} \\ &= \frac{-dz_{const}}{-d} \\ &= z_{const}\end{aligned}$$

As defined in equation 8.3, z_{view} is an expensive value to compute per pixel (in the general, nonconstant depth case), because it is a fraction with a nonconstant denominator. This would require a per-pixel division to compute

z_{view}, which is more expensive than we would like. However, depth buffering requires only the ability to compare depth values against one another. If we are comparing z_{view} values, we know that they decrease with increasing depth (as the view direction is $-z$), giving a depth test of

$$\begin{aligned} z_{view} &\geq DepthBuffer \rightarrow \text{New triangle is visible at pixel} \\ z_{view} &< DepthBuffer \rightarrow \text{New triangle is not visible at pixel} \end{aligned}$$

However, if we compute and store inverse z_{view}, then a similar comparison still works in the same manner. If we invert all of the z_{view} values, we get

$$\begin{aligned} \frac{1}{z_{view}} &\leq DepthBuffer \rightarrow \text{New triangle is visible at pixel} \\ \frac{1}{z_{view}} &> DepthBuffer \rightarrow \text{New triangle is not visible at pixel} \end{aligned}$$

If we invert equation 8.3, we can see that the per-pixel computation becomes simpler:

$$\begin{aligned} \frac{1}{z_{view}} &= \frac{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z d}{dc} \\ &= \left(\frac{\hat{\mathbf{n}}_x}{dc}\right) x_{ndc} + \left(\frac{\hat{\mathbf{n}}_y}{dc}\right) y_{ndc} - \left(\frac{\hat{\mathbf{n}}_z d}{dc}\right) \end{aligned}$$

where all of the parenthesized terms are constant across a triangle. In fact, this forms an affine mapping of NDC coordinates to $1/z_{view}$. Since we know that there is an affine mapping from pixel coordinates (x_s, y_s) to NDC coordinates (x_{ndc}, y_{ndc}), we can compose these affine mappings into a single affine mapping from screen space pixel coordinates to $1/z_{view}$. As a result, for a given projected triangle

$$\frac{1}{z_{view}} = f x_s + g y_s + h$$

where f, g, and h are real values and are constant per triangle. We define the preceding mapping for a given triangle as

$$InvZ(x_s, y_s) = f x_s + g y_s + h$$

An interesting property of $InvZ(x_s, y_s)$ (or of any affine mapping, for that matter) can be seen from the derivation below

$$\begin{aligned} InvZ(x_s + 1, y_s) - InvZ(x_s, y_s) &= (f(x_s + 1) + gy_s + h) - (fx_s + gy_s + h) \\ &= f(x_s + 1) - (fx_s) \\ &= f \end{aligned}$$

meaning that

$$InvZ(x_s + 1, y_s) = InvZ(x_s, y_s) + f$$

and similarly

$$InvZ(x_s, y_s + 1) = InvZ(x_s, y_s) + g$$

In other words, once we compute our $InvZ$ depth buffer value for any "base" pixel, we can compute the depth buffer value of the next pixel in the scanline by simply adding f. Once we compute a base depth buffer value for a given span, as we step along the scanline, filling the span, all we need to do is add f to our current depth between each pixel (Figure 8.8). This makes the per-pixel computation of a depth value very fast indeed. In fact, once the base $InvZ$ of the first span is computed, we may add or subtract f and g to or from the previous span's base depth to compute the base depth of the next span. This technique is known as *forward differencing*, as we use the difference (or delta) between the value at a pixel and the value at the next pixel to step along, updating the current depth. This method will work for *any* value for which there is an affine mapping from screen space. We refer to such values as *affine in screen space*, or *screen-affine*.

In fact, we can use the z_{ndc} value that we computed during projection as a replacement for $InvZ$. In Chapter 5, on viewing and projection, we computed a z_{ndc} value that is equal to -1 at the near plane and 1 at the far plane and was of the form

$$z_{ndc} = \frac{a + bz_{view}}{z_{view}} = a\frac{1}{z_{view}} + b$$

which is an affine mapping of $InvZ$. As a result, we find that our existing value z_{ndc} is screen-affine and is suitable for use as a depth buffer value. This is the special case of depth buffering we mentioned earlier, often called "z-buffering," as it uses z_{ndc} directly.

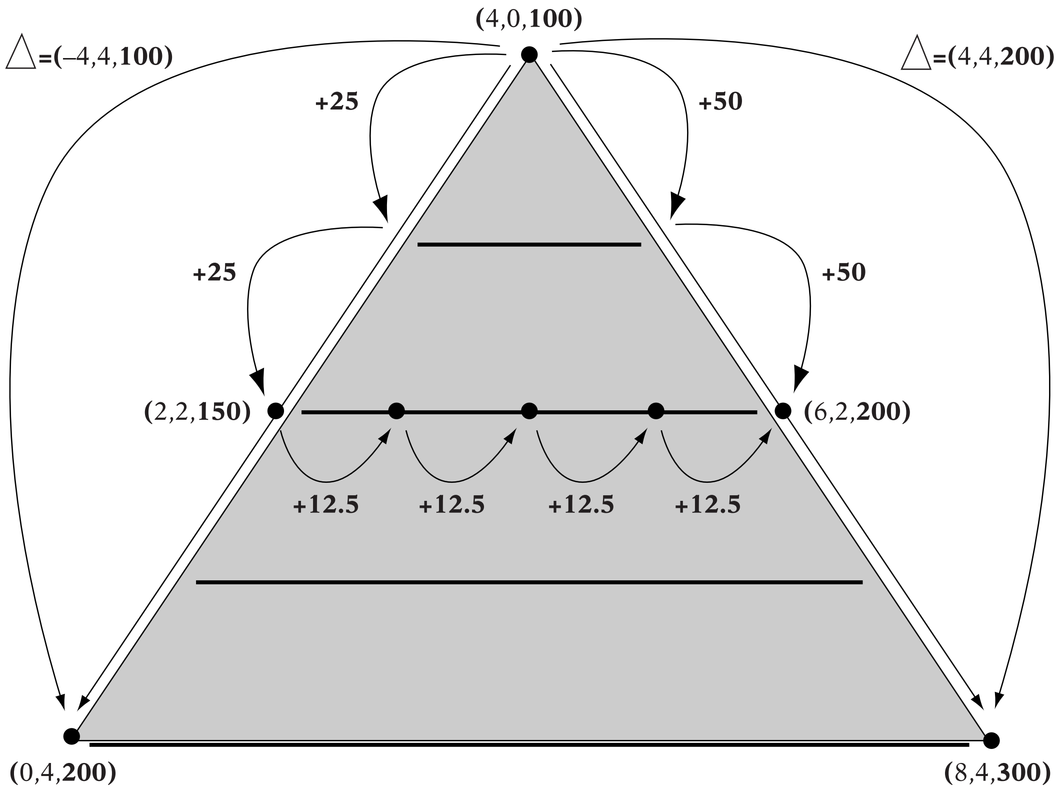


FIGURE 8.8 Forward differencing the depth value.

Numerical Precision and Z-Buffering

In practice, depth buffering in screen space has some numerical precision limitations that can lead to visual artifacts. As was mentioned earlier in the discussion of depth buffers, the order in which objects are drawn to a depth buffering system (at least in the case of opaque objects) is only an issue if the depth values of the two surfaces are equal at a given pixel. In theory, this is unlikely to happen unless the geometric objects in question are truly coplanar. However, because computer number representations do not have infinite precision (recall the discussion in Chapter 4), surfaces that are not coplanar can map to the same depth value. This can lead to objects being drawn in the wrong order.

If our depth values were mapped linearly into view space, then a 16-bit, fixed-point depth buffer would be able to correctly sort any objects whose surfaces differed in depth by about one 60,000th of the difference between the near and far plane distances. This would seem to be more than enough for almost any application. For example, with a view distance of 1 km, this would be equal to about 1.5 cm of resolution. Moving to a higher-resolution depth buffer would make this value even smaller.

However, in the case of z-buffering, representable depth values are not evenly distributed in view space. In fact, the depth values stored to the buffer are basically $1/Z_{view}$, which is definitely not an even distribution of view space Z. A graph of the depth buffer value over view space Z is shown in Figure 8.9. This is a hyperbolic mapping of view space Z into depth buffer values — notice how little the depth value changes with change in Z toward the far plane. Using a fixed-point value for this leads to very low precision in the distance, as large intervals of Z map to the same fixed-point value of inverse Z. In fact, a common estimate is that a z-buffer focuses 90 percent of its precision in the closest 10 percent of view space Z. This means that the triangles of distant objects are often sorted incorrectly with respect to one another.

The simplest way to avoid these issues is to maximize usage of the depth buffer by moving the near plane as far out as possible so that the accuracy close to the near plane is not wasted. Another method that is popular in 3D hardware is known as the w-buffer. The w-buffer interpolates a screen-affine value for depth (often 1/w) at a high precision, then computes the inverse of

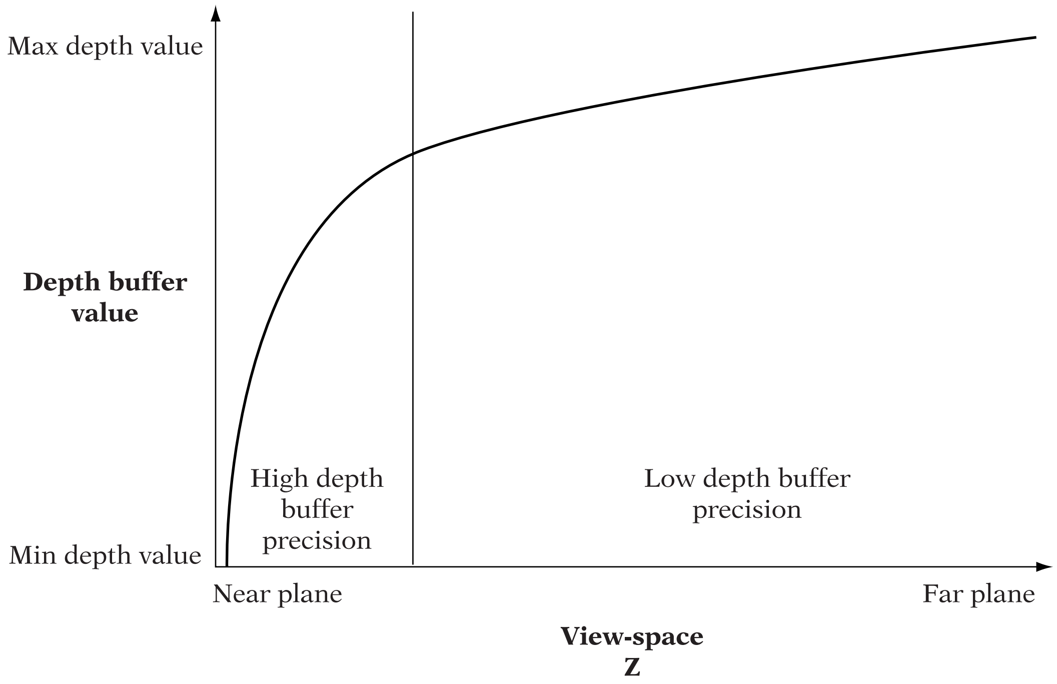


Figure 8.9 Depth buffer value as a function of view-space Z.

the interpolation at each pixel to produce a value that is linear in view space (i.e., $1/\frac{1}{w}$). It is this inverted value that is then stored in the depth buffer. By quantizing (dropping the extra precision used during interpolation) and storing a value that is linear in *view* space, the hyperbolic nature of the z-buffer can be avoided to some degree.

8.5.3 Depth-Buffering in OpenGL

Source Code Demo DepthBuffer

Using depth buffering in OpenGL requires additions to several points in rendering code, somewhat analogous to the stages of rendering color to a pixel. The first step is to ensure that the rendering window or device is created with a depth buffer. This step is platform-dependent in OpenGL. The samples abstract this step into the `IvDisplay` object. The samples request a 16-bit depth buffer, but 32-bit is also common (and growing in popularity).

Having requested the creation of a depth buffer (and in most cases, it is just that — a *request* for a depth buffer, dependent upon hardware support), the buffer must be cleared at the start of each frame. The depth buffer is cleared using the same function as the framebuffer clear, `glClear`, but with a new argument, `GL_DEPTH_BUFFER_BIT`. While the depth buffer can be cleared independently of the framebuffer using

```
glClear(GL_DEPTH_BUFFER_BIT);
```

if you are clearing both buffers, it can be faster on some systems to clear them both with a single call, combining the masks together with a bitwise "Or" operation as follows:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

To enable or disable depth testing, use `glEnable(GL_DEPTH_TEST)` and `glDisable(GL_DEPTH_TEST)`, respectively. By default, depth buffering is disabled, so the application should enable it explicitly prior to rendering. When enabled, depth buffering defaults to a mode in which a new pixel is written only if its depth value is less than the current pixel. In other words, in cases of multiple surfaces sharing the same minimum depth in a given pixel, the first surface drawn "wins." To change this, use the function `glDepthFunc` (the default value is equivalent to the argument `GL_LESS`). The *OpenGL Programming Guide* [83] details all of the possible options, but the next most common mode is `GL_LEQUAL`, which causes the depth "tiebreaker" to favor the last surface drawn at a given depth.

In the somewhat rare situation that the application needs to change the depth to which the z-buffer is cleared (by default, it is the maximum representable distance), it may do so using `glClearDepth`, passing in the desired floating-point clearing depth as the only argument.

8.6 Computing Source Pixel Colors

The next stage in the rasterization pipeline is to compute the overall color (and possibly alpha value) of a triangle at a given pixel. These source colors can come in numerous forms, as discussed in the previous two chapters. Common sources include:

- Per-triangle ("flat" colors) diffuse colors, including those generated by lighting
- Per-vertex (Gouraud colors) diffuse colors, including those generated by lighting
- Per-vertex specular colors
- Textures

Note that several sources may exist for a given triangle. Each of them must be independently computed per-pixel as a part of source color generation. Having computed the per-pixel source colors, a final source pixel color must be generated. Chapter 7 discussed the various ways that per-pixel diffuse, specular, and texture colors are combined. These methods all generate a final source pixel color that is passed to the last stage of the rasterization pipeline, blending (which will be discussed later in this chapter).

The next few sections will discuss how source colors are computed per-pixel from the sources we have listed. While there are many possible methods that may be used, we will focus on methods that are fast to compute and are well-suited to the scanline-centric nature of most rasterizer software and hardware.

8.6.1 Flat Colors

As with all other stages in the pipeline, per-triangle, flat-shaded colors are the easiest to rasterize. For each visible pixel in each span, the triangle color is the source pixel color. In fact, if the source pixel color is to be used directly as the final pixel color (i.e., blending and textures are not enabled, as we discuss in Section 8.8 on blending), then the entire span may be drawn very quickly by writing the given triangle color to the consecutive pixels in an extremely tight code loop. This is one of the reasons that flat-shaded triangles were the first primitives to be rasterized in early raster-based 3D graphics systems, where per-pixel computation had to be kept to an absolute minimum.

8.6.2 Gouraud Colors

Gouraud shaded colors are defined by the colors at the three vertices of each triangle, and thus their values must be interpolated and recomputed for each pixel in the triangle. In the general case this can be an expensive operation to compute correctly. However, we will first look at the special case of triangles of constant depth. The mapping in this case is not at all expensive, making it a tempting approximation to use even when rendering triangles of nonconstant depth.

To analyze the constant-depth case, we will determine the nature of the mapping of our constant-depth triangle from pixel space, through NDC space, into view space, through barycentric coordinates, and finally to color. We start first with a special case of the mapping from pixel space to view space.

The overall projection equations derived in Chapter 5 (mapping from view space through NDC space to pixel coordinates) were all of the form

$$x_s = \frac{a x_{view}}{z_{view}} + b$$
$$y_s = \frac{c y_{view}}{z_{view}} + d$$

where both $a, c \neq 0$. If we assume that a triangle's vertices are all at the same depth (i.e., view space Z is equal to a constant z_{const} for all points in the triangle), then the projection of a point in the triangle is

$$x_s = \frac{a x_{view}}{z_{const}} + b = \left(\frac{a}{z_{const}}\right) x_{view} + b = a' x_{view} + b$$
$$y_s = \frac{c y_{view}}{z_{const}} + d = \left(\frac{c}{z_{const}}\right) y_{view} + d = c' y_{view} + d$$

Note that $a, c \neq 0$ implies that $a', c' \neq 0$, so we can rewrite these such that

$$x_{view} = \frac{x_s - b}{a'}$$
$$y_{view} = \frac{y_s - d}{c'}$$

Thus, for triangles of constant depth z_{const}

- Projection forms an affine mapping from screen vertices to view-space vertices on the $z_{view} = z_{const}$ plane.
- Barycentric coordinates are an affine mapping of view-space vertices (as we saw in Chapter 1).
- Vertex colors define an affine mapping from a barycentric coordinate to a color (Gouraud shading, as seen in Chapter 6).

If we compose these affine mappings, we end up with an affine mapping from screen space pixel coordinates to color. We can write this affine mapping from pixel coordinates to colors as

$$Color(x_s, y_s) = C_x x_s + C_y y_s + C_0$$

where C_x, C_y, and C_0 are all colors (each of which are possibly negative or greater than 1.0). For a derivation of the formula that maps the three screen space pixel positions and corresponding trio of vertex colors to the three colors C_x, C_y, and C_0, see page 126 of [27]. From our earlier derivation of the properties of inverse Z in screen space, we note that $Color(x_s, y_s)$ is screen-affine *for triangles of constant* z:

$$\begin{aligned} Color(x_s+1, y_s) - Color(x_s, y_s) &= (C_x(x_s+1) + C_y y_s + C_0) - (C_x x_s + C_y y_s + C_0) \\ &= C_x(x_s+1) - (C_x x_s) \\ &= C_x \end{aligned}$$

meaning that

$$Color(x_s + 1, y_s) = Color(x_s, y_s) + C_x$$

and similarly

$$Color(x_s, y_s + 1) = Color(x_s, y_s) + C_y$$

As with inverse Z, we can compute per-pixel Gouraud colors for a constant-z triangle simply by computing forward differences of the color of a "base pixel" in the triangle.

When a triangle that does not have constant depth in camera space is projected using a perspective projection, the resulting mapping is not screen-affine. From our discussion of depth buffer values, we can see that given a general (not necessarily constant depth) triangle in view space, the mapping from NDC space to the view-space point on the triangle is of the form

$$x_{view} = \frac{d x_{ndc}}{a x_{ndc} + b y_{ndc} + c}$$

$$y_{view} = \frac{d' y_{ndc}}{a x_{ndc} + b y_{ndc} + c}$$

$$z_{view} = \frac{d''}{a x_{ndc} + b y_{ndc} + c}$$

These are projective mappings, not affine mappings as we had in the constant-depth case. This means that the overall mapping from screen space to Gouraud colors is also projective. Such a projective mapping requires two forward differences (one for the numerator and one for the denominator) and a division per color component, *per pixel*. In order to correctly interpolate vertex colors of a triangle in perspective, we must use this more complex projective mapping.

Keeping in mind that Gouraud shading is an approximation method in the first place, there is somewhat decreased justification for using the projective mapping on the basis of "correctness." Furthermore, Gouraud-shaded colors tend to interpolate so smoothly that it can be difficult to tell whether the interpolation is perspective correct or not. In fact, Heckbert and Moreton mention in [58] that the New York Institute of Technology's off-line renderer interpolated colors incorrectly in perspective for several years before anyone noticed! As a result, hardware and (especially) software graphics systems have often avoided the expensive, perspective-correct projective interpolation of Gouraud colors and have simply used the affine mapping and forward differencing. However, our next interpolant, texture coordinates, will require us to be far more careful with perspective issues.

8.7 Rasterizing Textures

Rasterizing textures requires several independent steps. First, the texture coordinates must be correctly interpolated to determine a value at each pixel. Then, these texture coordinates must be mapped into the texture to produce a color. Both of these steps raise completely different issues, both mathematical and algorithmic. The following sections will detail the most important issues arising from each step in the process.

8.7.1 Texture Coordinate Review

We will be using a number of different forms of coordinates throughout our discussion of rasterizing textures. This section will list and review these various texture-related coordinates and their notations.

The first form of coordinates is most commonly known simply as *texture coordinates*. These were the most common form of texture-related coordinates in our initial discussion of texturing. These are independent of the height and width of a texture and are normalized such that (0,0) represents the bottom-left corner of a texture image, and (1,1) represents the upper-right corner of a texture image. These are generally stored as real-valued numbers, namely, floating-point or fixed-point coordinates. They are the coordinates that most graphics systems use at the application level. They are very convenient for most applications, as they are independent of the exact resolution of the texture. However, they are not very useful at all when rasterizing textures, and we will use them very rarely in the following rasterization discussions. We notate texture coordinates simply as (u, v).

The next form of coordinates is often referred to as *texel coordinates*. Like texture coordinates, texel coordinates are represented as real-valued numbers. However, unlike texture coordinates, texel coordinates are dependent upon the width ($w_{texture}$) and height ($h_{texture}$) of the texture image being used. We will notate texel coordinates as (u_{texel}, v_{texel}). The mapping from (u, v) to (u_{texel}, v_{texel}) is

$$(u_{texel}, v_{texel}) = (u \cdot w_{texture} - \frac{1}{2}, v \cdot h_{texture} - \frac{1}{2})$$

The shift of 1/2 may seem odd, but Figure 8.10 shows why this is necessary. Texel coordinates are relative to the texel *centers*. A texture coordinate of zero is on the boundary between two repetitions of a texture. Since the texel centers are at the middle of a texel, a texture coordinate that falls on an integer value is really halfway between the center of the last texel of one repetition of the texture, and the center of the first texel in the next repetition. This is equivalent to a texel coordinate of $-1/2$. See [77] (the section "Directly Mapping Texels to Pixels") for details of one common graphics systems texture coordinate to texel mapping.

Another form of coordinate is the *integer texel coordinate*, or *texel address*. Unlike the other forms of coordinates, these are (as the name implies) integral values. As such, they can be used to index a texture directly (once the wrapping or clamping mode is applied as we first discussed in Chapter 6 on texturing). We will notate integer texel coordinates as (u_{int}, v_{int}). Integer texel coordinates are the values sent to the image lookup function $Image(u_{int}, v_{int})$, discussed in the introduction to texturing. The mapping from texel coordinates to integer texel coordinates is not universal and is dependent upon the

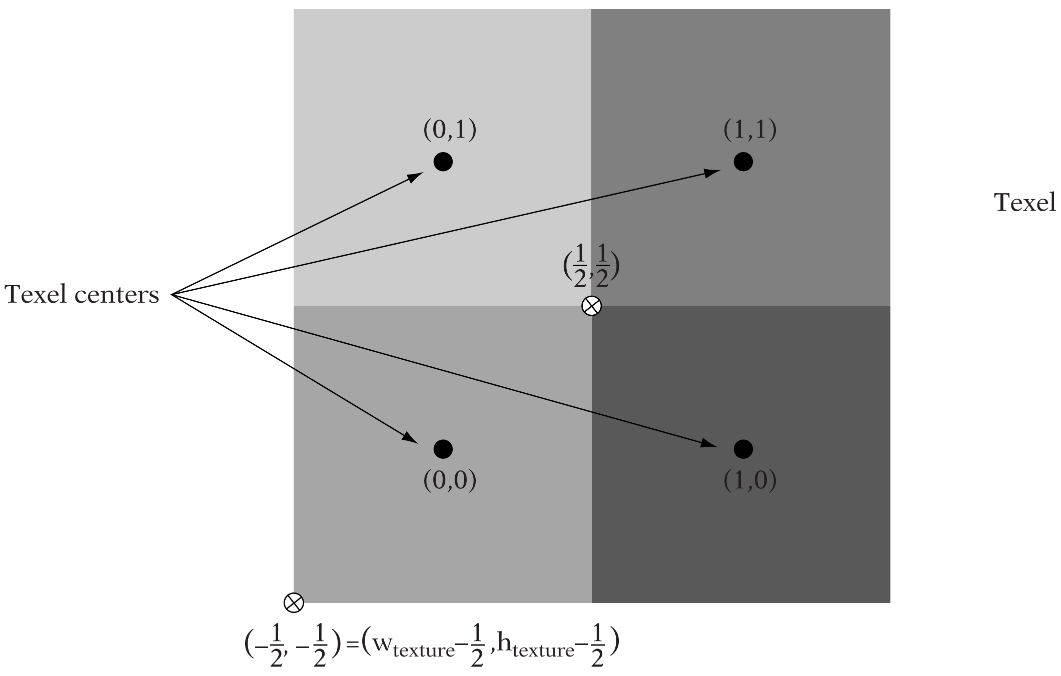

(0,1)
(1,1)
Texel
$(\frac{1}{2},\frac{1}{2})$
Texel centers
(0,0)
(1,0)
$(-\frac{1}{2},-\frac{1}{2}) = (w_{texture}-\frac{1}{2}, h_{texture}-\frac{1}{2})$

Figure 8.10 Texel coordinates and texel centers.

texture filtering mode, which will be discussed in Section 8.7.4 under "Texture Filtering and Mipmaps."

8.7.2 Interpolating Texture Coordinates

The process of rasterizing a texture starts by interpolating the per-vertex texture coordinates to determine the correct value at each pixel. Actually, as alluded to in the previous section, it is generally the texel coordinates that are interpolated in a rasterizer. This is a process that is very similar to interpolating colors for Gouraud shading. However, because texture coordinates are used somewhat differently than vertex colors, we are rarely able to use the screen-affine approximation that is used for Gouraud colors.

The most basic issue has to do with the properties of affine and projective transformations. Affine transformations map parallel lines to parallel lines, while projective transformations guarantee only to map straight lines to straight lines. Anyone who has ever looked down a long, straight road knows that the two lines that form the edges of the road appear to meet in

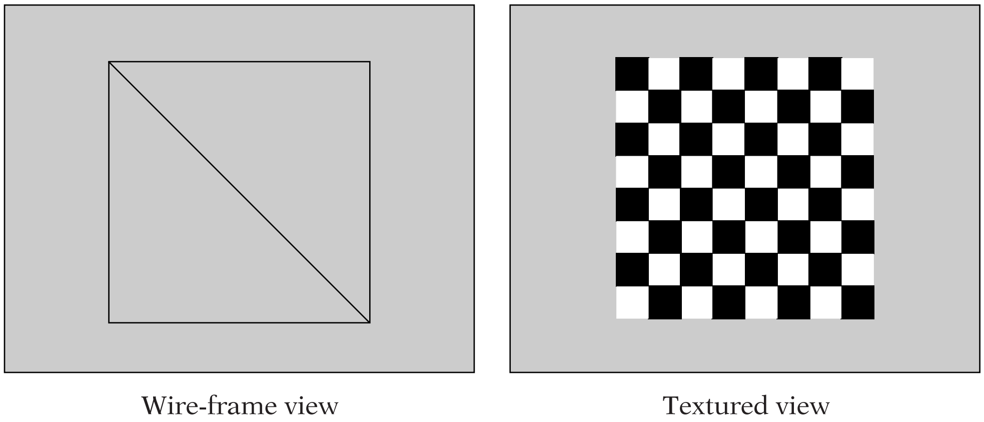


FIGURE 8.11 Two textured triangles parallel to the view plane.

the distance, even though they are parallel. Perspective, being a projective mapping, does not preserve parallel lines.

The classic example of the difference between affine and projective interpolations is the checkerboard square, drawn in perspective. Figure 8.11 shows a checkered texture as an image, along with the image applied with wrapping to a square formed by two triangles (the two triangles are shown in outline, or *wire frame*). When the top is tilted away in perspective, note that if the texture is mapped using a projective mapping (Figure 8.12), the vertical lines converge into the distance as expected.

If the texture coordinates are interpolated using an affine mapping (Figure 8.13), we see two distinct visual artifacts. First, within each triangle, all of the parallel lines remain parallel, and the vertical lines do not converge the way we expect. Furthermore, note the obvious "kink" in the lines along the square's diagonal (the shared triangle edge). This might at first glance seem to be a bug in the interpolation code, but a little analysis shows that it is actually a basic property of an affine transformation. An affine transformation is defined by the three points of a triangle. As a result, having defined the three points of the triangle and their texture coordinates, there are no more degrees of freedom in the transformation. Each triangle defines its transform independent of the other triangles, and the result is a bend in what should be a set of lines across the square.

The projective transform, however, has additional degrees of freedom, represented by the depth values associated with each vertex. These depth values change the way the texture coordinate is interpolated across the

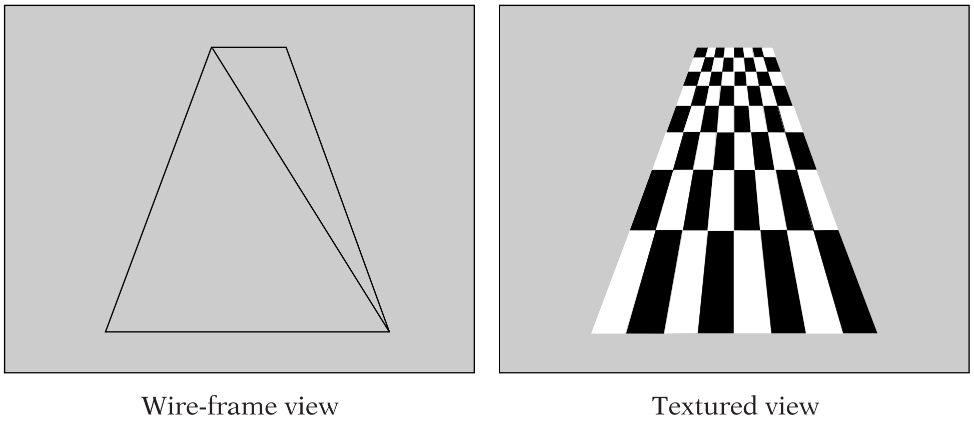


FIGURE 8.12 Two textured triangles oblique to the view plane, drawn using a perspective mapping.

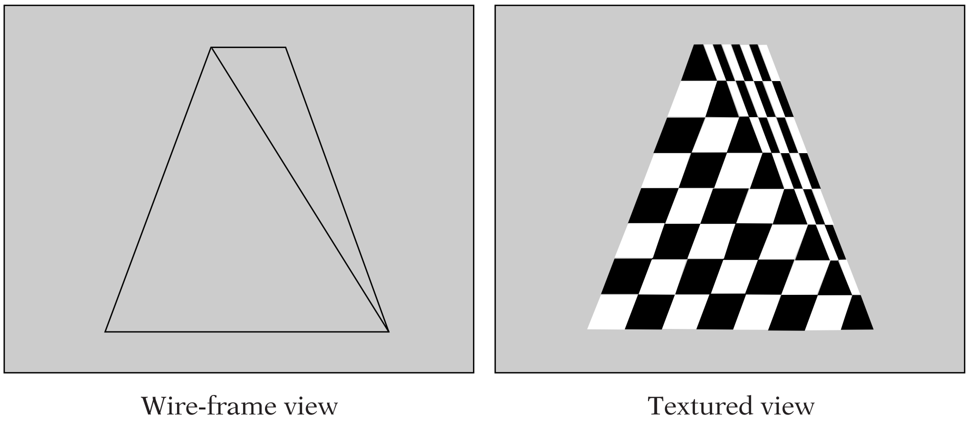


FIGURE 8.13 Two textured triangles oblique to the view plane, drawn using an affine mapping.

triangle and allow the lines to remain straight, even across the triangle boundaries. The downside of this projective mapping is that it requires the following operations per pixel for correct evaluation:

1. An affine forward difference operation to update the numerator for u_{texel}
2. An affine forward difference operation to update the numerator for v_{texel}
3. An affine forward difference operation to update the shared denominator (both u_{texel} and v_{texel} can use the same denominator, as it is based on inverse depth of the triangle at the pixel)
4. A division to recover the perspective-correct u_{texel}
5. A division to recover the perspective-correct v_{texel}

Hardware rasterization systems generally support this operation (or at least a carefully-constructed approximation of it), but for software rasterizers, this is simply too expensive to compute for each pixel. There are numerous optimizations and approximations that have been used in software rasterizers to speed up this process, but they generally fall into two basic categories: (1) subdividing and using piecewise-affine mappings for the resulting short spans and (2) fitting higher-order (e.g., quadratic) curves to approximate the perspective curve. Each method is detailed in [60], including the arguments in favor of and in opposition to each. However, for most modern hardware rasterization systems, per-pixel perspective correct texturing is simply assumed.

8.7.3 Mapping a Coordinate to a Texel

When rasterizing textures, we will find that—due to the nature of perspective projection, the shape of objects, and the way texture coordinates are generated—pixels will rarely correspond directly and exactly to texels in a one-to-one mapping. Any rasterizer that supports texturing will need to handle a wide range of texel-to-pixel mappings. In the initial discussions of texturing in Chapter 6, we noted that texel coordinates generally include precision (via either floating-point or fixed-point numbers) that is much more fine-grained than the per-texel values that would seem to be required. As we shall see, in several cases we will use this so-called sub-texel precision to improve the quality of rendered images in a process known as *texture filtering*.

Texture filtering (in its numerous forms) performs the mapping from real-valued texel coordinates to final colors, through a mixture of texel coordinate mapping and combinations of the colors of the resulting texel or texels. We will break down our discussion of texture filtering into two major cases: one in which a single texel maps to multiple pixels (magnification), and one in which a number of texels map to a single pixel ("minification"), as they are handled quite differently.

Magnifying a Texture

Our initial texturing discussion stated that one common method of mapping these sub-texel precise values to colors was simply to select the nearest texel and use its color directly. This method, called *nearest-neighbor texturing*, is very simple to compute. For any (u_{texel}, v_{texel}) texel coordinate, the integer texel coordinate (u_{int}, v_{int}) is the nearest integer texel center, computed via rounding:

$$(u_{int}, v_{int}) = (\lfloor u_{texel} + 0.5 \rfloor, \lfloor v_{texel} + 0.5 \rfloor)$$

Having computed this integer texel coordinate, we simply use the $Image$ function to look up the color of the texel. The returned value is the source texture color for the pixel. While this method is easy and fast to compute, it has a significant drawback when the texture is mapped in such a way that a single texel covers more than one pixel. In such a case the texture is said to be "magnified," as a quadrilateral block of pixels on the screen is covered by one texel in the texture, as can be seen in Figure 8.14.

With nearest neighbor texturing, all (u_{texel}, v_{texel}) texel coordinates in the square

$$i_{int} - 0.5 \le u_{texel} < i_{int} + 0.5$$
$$j_{int} - 0.5 \le v_{texel} < j_{int} + 0.5$$

will map to the integer texel coordinates (i_{int}, j_{int}) and thus map to a constant color. This is a square of height and width 1 in texel space, centered at the texel center. This results in obvious squares of constant color, which tends to draw attention to the fact that a low-resolution image has been mapped onto the surface (see Figure 8.14). Often, this is not the desired visual impression.

The problem lies with the fact that nearest neighbor texturing represents the texture image as a piecewise constant function of (u, v). The color used is constant across a triangle until either u_{int} or v_{int} changes. Since the floor operation is discontinuous at integer values, this leads to sharp edges in the color function over the surface of the triangle. This is not unlike the issues we encountered with flat shading.

In the case of flat shading, the answer to the issue of discontinuous colors was to interpolate between the colors at each vertex. In the case of texturing, it involves interpolating between the colors at each texel center. Rather than

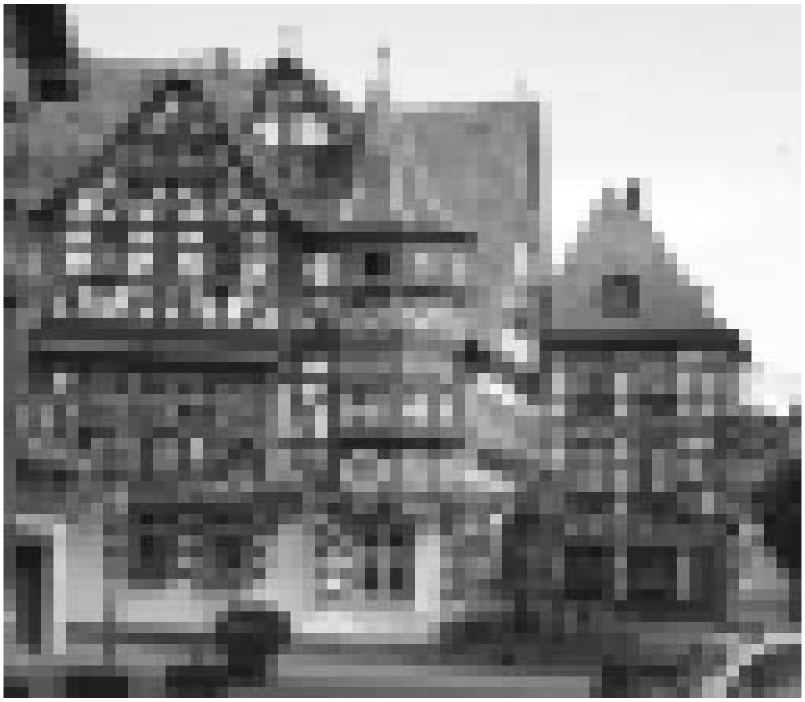


FIGURE 8.14 Nearest-neighbor magnification.

creating a piecewise constant function, we create a piecewise smooth color function. The method first computes the maximum integer texel coordinate (u_{int}, v_{int}) that is less than (u_{texel}, v_{texel}), the texel coordinate (i.e., the floor of the texel coordinates):

$$(u_{int}, v_{int}) = (\lfloor u_{texel} \rfloor, \lfloor v_{texel} \rfloor)$$

In other words, (u_{int}, v_{int}) defines the minimum (lower-left in texture image space) corner of a square of four adjacent texels that "bound" the texel coordinate (Figure 8.15). Having found this square, we can also compute a *fractional texel coordinate* $0.0 \le u_{frac}, v_{frac} < 1.0$ that defines the position of the texel coordinate within the 4-texel square (see Figure 8.15).

$$(u_{frac}, v_{frac}) = (u_{texel} - u_{int}, v_{texel} - v_{int})$$

We use $Image()$ to look up the texel colors at the four corners of the square. For ease of notation, we define the following shorthand

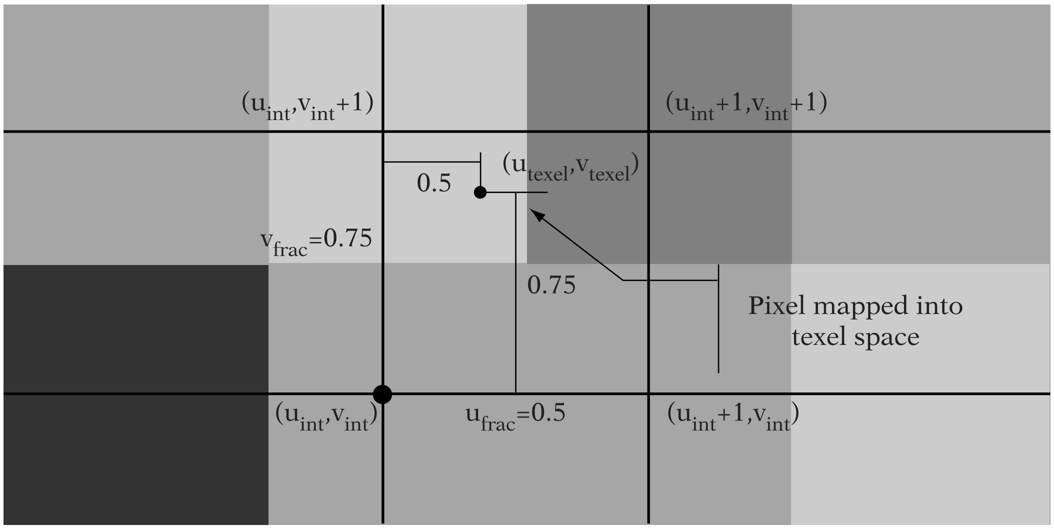


FIGURE 8.15 Finding the four texels that “bound” a pixel center and the fractional position of the pixel.

for the color of the texture at each of the four corners of the square (Figure 8.16):

$$\begin{aligned}
C_{00} &= Image(u_{int}, v_{int}) \\
C_{10} &= Image(u_{int} + 1, v_{int}) \\
C_{01} &= Image(u_{int}, v_{int} + 1) \\
C_{11} &= Image(u_{int} + 1, v_{int} + 1)
\end{aligned}$$

Then, we define a smooth interpolation (called “bilinear filtering”) of the four texels surrounding the texel coordinate. We define the smooth mapping in two stages as shown in Figure 8.17. First, we interpolate between the colors along the minimum-v edge of the square, based on the fractional u coordinate:

$$C_{MinV} = C_{00}(1 - u_{frac}) + C_{10}u_{frac}$$

and similarly along the maximum-v edge:

$$C_{MaxV} = C_{01}(1 - u_{frac}) + C_{11}u_{frac}$$

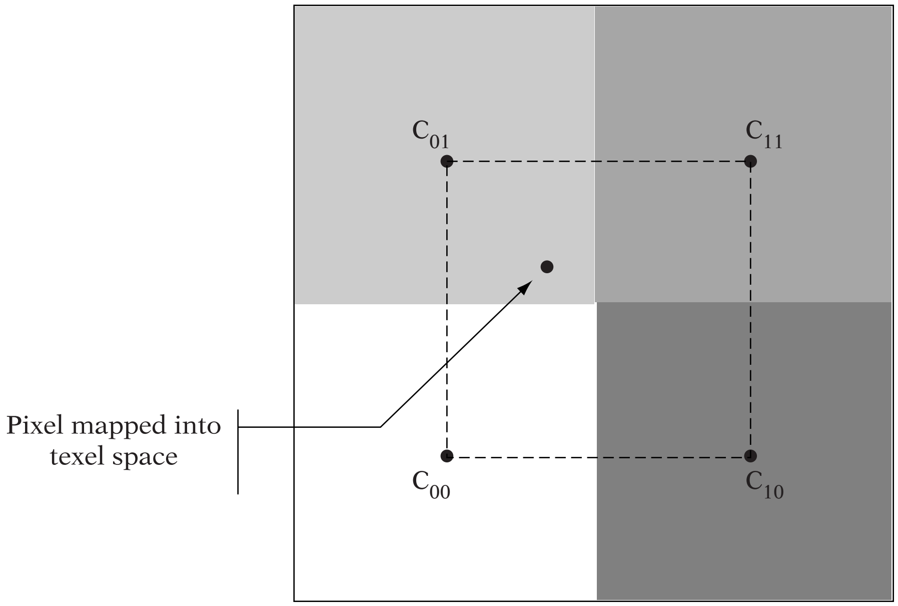


FIGURE 8.16 The four corners of the texel-space bounding square around the pixel center.

Finally, we interpolate between these two values using the fractional v coordinate:

$$C_{Final} = C_{MinV}(1 - v_{frac}) + C_{MaxV} v_{frac}$$

See Figure 8.17 for a graphical representation of these two steps. Substituting these into a single, direct formula, we get

$$\begin{aligned} C_{Final} =& C_{00}(1 - u_{frac})(1 - v_{frac}) + C_{10} u_{frac}(1 - v_{frac}) \\ &+ C_{01}(1 - u_{frac}) v_{frac} + C_{11} u_{frac} v_{frac} \end{aligned}$$

This is known as *bilinear texture filtering*, and is extremely popular in hardware 3D graphics systems. The fact that we interpolated along u first and then interpolated along v does not affect the result (other than by potential precision issues). A quick substitution shows that the results are the same either way. However, note that this is *not* an affine mapping. Four points are not

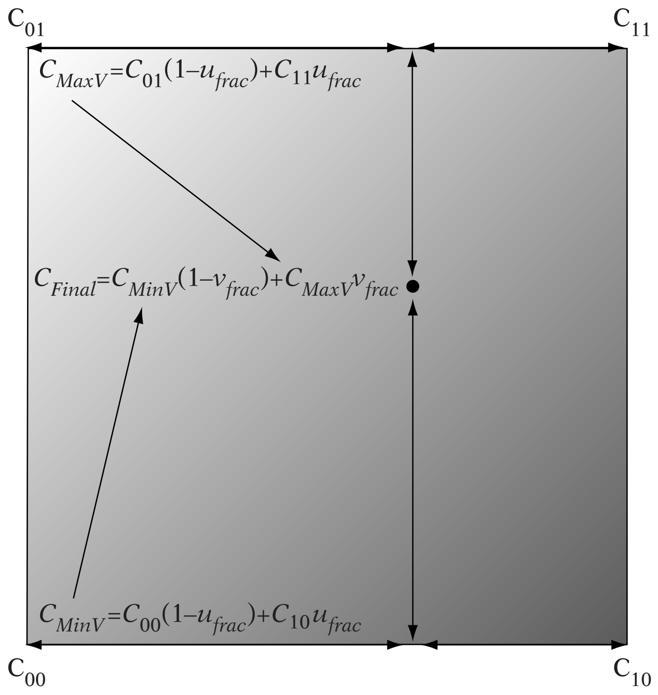

C_{01}
C_{11}
$C_{MaxV}=C_{01}(1-u_{frac})+C_{11}u_{frac}$
$C_{Final}=C_{MinV}(1-v_{frac})+C_{MaxV}v_{frac}$
$C_{MinV}=C_{00}(1-u_{frac})+C_{10}u_{frac}$
C_{00}
C_{10}

FIGURE 8.17 Bilinear filtering.

always coplanar and as a result, in order to fit the four points, the resulting "surface" is not planar.

As with Gouraud shading, the colors along the four boundary edges are continuous — the color at each texel edge is dependent on only the colors at either end of the edge. An example of the visual difference between nearest-neighbor and bilinear filtering is shown in Figure 8.18. While bilinear filtering can greatly improve the image quality of magnified textures by reducing the visual "blockiness," it will not add detail to a texture. If a texture is magnified considerably (i.e., one texel maps to many pixels), the image will look blurry due to a lack of detail. The texture shown in Figure 8.18 is highly magnified, leading to obvious blockiness in the left image and blurriness in the right image.

Texture Magnification in OpenGL

SOURCE CODE DEMO TextureFilter

OpenGL uses the function `glTexParameteri` to control numerous texturing features. The general format of the function for 2D texturing is

```
glTexParameteri(GL_TEXTURE_2D, setting, value);
```

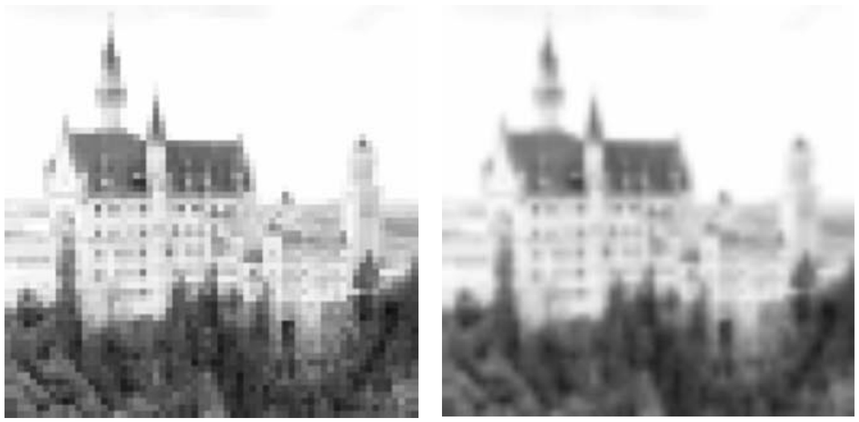

Extreme magnification using nearest-neighbor filtering
Extreme magnification using bilinear filtering

Figure 8.18 Extreme magnification of a texture.

In order to set the magnification method (or filter), `setting` should be passed as `GL_TEXTURE_MAG_FILTER`. OpenGL supports both bilinear filtering and nearest-neighbor selection. They are each set as follows:

```
// Nearest-neighbor
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

// Bilinear interpolation
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

"Minifying" a Texture

Throughout the course of our discussions of coloring and rasterization, we have referred to pixels by their pixel centers—infinite points located at the center of a square pixel. However, pixels (whether on the screen or on the projection plane) have nonzero area. This difference between the area of a pixel and the point sample representing it becomes very obvious in a common case of texturing.

As an example, imagine an object that is distant from the camera. Objects in a scene are generally textured at high detail. This is done to avoid the blurriness (such as the blurriness we saw in Figure 8.18) that can occur when an object that is close to the camera has a low-resolution texture applied to it.

As that same object and texture is moved into the distance (a common situation in a dynamic scene), the detailed texture will be mapped to smaller and smaller regions of the screen due to perspective scaling of the object. This is known as "minification" of a texture, as it is the inverse of magnification.

In an extreme (but actually quite frequent) case, the entire high-detail texture could be mapped in such a way that it covers only a few pixels. Figure 8.19 provides such an example; in this case, note that if the object moves even slightly (even less than a pixel), the exact texel covering the pixel's center point can change drastically. In fact, such a point sample is almost random in the texture and can lead to the color of the pixel changing wildly from frame to frame as the object moves in tiny, sub-pixel amounts on the screen. This can lead to flickering over time, a distracting artifact in an animated, rendered image.

The problem lies in the fact that most of the texels in the texture have almost equal "claim" to the pixel, as all of them are projected within the rectangular area of the pixel on the projection plane. The overall color of the pixel *should* represent all of the texels that fall inside of it. One way of thinking

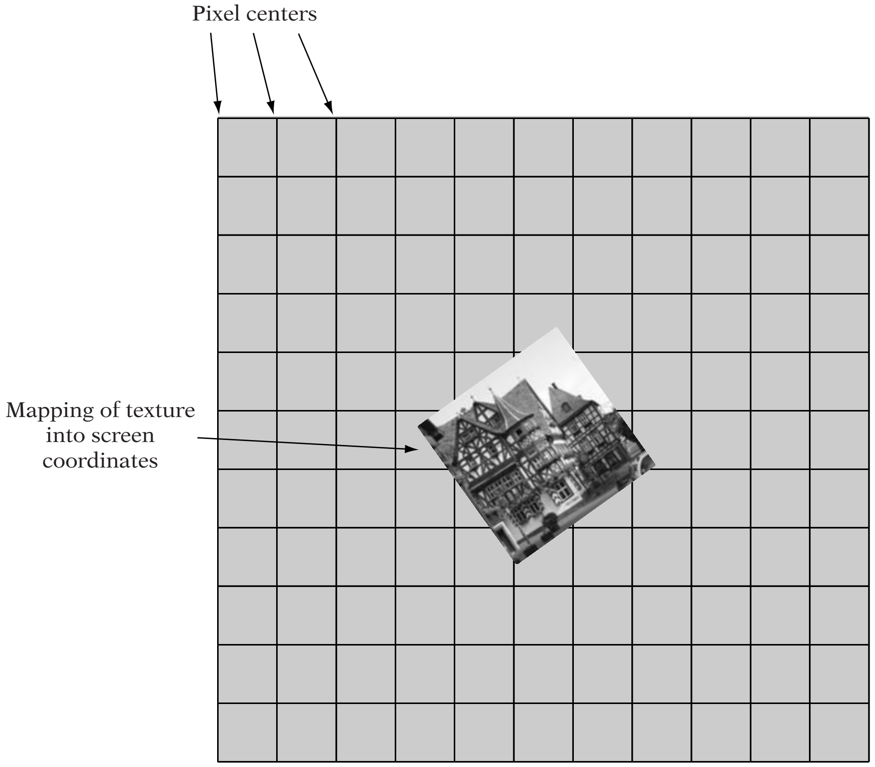


FIGURE 8.19 Extreme "minification" of a texture.

of this is to map the square pixel on the projection plane onto the plane of the triangle, giving a (possibly skewed) quadrilateral, as seen in Figure 8.20. In order to color the pixel "fairly," we need to compute a weighted average of the colors of all of the texels in this quadrilateral, based on the relative area of the quadrilateral covered by each texel. The more of the pixel that is covered by a given texel, the greater the contribution of that texel's color to the final color of the pixel.

While such a method would give a correct pixel color and would avoid the issues seen with point sampling, in reality this is not an algorithm that is best suited for real-time rasterization. Depending on how the texture is mapped, a pixel could cover an almost unbounded number of texels. Finding and summing these texels on a per-pixel basis would require a potentially unbounded amount of per-pixel computation, which is well beyond the means of even hardware rasterization systems. A faster (preferably constant-time) method of approximating this texel averaging algorithm is required. For most modern graphics systems, a method known as *mipmapping* satisfies these requirements.

8.7.4 Mipmapping

Mipmapping [118] is a texture filtering method that avoids the per-pixel expense of computing the average of a large number of texels. It does so by

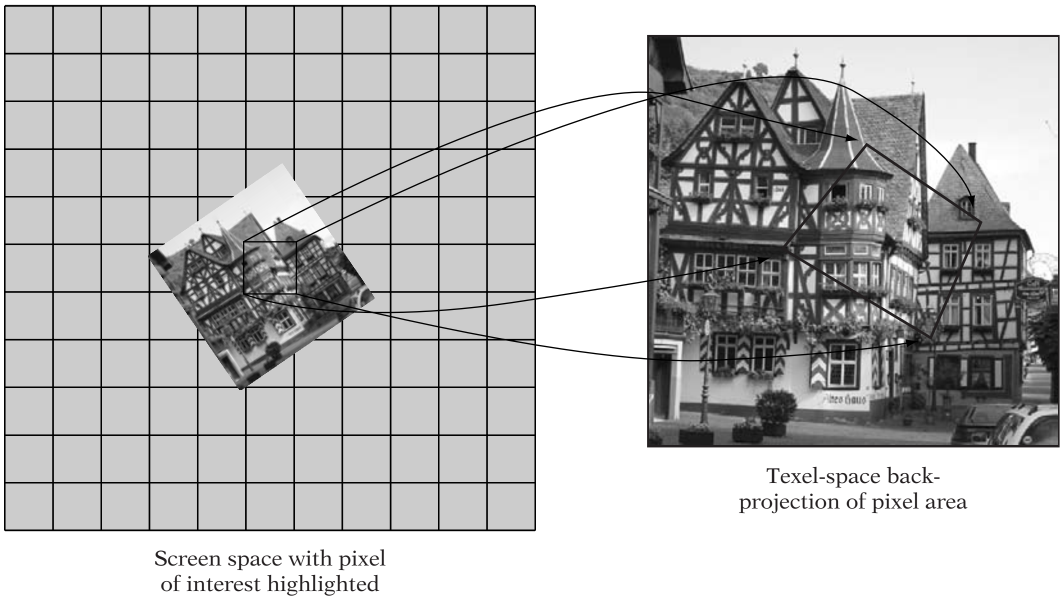

Texel-space back-projection of pixel area
Screen space with pixel of interest highlighted

Figure 8.20 Mapping the square screen-space area of a pixel back into texel space.

precomputing and storing additional information with each texture, requiring some additional memory over standard texturing. Mipmapping is a constant-time operation per pixel and requires a fixed amount of extra storage per texture (in fact, it increases the number of texels that must be stored by approximately one-third). Mipmapping is a popular filtering algorithm in both hardware and software rasterizers and is relatively simple conceptually.

To understand the basic concept behind mipmapping, imagine a 2 × 2–texel texture. If we look at a case where the entire texture is mapped to a single pixel, we could replace the 2 × 2 texture with a 1 × 1 texture (a single color). The appropriate color would be the mean of the four texels in the 2 × 2 texture. We could use this new texture directly. If we precompute the 1 × 1–texel texture at loadtime, we can simply choose between the two textures as needed (Figure 8.21). When the given pixel maps to only one of the four texels in the original texture, we simply use a magnification method and the original texture to determine the color. When the pixel covers the

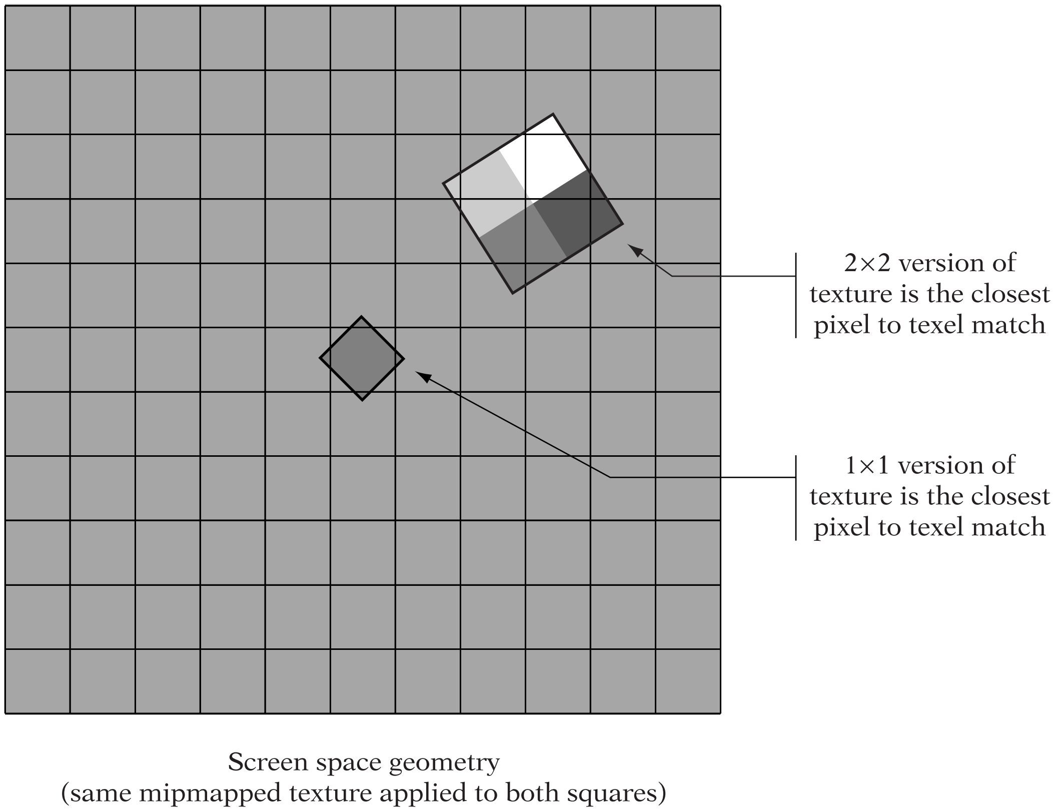

2×2 version of texture is the closest pixel to texel match
1×1 version of texture is the closest pixel to texel match
Screen space geometry
(same mipmapped texture applied to both squares)

FIGURE 8.21 Choosing between two sizes of a texture.

entire texture, we would use the 1 × 1 texture directly, again applying the magnification algorithm to it (although with a 1 × 1 texture, this is just the single texel color). The 1 × 1 texture adequately represents the overall color of the 2 × 2 texture in a single texel, but it does not include the detail of the original 2 × 2 texel texture. Each of these two versions of the texture has a useful feature that the other does not.

Mipmapping takes this method and generalizes it to any texture with power-of-two dimensions. For the purposes of this discussion, we assume that textures are square (the algorithm does not require this, as we shall see later in our discussion of OpenGL's mipmapping support). Mipmapping takes the initial texture image $Image_0$ (abbreviated I_0) of dimension $w_{texture} = h_{texture} = 2^L$ and generates a new version of the texture by averaging each square of four adjacent texels into a single texel. This generates a texture image $Image_1$ of size

$$\frac{1}{2}w_{texture} = \frac{1}{2}h_{texture} = 2^{L-1}$$

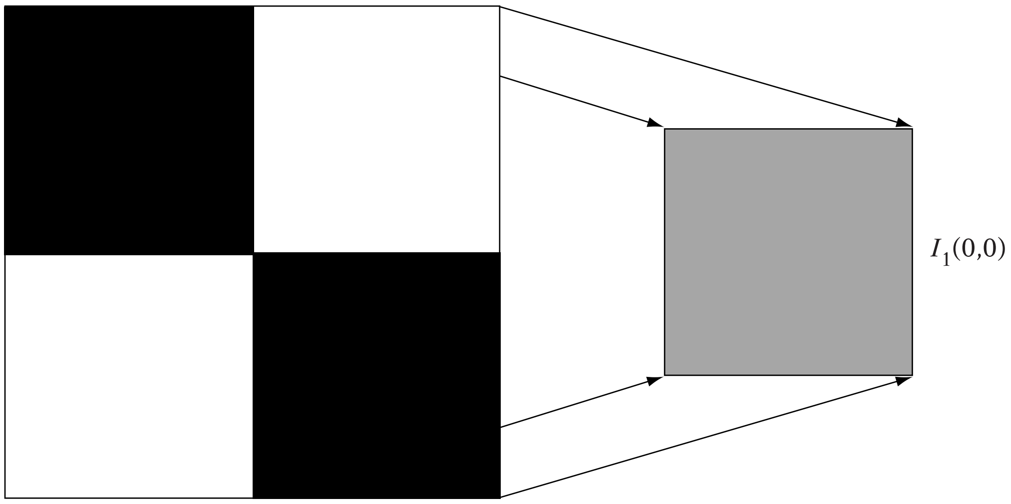
as follows:

$$Image_1(i, j) = \frac{I_0(2i, 2j) + I_0(2i+1, 2j) + I_0(2i, 2j+1) + I_0(2i+1, 2j+1)}{4}$$

where $0 \le i, j < \frac{1}{2}w_{texture}$. Each of the texels in $Image_1$ represents the overall color of a block of the corresponding four texels in $Image_0$ (see Figure 8.22). Note that if we use the same original texture coordinates for both versions of the texture, $Image_1$ simply appears as a blurry version of $Image_0$ (with half the detail of $Image_0$). If a block of about four adjacent texels in $Image_0$ covers a pixel, then we can simply use $Image_1$ when texturing. But what about more extreme cases of minification? The algorithm can be continued recursively. For each image $Image_i$ whose dimensions are greater than 1, we can define $Image_{i+1}$, whose dimensions are half of $Image_i$, and average texels of $Image_i$ into $Image_{i+1}$. This generates an entire set of $L + 1$ versions of the original texture, where the dimensions of $Image_i$ are equal to

$$\frac{w_{texture}}{2^i}$$

This forms a pyramid of images, each one-half the dimensions (and containing one-quarter the texels) of the previous image in the pyramid. Figure 8.23 provides an example of such a pyramid. We compute this pyramid for each texture in our scene once at loadtime and store each entire pyramid in memory. This simple method of computing the mipmap images is known as *box filtering* (as we are averaging a 2 × 2 "box" of pixels into a single pixel). Box filtering is not the sole method for generating the mipmap pyramid, nor is it



$$I_1(0,0) = \frac{I_0(0,0) + I_0(1,0) + I_0(0,1) + I_0(1,1)}{4}$$

$$I_1(0,0) = \frac{(1,1,1) + (0,0,0) + (0,0,0) + (1,1,1)}{4} = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$$

FIGURE 8.22 Texel-block to texel mapping between mipmap levels.

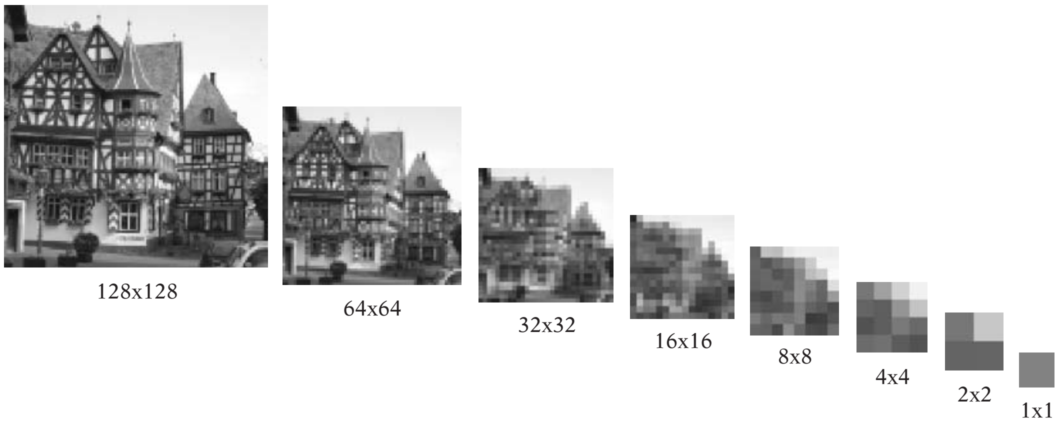


FIGURE 8.23 A full mipmap pyramid for a texture.

the highest-quality. Other, more complex methods are often used to filter each mipmap level down to the next lower level. These methods can avoid some of the visual issues that can crop up from the simple box filter. See Foley, van Dam, Feiner, and Hughes [36], or Wohlberg [120] for details of other image filtering methods.

Texturing a Pixel with a Mipmap

The most simple, general algorithm for texturing a pixel with a mipmap can be summarized as follows:

1. Determine the mapping of the pixel's screen space rectangle into texture space.
2. Having mapped the pixel into a quadrilateral in texture space, select whichever mipmap level comes closest to exactly mapping the quadrilateral to a single texel.
3. Texture the pixel with the "best match" mipmap level selected in the previous step, using the desired magnification algorithm.

There are numerous common ways of determining the "best match" mipmap level, and there are numerous methods of filtering this mipmap level into a final source pixel color. We would like to avoid having to explicitly map the pixel corners back into texture space. As a part of rasterization, it is common to compute the difference between the texel coordinates at a given pixel and those of the pixel to the right and below the given pixel. Such differences are used to step the texture coordinates from one pixel to the next. These differences are written as derivatives. The listing that follows is designed to assign intuitive values to each of these four partial derivatives. (For those unfamiliar with ∂, it is the symbol for a partial derivative, a basic concept of multivariable calculus involving the change of one component of the value of a vector-valued function over change in one of the input components.)

$$\frac{\partial u_{texel}}{\partial x_s} = \text{Change in } u_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial u_{texel}}{\partial y_s} = \text{Change in } u_{texel} \text{ per vertical pixel step}$$

$$\frac{\partial v_{texel}}{\partial x_s} = \text{Change in } v_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial v_{texel}}{\partial y_s} = \text{Change in } v_{texel} \text{ per vertical pixel step}$$

If a pixel maps to about 1 texel, then

$$\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2 \approx 1, \text{ and } \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2 \approx 1$$

In other words, even if the texture is rotated, if the pixel is about the same size as the texel mapped to it, then the overall change in texture coordinates over a single pixel has a length of about 1 texel. Note that all four of these differences are independent. These partials are dependent upon u_{texel} and v_{texel}, which are in turn dependent upon texture size. In fact, for each of these differentials, moving from $Image_i$ to $Image_{i+1}$ causes the differential to be halved. As we shall see, this is a useful property when computing mipmapping values.

A common formula that is used to turn these differentials into a metric of pixel-texel size ratio is described in [57], which defines a formula for the *radius* of a pixel as mapped back into texture space

$$size = \max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)$$

This value is halved each time we move from $Image_i$ to $Image_{i+1}$. So, in order to find a mipmap level at which we map one texel to the pixel, we must compute the L such that

$$\frac{size}{2^L} \approx 1$$

where $size$ is computed using the texel coordinates for $Image_0$. Solving for L,

$$L = \log_2 size$$

This value of L is the mipmap level index we should use. Note that if we plug in partials of 1, we get $size = 1$, which leads to $L = 0$, which corresponds to the original texture image as expected.

This gives us a closed-form method that can convert existing partials (used to interpolate the texture coordinates across a scanline) to a specific mipmap level L. The final formula is

$$
\begin{aligned}
L &= log_2\left(\max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)\right) \\
&= log_2\left(\sqrt{\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)}\right) \\
&= \frac{1}{2}log_2\left(\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)\right)
\end{aligned}
$$

Note that the value of L is real, not an integer—we will discuss the methods of mapping this value into a discrete mipmap pyramid later. The preceding function is only one possible option for computing the mipmap level L. Graphics systems use numerous simplifications and approximations of this value (which is itself an approximation) or even other functions to determine the correct mipmap level. In fact, the particular approximations of L used by some hardware devices are so distinct that some experienced users of 3D hardware can actually recognize a particular piece of display hardware by looking at rendered, mipmapped images. Other pieces of 3D hardware allow the developer (or even the end user) to adjust the mipmap level used, as some users prefer "crisp" images (tending toward a more detailed mipmap level and more texels per pixel) while others prefer "smooth" images (tending toward a less detailed mipmap level and fewer texels per pixel). For a detailed derivation of one case of mipmap level selection, see page 106 of Eberly [27].

Another method that has been used to lower the per-pixel expense of mipmapping is to select a single mipmap level per triangle in each frame and rasterize the entire triangle using that mipmap level. While this is a very fast method, it can lead to serious visual artifacts, especially at the edges of triangles, where the mipmap level may change sharply. Software rasterizers that support mipmapping often use this method, known as *per-triangle mipmapping*.

Note that by its very nature, mipmapping tends to use smaller textures on distant objects. When used with software rasterizers, this means that mipmapping can actually *increase* performance, because the smaller mipmap levels are more likely to fit in the processor's cache than the full-detail texture. Most software rasterizers that support texturing are performance-bound to some degree by the memory bandwidth of reading textures. Keeping a texture in the cache can decrease these bandwidth requirements significantly. Furthermore, if point sampling is used with a non-mipmapped texture, adjacent pixels may require reading widely separated parts of the texture. These large per-pixel strides through a texture can result in horrible cache behavior and can impede the performance of non-mipmapped rasterizers severely. These cache miss stalls make the cost of computing mipmapping information (at least on a per-triangle basis) worthwhile, independent of the significant increase in visual quality. In fact, many hardware platforms also see performance increases when using mipmapping, owing to the small, on-chip texture cache memories used to hold recently used textures.

Texture Filtering and Mipmaps

The methods described above work on the concept that there will be a single, "best" mipmap level for a given pixel. However, since each mipmap level is twice the size of the next mipmap level in each dimension, the "closest" mipmap level may not be an exact pixel-to-texel mapping. Rather than selecting a given mipmap level as the best, linear mipmap filtering uses a method similar to linear texture filtering. Basically, mipmap filtering computes a real-valued L, which is used to find the pair of adjacent mipmap levels that bound the given pixel-to-texel size. The two adjacent mipmap levels that bound the pixel may be found using $\lfloor L \rfloor$ and $\lceil L \rceil$. The remaining fractional component is used to blend between texture colors found in the two mipmap levels.

Put together, there are now two independent filtering axes, each with two possible filtering modes, leading to four possible mipmap filtering modes as shown in Table 8.1. Of these methods, the most popular is *linear-bilinear*, which is also known as *trilinear interpolation* filtering, or *trilerp*, as it is the exact 3D analog to bilinear interpolation. It is the most expensive of these mipmap filtering operations, requiring the lookup of eight texels per pixel, as well as seven linear interpolations (three per each of the two mipmap levels, and one additional to interpolate between the levels), but it also produces the smoothest results. Filtering between mipmap levels also increases the

Table 8.1 Mipmap Filtering Modes

Mipmap filter	Texture filter	Result
Nearest	Nearest	Select "best" mipmap level and then select closest texel from it
Nearest	Bilinear	Select "best" mipmap level and then interpolate four texels from it
Linear	Nearest	Select two "bounding" mipmap levels, select closest texel in each, and then interpolate between the two texels
Linear	Bilinear	Select two "bounding" mipmap levels, interpolate four texels from each, and then interpolate between the two results

amount of texture memory bandwidth used, as the two mipmap levels must be accessed per sample. Thus, multilevel mipmap filtering often counteracts the aforementioned performance benefits of mipmapping on hardware graphics devices.

A final, newer form of mipmap filtering is known as *anisotropic* filtering. The mipmap filtering methods discussed thus far implicitly assume that the pixel, when mapped into texture space, produces a quadrilateral that is approximated quite well by a circle. In other words, the quadrilateral in texture space is basically square. In practice, this is often not the case. With polygons in extreme perspective, a pixel often maps to a very long, thin quadrilateral in texture space. The standard *isotropic* filtering modes can tend to look too blurry (having selected the mipmap level based on the long axis of the quad) or too sharp (having selected the mipmap level based on the short axis of the quad). Anisotropic texture filtering takes the aspect ratio of the texture-space quadrilateral into account when sampling the mipmap and is capable of filtering nonsquare regions in the mipmap to generate a result that accurately represents the tilted polygon's texturing. As of the writing of this text, anisotropic filtering is a common but not universal feature in consumer 3D hardware.

Mipmapping in OpenGL

SOURCE CODE DEMO Mipmap

The individual levels of a mipmap pyramid may be specified manually in OpenGL through the use of the `glTexImage2D` function described in the introduction to texturing (Chapter 6). However, in the case of mipmaps, the (previously ignored) second argument, `GLint level`, specifies the mipmap level. The mipmap level of the highest-resolution image is 0. Each subsequent level number (1, 2, 3 . . .) represents the mipmap pyramid image with half the dimensions of the previous level. OpenGL requires that a "full" pyramid be specified for mipmapping to work correctly. The number of mipmap levels in a full pyramid is equal to

$$Levels = \log_2(max(w_{texture}, h_{texture})) + 1$$

Note that the number of mipmap levels is based on the larger dimension of the texture. Once a dimension falls to 1 texel, it stays at 1 texel while the larger dimension continues to decrease. So, for a 32 × 8–texel texture, the mipmap levels are shown in Table 8.2.

Note that the texels of the mipmap level images passed to `glTexImage2D` must be computed by the application. OpenGL simply accepts these images as the mipmap levels and uses them directly. Once all of the mipmap levels for a texture are specified, `glBindTexture` may be used as before (see Chapter 6)

Table 8.2 Mipmap levels for a 32 × 8–texel texture

Level	Width	Height
0	32	8
1	16	4
2	8	2
3	4	1
4	2	1
5	1	1

to bind an identifier to the entire mipmap pyramid for later use. An example of specifying an entire pyramid directly follows.

```
char* texels0 = new unsigned char[16 * 16 * 4];
char* texels1 = new unsigned char[8 * 8 * 4];
char* texels2 = new unsigned char[4 * 4 * 4];
char* texels3 = new unsigned char[2 * 2 * 4];
char* texels4 = new unsigned char[1 * 1 * 4];
// fill texels0 with the image data
// filter the image data down into texels1-4
// ...

// the top-level 16x16 image
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 16, 16,
    0, GL_RGBA, GL_UNSIGNED_BYTE, texels0);

// the additional mipmap levels
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 8, 8,
    0, GL_RGBA, GL_UNSIGNED_BYTE, texels1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 4, 4,
    0, GL_RGBA, GL_UNSIGNED_BYTE, texels2);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 2, 2,
    0, GL_RGBA, GL_UNSIGNED_BYTE, texels3);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 1, 1,
    0, GL_RGBA, GL_UNSIGNED_BYTE, texels4);
```

As a convenience, OpenGL supports automatic filtering and creation of mipmap pyramids from a single image via the `gluBuild2DMipmaps` function. The function arguments are very similar to those of `glTexImage2D`, with the exception of the missing mipmap level and the loss of one other parameter (which we had ignored in the initial discussion of Chapter 6). After generating the pre-filtered mipmap data internally, `gluBuild2DMipmaps` calls the equivalent of `glTexImage2D` on each of the mipmap levels. The preceding code could be completely replaced with the following automatic mipmap generation:

```
char* texels = new unsigned char[16 * 16 * 4];
// fill texels with the image data
// ...

// the entire mipmap pyramid
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 16, 16,
    GL_RGBA, GL_UNSIGNED_BYTE, texels);
```

In order to set the minification method (or filter), `glTexParameteri` is called with a `setting` parameter of `GL_TEXTURE_MIN_FILTER`. OpenGL supports both non-mipmapped modes (bilinear filtering and nearest-neighbor selection), as well as all four mipmapped modes. The most common mipmapped mode (as described previously) is trilinear filtering, which is set using

```
// Trilinear filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
```

8.8 Blending

Thus far, this chapter has discussed generating pixel addresses that represent a triangle, as well as *source colors* that represent the color of the current triangle at those pixels. The reason that we have referred to these as "source colors" is that there is one more (optional) step in the rasterization pipeline, pixel blending (or more simply, blending). Pixel blending is sometimes referred to as alpha blending (which is really just a special case of general blending), because it often involves blending (or interpolating) between the existing color of the pixel and the new source color of the pixel based on the alpha value. At long last, blending brings to closure the path from model-space geometry to writing pixel colors *and* actually uses the alpha values we have been computing, interpolating, and carrying through the pipeline! However, as we shall see, pixel blending does not always use the alpha channel.

Once again, pixel blending is a per-pixel, nongeometric function that takes as its inputs the source color of the current triangle at the given pixel (which we will call C_{src}), the source alpha value (which is properly a component of the source color but which we will refer to as A_{src} for convenience), the current color of the pixel in the framebuffer (C_{dst}), and sometimes an existing alpha value in the framebuffer at that pixel (A_{dst}). These inputs, along with a pair of blending functions F_{src} and F_{dst}, define the final color (and potentially alpha value) that will be written to the pixel in the framebuffer, C_P. The general form of blending is

$$C_P = F_{src}C_{src} + F_{dst}C_{dst}$$

The simplest form of pixel blending is to disable blending entirely, which is equivalent to

$$\begin{aligned}
F_{src} &= 1 \\
F_{dst} &= 0 \\
C_P &= F_{src}C_{src} + F_{dst}C_{dst} = (1)C_{src} + (0)C_{dst} = C_{src}
\end{aligned}$$

Alpha blending is a blending mode that involves using the source alpha value A_{src} as the opacity of the new triangle and linearly interpolating between C_{src} and C_{dst} based on A_{src}:

$$\begin{aligned}
F_{src} &= A_{src} \\
F_{dst} &= (1 - A_{src}) \\
C_P &= F_{src}C_{src} + F_{dst}C_{dst} = A_{src}C_{src} + (1 - A_{src})C_{dst}
\end{aligned}$$

Alpha blending requires that C_{dst} be referenced. Because C_{dst} is stored in the framebuffer, alpha blending requires that the framebuffer be read for each pixel blended. This increased memory bandwidth means that alpha blending can impact performance on some systems (in a manner similar to z-buffering). In addition, alpha blending has several other properties that make its use somewhat challenging in practice.

Alpha blending is designed to compute a new color based on the idea that the source pixel color represents the color of a (possibly translucent) surface whose opacity is given by A_{src}. As a result, alpha blending only uses the alpha value of the source color, not the destination color. The destination color is assumed to be the "background," in front of which the translucent source surface is placed. For the following discussion, we will write alpha blending as

$$Blend(C_{src}, A_{src}, C_{dst}) = A_{src}C_{src} + (1 - A_{src})C_{dst}$$

The result of multiple alpha blending operations are order-dependent. Each alpha blending operation assumes that C_{dst} represents the final color of all objects that are seen through the current surface. If we view the blending of two possibly translucent surfaces (C_1, A_1) and (C_2, A_2) onto a background color C_0 as a sequence of two blends, we can quickly see that, in general, changing the order of blending changes the result. If we compare the two orders and expand the functions:

$$\begin{aligned}
Blend(C_2, A_2, Blend(C_1, A_1, C_0)) &\overset{?}{=} Blend(C_1, A_1, Blend(C_2, A_2, C_0)) \\
A_2C_2 + (1 - A_2)(A_1C_1 + (1 - A_1)C_0) &\overset{?}{=} A_1C_1 + (1 - A_1)(A_2C_2 + (1 - A_2)C_0) \\
A_2C_2 + (1 - A_2)(A_1C_1 + C_0 - A_1C_0) &\overset{?}{=} A_1C_1 + (1 - A_1)(A_2C_2 + C_0 - A_2C_0) \\
-A_1A_2C_1 &\overset{?}{=} -A_1A_2C_2 \\
A_1A_2C_1 &\overset{?}{=} A_1A_2C_2
\end{aligned}$$

These two sides are equal if and only if either $A_1 = 0$, $A_2 = 0$, or $C_1 = C_2$. Thus, unless one of these three cases is true, the two blending orders will produce different results. In visual terms, alpha blending of two surfaces with a background color is order-independent if and only if

1. One or more of the two surfaces are completely translucent (in which case they could be ignored, anyway).
2. Or, the two translucent surfaces have the same color as one another (in which case the two blending operations could have been combined into one).

In all other cases, we cannot switch the order of alpha blending operations.

8.8.1 Blending and Z-Buffering

In practice, order dependence when drawing alpha blended objects has significant effects on our visible-pixel algorithm, the z-buffer. Specifically, the z-buffer is based around the theory that a surface at a given depth will completely obscure any surface at the pixel that is at a greater depth. With opaque objects, this is true. However, in the presence of blending, it is not true, because a surface that is to be blended relies on the color of the surfaces behind it. In fact, the function in the preceding section is basically a recursive function. C_{dst} must represent the final combined color of all surfaces behind the current surface. This is true for each alpha blended surface. Thus, in the presence of alpha blending, we must compute the pixel color in a very specific ordering. Given a set of "surfaces" (i.e., colors and depths at the current pixel), the method of correctly coloring the pixel with alpha blending is as follows:

1. Compute the color and depth of the closest opaque surface (C_{opaque}, D_{opaque}). Set this color as the initial C_{dst} and the depth as the initial D_{dst}.

2. For each translucent surface $(C_{src}, A_{src}, D_{src})$ that is closer than D_{opaque} *in order of far to near depth*, set

$$C_{dst} = Blend(C_{src}, A_{src}, C_{dst})$$

In terms of our standard z-buffering method, this is implemented at the triangle level as

1. Collect the opaque triangles in the scene.
2. Collect the translucent triangles in the scene.
3. Render the opaque triangles normally, using z-buffering.
4. Sort the alpha blended triangles by depth into a far-to-near ordering.
5. Render the alpha blended triangles (in order) with blending, using z-buffering.

Actually, since the alpha blended triangles are rendered in back-to-front order, the z-buffer test in the final step is only to ensure that no alpha blended pixel whose depth is greater than the closest opaque triangle at that pixel will be drawn. Thus, the depth of the alpha blended triangles need not be written to the z-buffer. Many systems disable writing the z-buffer (but continue testing, of course) when writing alpha blended pixels, in order to decrease the required memory bandwidth. Since alpha blending already adds additional memory bandwidth requirements, this can be a very useful optimization.

8.8.2 Alternative Blending Modes

As we have mentioned, depth-sorting of triangles is expensive and does not always work without splitting triangles on a per-frame basis. If possible, we would like to avoid depth-sorting the blended triangles. One popular trick to avoid the sort is application-specific, but useful. If the blended objects can "glow" or "filter" rather than alpha blend with the scene, then one of a pair of other pixel-blending functions may be used. The two blending modes are known as *additive* and *modulate*. Additive implements glowing objects and is defined as follows:

$$F_{src} = 1$$
$$F_{dst} = 1$$
$$C_P = F_{src}C_{src} + F_{dst}C_{dst} = (1)C_{src} + (1)C_{dst} = C_{src} + C_{dst}$$

Note that this blending operation is clearly commutative and associative, and thus no sorting is required. Note further that no alpha channel is required for the effect.

Modulate blending implements color filtering. It is similar to additive blending in several ways and is defined as

$$
\begin{aligned}
F_{src} &= 0 \\
F_{dst} &= C_{src} \\
C_P &= F_{src}C_{src} + F_{dst}C_{dst} = (0)C_{src} + C_{src}C_{dst} = C_{src}C_{dst}
\end{aligned}
$$

This blending operation is also commutative and does not involve the alpha channel. Modulate blending is best known for creating so-called *darkmap* effects, where a textured object is drawn once with its main texture, and then the object is drawn again using modulate blending, this time with a texture that represents the lighting applied to the scene. Put together, these two rendering "passes" generate a far more complex effect, one of a detailed, textured surface (the first rendering pass, or *base map*) that is also lit by complex, subtle lighting effects (the second pass, or "darkmap"). Darkmaps are so named because the blending mode modulates the two passes, meaning that the resulting pixel color is always as dark or darker than either of the passes individually; thus, the second pass darkens the first pass — a "darkmap." Other blending effects are also possible.

Both additive and modulate blending modes still require the opaque objects to be drawn first, followed by the blended objects, but neither requires the blended objects to be sorted into a depthwise ordering. As a result, these blending modes (especially additive) are very popular with so-called particle systems, which involve rendering hundreds or thousands of small, blended triangles (generally to simulate smoke, water, or other natural phenomena) and could be far too computationally expensive to sort on a per-frame basis.

Note that if depth buffering is used with these blending modes, the blended objects (either additive or modulated) must be drawn with depth buffer *writing* disabled, or else any out of order (front to back) rendering of two blended objects will result in the more distant object not being drawn. If depth buffer writing is enabled, the closer of the two blended objects will write its depth value to the depth buffer first, and the more distant object will fail the depth buffer test. Again, disabling the depth buffer writes also offers the advantage of further increasing performance on some systems by avoiding additional memory write operations to the depth buffer.

8.8.3 Blending and OpenGL

Source Code
Demo
Blending

Blending is enabled and controlled quite simply in OpenGL, although there are many options beyond what we have discussed here. Enabling and disabling blending are accomplished through the use of `glEnable(GL_BLENDING)` and `glDisable(GL_BLENDING)`, respectively. The blending modes are set via the function `glBlendFunc`, which sets both F_{src} and F_{dst} in a single function call. To use classic alpha blending, the function call is

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Additive mode is set using the call

```
glBlendFunc(GL_ONE, GL_ONE);
```

and modulate blending may be used via the call

```
glBlendFunc(GL_ZERO, GL_SRC_COLOR);
```

This interface is very flexible and direct. There are far more blending functions available in OpenGL (although, in practice, some hardware devices may not be able to support all of them), and they are detailed in the *OpenGL Programming Guide* [83]. The three modes described here are the most common, with alpha blending being universal. Other, more esoteric combinations may not be supported. Note that while a hardware device may support all of the possible source and destination functions, it may not support all possible *combinations* thereof.

Recall that it is often useful to disable z-buffer writing while rendering blended objects. This is accomplished via depth-buffer "masking." A call to `glDepthMask(GL_TRUE)` will enable writing the z-buffer—this is the default setting in OpenGL. To disable writing the depth buffer, simply call `glDepthMask(GL_FALSE)`.

8.9 Antialiasing

In the absence of translucent objects, we have thus far discussed rasterizers with the assumption that a single triangle "wins" a pixel and determines its color. This is reasonable if we treat pixels as pure points, with no size. However, in our discussion of mipmapped textures, we saw that this is not the case; each pixel represents a rectangular region on the screen with a nonzero area. Because of this, more than one triangle may be visible inside of a pixel's rectangular region (just as more than one texel could fall within the region of a pixel). Figure 8.24 provides an example of such a pixel.

Using the point-sampled methods discussed, we will select a color sample from a single triangle to represent the entire area of the triangle.

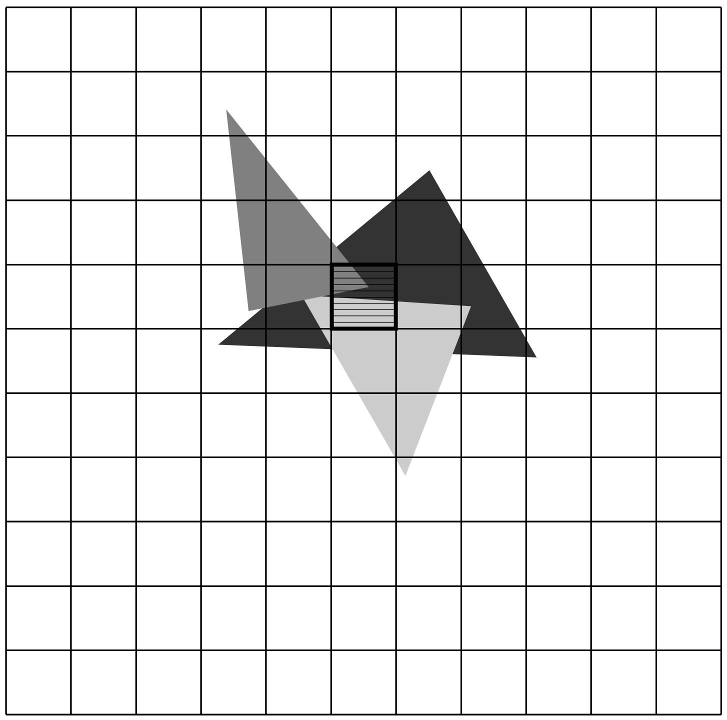


Figure 8.24 Multiple triangles falling inside the area of a single pixel.

According to the z-buffer method, whichever triangle has the closest depth value at the infinitesimal sample point (located at the center of the pixel) will win the pixel. However, as can be seen in Figure 8.25, this sample point may not represent the color of the pixel as a whole. In the figure, we see that most of the area of the pixel is dark gray, with only a very small square in the center being bright white. As a result, selecting a pixel color of bright white does not accurately represent the color of the pixel rectangle as a whole. Our perception of the color of the rectangle has to do with the relative areas of each color in the rectangle, something the point sampling method cannot represent.

Figure 8.26 makes this even more apparent. In this situation, we see two pixels. In both pixels, the vast majority of the surface area is dark gray. In each of the two pixels, there is a small white rectangle. The white rectangles are the same size in both triangles, but they are in slightly different positions in each of the two pixels. In each of the top examples, the white rectangle happens to contain the pixel center, while in the bottom cases, the white rectangle does not contain the pixel centers. To the right of each pixel's configuration is the color that will be assigned to that pixel. Very different colors are assigned to these two pixels, even though their geometric configurations are

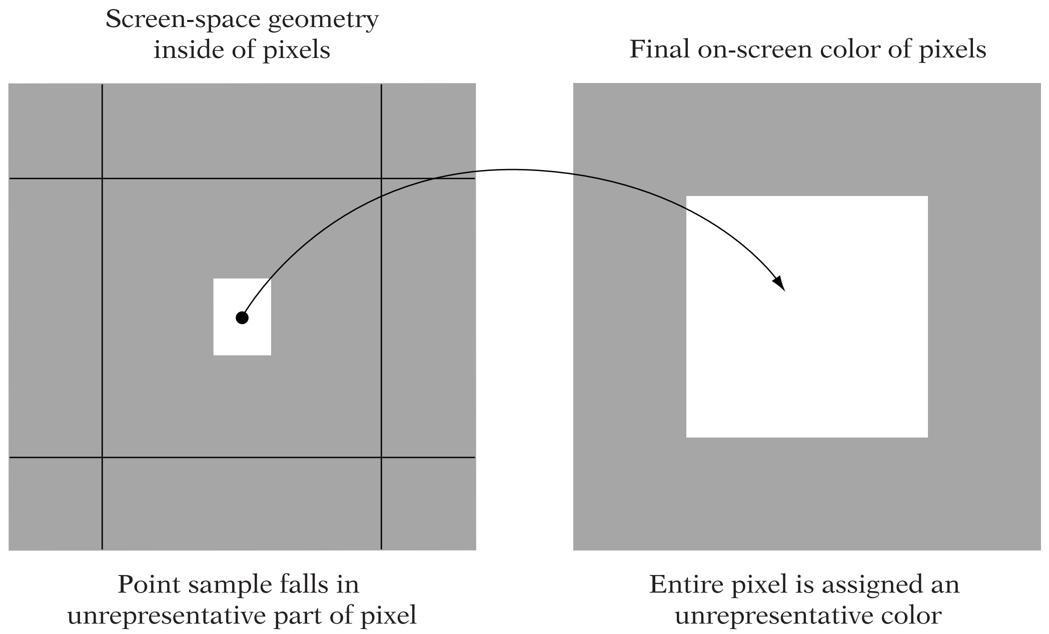

Screen-space geometry inside of pixels
Final on-screen color of pixels
Point sample falls in unrepresentative part of pixel
Entire pixel is assigned an unrepresentative color

FIGURE 8.25 A point sample may not accurately represent the overall color of a pixel.

almost identical. This demonstrates the fact that point sampling of the color of a pixel can lead to rather arbitrary results. In fact, if we imagine that the white rectangle were to move across the screen over time, the pixel would flash between white and gray as the white rectangle moved through the pixel center.

It is possible to determine a more accurate color for the two pixels in the figure. If the graphics system uses the relative areas of each color within the pixel's rectangle to weight the color of the pixel, the results will be much better. In Figure 8.27, we can see that the white rectangle covers approximately 10 percent of the area of the pixel, leaving the other 90 percent as dark gray. Weighting the color by the relative areas, we get a pixel color of

$$C_{area} = 0.1 \times (1.0, 1.0, 1.0) + 0.9 \times (0.25, 0.25, 0.25) = (0.325, 0.325, 0.325)$$

Note that this computation is independent of *where* the white rectangle falls within the pixel (assuming the white rectangle is entirely within the pixel). Such an area-based method avoids the point-sampling errors we have seen. Such a system can be extended to any number of different-colored areas within a given pixel. Given a pixel with area a_{pixel} and a set of n different subsections of the pixel (each generated by a piece of visible geometry that intersects the pixel's rectangle), each with an area within the pixel a_i and a

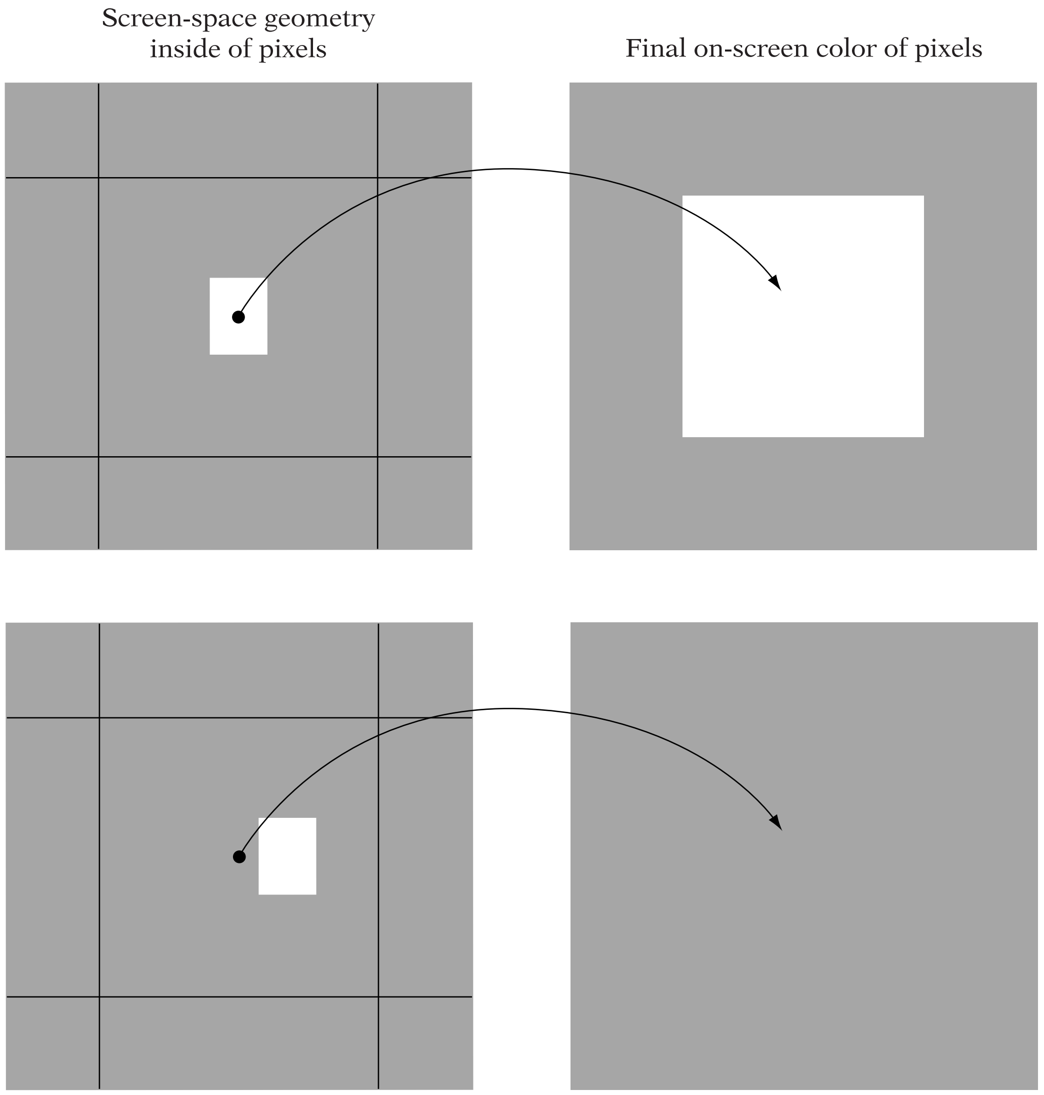

Screen-space geometry inside of pixels
Final on-screen color of pixels

FIGURE 8.26 Sub-pixel motion causing a large change in point-sampled pixel color.

color C_i, the final color of the pixel is then

$$\frac{\sum_{i=1}^{n} a_i \times C_i}{a_{pixel}} = \sum_{i=1}^{n} \frac{a_i}{a_{pixel}} \times C_i = \sum_{i=1}^{n} F_i \times C_i$$

where F_i is the fraction of the pixel covered by the given color, or the "coverage." This method is known as *area sampling*. In fact, this is really a

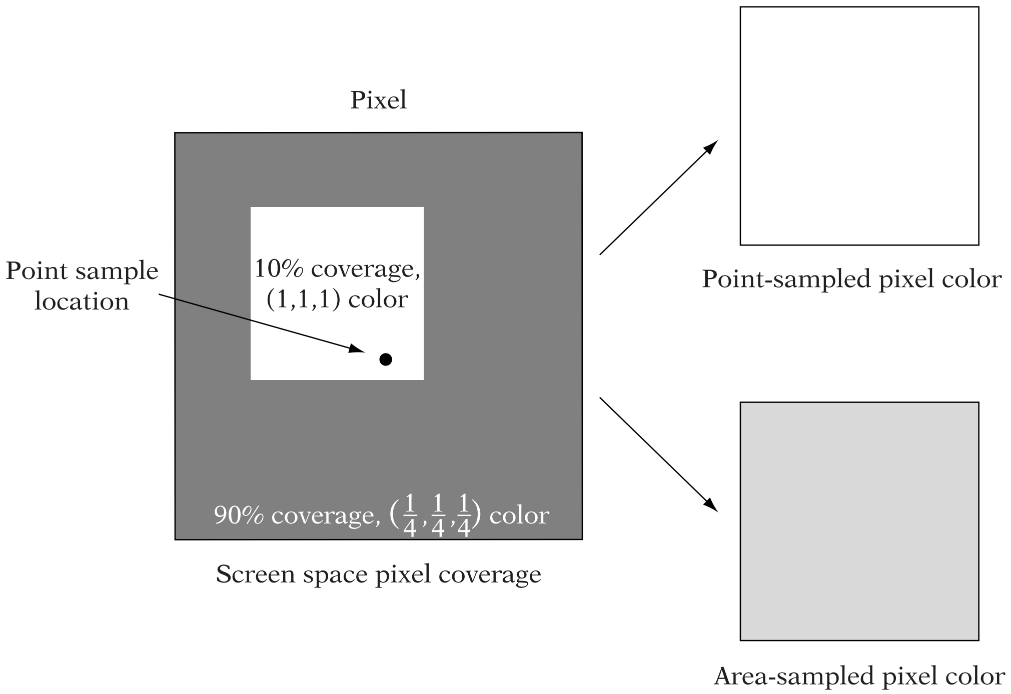

Pixel
Point sample location
10% coverage, (1,1,1) color
90% coverage, $(\frac{1}{4},\frac{1}{4},\frac{1}{4})$ color
Screen space pixel coverage
Point-sampled pixel color
Area-sampled pixel color

FIGURE 8.27 Area sampling of a pixel.

special case of a more general definite integral. If we imagine that we have a screen-space function that represents the color of every point on the screen (independent of pixels) $C(x, y)$, then the color of a pixel defined as the region $l \le x \le r, t \le y \le b$ (the left, right, top, and bottom screen coordinates of the pixel), then using this area sampling method is equivalent to

$$\frac{\int_t^b \int_l^r C(x, y) dx dy}{\int_t^b \int_l^r dx dy} = \frac{\int_t^b \int_l^r C(x, y) dx dy}{(b - t)(r - l)} = \frac{\int_t^b \int_l^r C(x, y) dx dy}{a_{pixel}} \tag{8.4}$$

which is the integral of color over the pixel's area, divided by the total area of the pixel. The summation version of equation 8.4 is a simplification of this more general integral, using the assumption that the pixel consists entirely of areas of piecewise constant color.

As a verification of this method, we shall assume that the pixel is entirely covered by a single triangle with fixed color $C(x, y) = C_T$, giving

$$\frac{\int_t^b \int_l^r C(x, y) dx dy}{a_{pixel}} = \frac{\int_t^b \int_l^r C_T dx dy}{a_{pixel}} = C_T \frac{\int_t^b \int_l^r dx dy}{a_{pixel}} = C_T \frac{a_{pixel}}{a_{pixel}} = C_T \tag{8.5}$$

which is the color we would expect in this situation.

While area sampling does avoid completely missing or overemphasizing any single sample, it is not the only method used, nor is it the best at representing the realities of display devices. The area sampling shown in equation 8.5 implicitly weights all regions of the pixel equally, giving the center of the pixel weighting equal to that of the edges. As a result, it is often called *unweighted area sampling*. *Weighted area sampling*, on the other hand, adds a weighting function that can bias the importance of the colors in any region of the pixel as desired. If we simplify the original pixel boundaries and the functions associated with equation 8.4 such that boundaries of the pixel are $0 \leq x, y \leq 1$, then equation 8.4 becomes

$$\frac{\int_t^b \int_l^r C(x, y)dxdy}{\int_t^b \int_l^r dxdy} = \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{1} \tag{8.6}$$

Having simplified equation 8.4 into equation 8.6, we define a weighting function $W(x, y)$ that allows regions of the pixel to be weighted as desired:

$$\frac{\int_0^1 \int_0^1 W(x, y)C(x, y)dxdy}{\int_0^1 \int_0^1 W(x, y)dxdy} \tag{8.7}$$

In this case, the denominator is designed to normalize according to the weighted area. A similar substitution to equation 8.5 shows that constant colors across a pixel map to the given color. Note also that (unlike unweighted area sampling) the position of a primitive within the pixel now matters. From equation 8.7, we can see that unweighted area sampling is simply a special case of weighted area sampling. With unweighted area sampling, $W(x, y) = 1$, giving

$$\begin{aligned}
&\frac{\int_0^1 \int_0^1 W(x, y)C(x, y)dxdy}{\int_0^1 \int_0^1 W(x, y)dxdy} \\
&= \frac{\int_0^1 \int_0^1 (1)C(x, y)dxdy}{\int_0^1 \int_0^1 (1)dxdy} \\
&= \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{\int_0^1 \int_0^1 dxdy} \\
&= \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{1}
\end{aligned}$$

A full discussion of weighted area sampling, the theory behind it, and numerous common weighting functions is given in [36]. For those desiring more depth, [120] and [41] detail a wide range of sampling theory.

8.9.1 Antialiasing in Practice

The methods so far discussed show theoretical ways for computing area-based pixel colors. These methods require that pixel-coverage values be computed per triangle, per pixel. Computing analytical (exact) pixel coverage values for triangles can be complicated. In practice, the pure area-based methods do not lead directly to simple, fast hardware antialiasing implementations.

Consumer 3D hardware is almost universally based upon the point sampling methods discussed earlier in this chapter. It is only natural that hardware developers would seek to create antialiasing-capable hardware that did not require entirely new, area-based sampling techniques. The most popular form of antialiasing on consumer hardware is based on sampling at multiple points inside of each pixel. This is known as *multisample antialiasing*. Area-based sampling is approximated by point sampling the scene at as few as two samples per pixel and as many as 16 or more samples per pixel. Figure 8.28 shows some sample patterns. Each square represents a single pixel. Filled (dark) circles represent the locations of rendered (rasterized) sample points in the pixel. Unfilled (white) sample points in the figure are

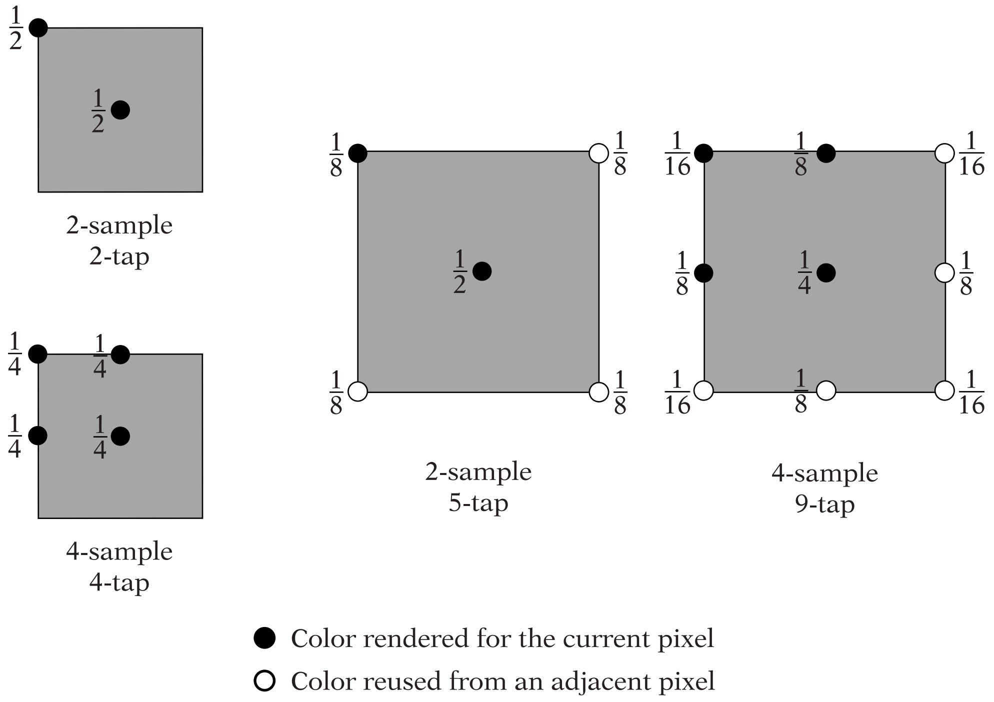


Figure 8.28 Common sample-point distributions for multisample-based antialiasing.

actually samples taken from adjacent pixels, reused for the current pixel. In common multisample antialiasing nomenclature, a "sample" refers to a color that is computed by actually rasterizing triangles at the current pixel, while a "tap" refers to a more general notion—a color that may either be a sample rendered for the current pixel or a sample that is simply reused from another pixel. All configurations with more "taps" than "samples" are reusing samples from other pixels as additional (low-cost) taps. As a result, it is the number of samples that best represents the rasterization expense for a given configuration. Reusing samples from other pixels as taps for a given pixel gives *some* of the benefits of a higher number of samples per pixel without the rasterization expense of additional per-pixel samples.

The colors of each of these samples are combined into a single pixel color via a weighted (or in some cases unweighted) sum. Common weights used with weighted-area versions of these sampling patterns are also shown in Figure 8.28.

Some systems that support these subpixel samples support two forms of multi-sample antialiasing. The first is automatic multi-sample antialiasing; a simple, easy-to-use system (essentially the one just described) that automatically places and renders the subpixel samples and then sums them into the final pixel color. The other mode is a manual mode, in which the application can choose to "mask" (i.e., disable) some of the samples and render colors to as few as one per-pixel sample per rendering "pass." This latter system requires the application to deal with the setup and rendering of each pass, but can allow for incredible flexibility. In fact, this manual mode can allow antialiasing in multiple dimensions, including

- Temporal sampling. The samples in a given pixel are each rendered at different values of "game time," with the scene and camera animated between each sample. This simulates motion blur, by causing the samples to represent the time over which the camera's "shutter" is open.
- Optical sampling. This is done by taking the samples from multiple, slightly different camera positions, which represent the centers of projection on the surface of a lens. The camera matrix is chosen such that points on the focal plane of the camera (a fixed distance into the scene) are the same across all subpixel samples. The resulting image will look perfectly sharp for objects at the focal plane and increasingly blurry away from the focal plane.
- Area lights. For systems that can render sharp shadows based on some lights, soft shadows can be created by rendering each subpixel sample with the exact position of the shadow-generating light shifted slightly from the other samples. In this way, once all of the samples are

rendered, surface points in the umbras of shadows will be the darkest (as they are in shadow for all of the jittered light positions), and surface points in the penumbrae of shadows will be lighter (as they will be in shadow from only some of the jittered light positions).

8.9.2 Antialiasing in OpenGL

OpenGL (in its unextended form) supports two forms of antialiasing; *pixel-coverage antialiasing* and the *accumulation buffer*. The first (and older) of these two, pixel-coverage antialiasing, is based on the direct computation (by the OpenGL implementation) of fractional pixel-coverage values for each triangle, per pixel. These fractional coverage values are analogous to the F_i values defined in Section 8.9. During rendering, these pixel-coverage values are used as alpha values in a pixel blending operation. Each triangle is blended with the existing pixel color, according to its coverage value.

Pixel-coverage antialiasing uses pixel blending, meaning that alpha blending and other such effects cannot be used simultaneously with this method of antialiasing. As with other operations involving pixel blending, depth buffering cannot resolve the visible surfaces correctly when using pixel-coverage antialiasing. The geometry must be rendered in back-to-front order manually, using some form of geometric sorting.

Pixel-coverage antialiasing is unsuitable for most modern, complex scenes, owing to the fact that it requires depth sorting of geometry and is incompatible with alpha blending. Readers interested in learning the details of using pixel-coverage antialiasing in OpenGL should read the *OpenGL Programming Guide* [83].

Owing to its heritage as an API for high-end graphics workstations, OpenGL also includes built-in support for a rather advanced form of antialiasing known as an *accumulation buffer*, or *a-buffer* (see [52] and [83] for details). While consumer 3D hardware support for a-buffers is far from universal, the concept of an a-buffer is worth discussing, owing to its powerful, general nature. Basically, the accumulation buffer is simply an extra, off-screen framebuffer. Accumulation buffers generally use higher-resolution color components (e.g., 10–16 bits per component) than the main framebuffer, to avoid color quantization artifacts. There are four basic operations with an a-buffer:

- Clear the a-buffer pixels to a color (`glClear(GL_ACCUM_BUFFER_BIT)`)
- Copy the framebuffer pixels (multiplied by a floating-point constant, `mult`) in the a-buffer (`glAccum(GL_LOAD, mult)`)
- Add the framebuffer pixels (multiplied by a floating-point constant, `mult`) in the a-buffer (`glAccum(GL_ACCUM, mult)`)

- Copy the a-buffer pixels (multiplied by a floating-point constant, `mult`) back into the framebuffer (`glAccum(GL_RETURN, mult)`)

This set of operations is deceptively simple and allows an immense range of options. For example, multisample antialiasing using N samples per pixel can be computed with the a-buffer as follows:

```
// Clear the a-buffer
glClear(GL_ACCUM_BUFFER_BIT);

// For subpixel samples 1 <= i <= N
for (unsigned int i = 1; i <= N; i++)
{
    // Clear the framebuffer and z-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Move the camera to subpixel sample position i
    // Render the scene to the framebuffer
    // ...

    // Accumulate the framebuffer into the a-buffer,
    // scaled by 1/N
    glAccum(GL_ACCUM, 1.0f/N);
}
// Read back the a-buffer into the framebuffer
glAccum(GL_RETURN, 1.0f);
// Display the framebuffer
// ...
```

Basically, this pseudocode renders the entire scene *N* times, once from each of *N* (slightly shifted) camera positions. These camera positions represent the positions of the subpixel samples at each pixel. So, if we wish to render a 3×3 grid of subpixel samples, we would render the scene nine times, computing a single subpixel sample for all pixels in each rendering.

The semantics of a-buffering make a-buffers expensive for 3D hardware to implement. The additional high-precision buffer requires considerable additional framebuffer memory. In addition, the required copy operations from framebuffer to a-buffer and back are computationally expensive. Most current hardware devices simply cannot support the accumulation buffer interfaces at high performance, supporting only multisample antialiasing. Hardware vendors have made other methods of antialiasing (especially so-called "single pass" methods that render all subpixel samples for the entire screen in a single rendering pass) available via OpenGL extensions such as

`GL_ARB_multisample` and via Microsoft's Direct3D multisample pixel formats. These multi-sample implementations can often support both automatic subpixel spatial antialiasing and the more complex multipass effects using sample masking. The developer Web sites of the popular 3D hardware vendors ([6], [82]) include detailed discussions of their devices' support for these features in both rendering APIs.

8.10 Chapter Summary

This chapter concludes the discussion of the rendering pipeline. Rasterization provides us with some of the lowest-level yet most mathematically interesting concepts in the entire pipeline. We have discussed the connections between mathematical concepts such as projective transforms and rendering methods such as perspective-correct texturing. In addition, we addressed issues of mathematical precision in our discussion of the depth buffer. Finally, the concept of point sampling versus area sampling appeared twice, relating to both mipmapping and antialiasing. Whether it is implemented in hardware, software, or a mixture of the two, the entire graphics pipeline is ultimately designed only to feed a rasterizer, making the rasterizer one of the most important yet least understood pieces of rendering technology.

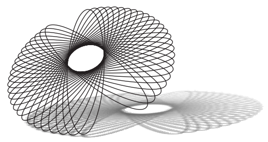
Thanks to the availability of high-quality, low-cost 3D hardware on a wide range of platforms, the percentage of readers who will ever have to implement their own rasterizer is quite small. However, an understanding of how rasterizers function is important even to those who will never need to write one. For example, even a basic practical understanding of the z-buffering system can help a programmer build a scene that avoids visual artifacts during visible surface determination. Understanding the inner workings of rasterizers can help a 3D programmer quickly debug problems in the geometry pipeline. Finally, this knowledge can guide the programmer to better optimize their geometry pipeline, "feeding" their rasterizer with high-performance datasets.

For further reading, we recommend the *OpenGL Programming Guide* [83], which details many more features of OpenGL, especially as they relate to rasterizing, texturing, and antialiasing. As referenced numerous times in this and other chapters, Chris Hecker's series on perspective texture mapping [60] is an excellent introduction to the many details that must be considered when designing a high-performance software rasterization system.

Part III

Animation

Chapter 9 Curves



9.1 Introduction

Up to this point, we have considered only motion (more specifically, transformations) that has been created programmatically. In order to create a particular motion (e.g., a submarine moving through the world), we have to write a specific program to generate the appropriate sequence of transformations for our model. However, this takes time and it can be quite tedious to move objects in this fashion. It would be much more convenient to predefine our transformation set in a tool and then somehow regenerate it within our game. An artist could create the sequence using a modeling package, and then a programmer would just write the code to play it back, much as a projector plays back a strip of film. This process of pregenerating a set of data and then playing it back is known as *animation*.

The best way to understand animation is to look at the art form in which it has primarily been used: motion pictures. In this case, the illusion of motion is created by drawing or otherwise recording a series of images on film and then projecting them at 24 or 30 frames per second (for film and video, respectively). The illusion is maintained by a property of the eye-brain combination known as persistence of motion: the eye-brain system sees two frames and invisibly (to our perception) fills in the gaps between them, thus giving us the notion of smooth motion.

We could do something similar in our game. Suppose we had a character that we want to move around the world. The artist could generate various animation sets at 60 frames per second (f.p.s.), and then when we want the character to run, we play the appropriate running animation. When we want the character to walk, we switch to the walking animation. The same process can be used for all the possible motions in the game.

However, there are a number of problems with this. First, by setting the animation set to a rate of 60 frames per second and then playing it back directly, we have effectively locked the frame rate for the game at 60 f.p.s. as well. Many monitors can run at 85 f.p.s., and when running in windowed mode, the graphics can be updated much faster than that. It would be much better if we could find some way to generate 85 f.p.s. or more from a 60 f.p.s. dataset. In other words, we need to take our initial dataset and generate a new one at a different rate. This is known as *resampling*.

This brings us to our second problem. Storing 60 f.p.s per animation adds up to a lot of data. As an example, if we have 10 data points per model that we're storing, with 16 floats per point (i.e., a 4×4 matrix), that adds up to about 38 KB per second of animation. A minute of animation adds up to over 2 MB of data, which can be a serious hit, particularly if we're running on a low-memory platform such as a console. It would be better if we could generate our data at a lower rate, say 10 or 15 f.p.s., and then resample up to the speed we need. This is essentially the same problem as our first one — it's just that our initial data set has fewer samples.

Alternately, we could take another cue from movie animation. The primary animators on a film draw only the important, infrequent "key" frames that capture the essential flow of an animation. The work of generating the remaining "in between" frames is left to secondary animators, who generate these intermediate frames from the supplied key frames. These artists are known as 'tweeners. In our case, we could store key frames that store the essential positions of our motion. These key frames would not have to be separated by a constant time interval, but at smaller intervals when the positions are changing quickly, and at larger intervals when the positions change very slowly. The resampling function would act as our 'tweener for this key frame data.

Fortunately, we have already been introduced to one technique for doing all of this, albeit in another form. This method is known as interpolation, and we first saw it when generating a line from two points. Interpolation takes a set of discrete sample points at given time intervals and generates a continuous function that passes through the points. Using this, we can pick any time along the domain of the function and generate a new point so that we might fill in the gaps. We're using the interpolation function to sample at a different rate.

An alternative is approximation, which uses the points to guide the resulting function. In this case the function does not pass through the points. This may seem odd, but it can help us better control the shape of the function. However, the same principle applies: we generate a function based on the initial sample data and resample later at a different frame rate. The general class of functions we'll be using for both interpolating and approximating are called parametric curves.

9.2 General Definitions

A *parametric curve* is a function $Q(u)$ that maps a set of real values (represented by the parameter u) to a set of points. The derivative $\mathbf{Q}'(u)$ for parameter u is a tangent vector to the curve at location $Q(u)$. When mapping to $\mathbb{R}^3$, we commonly use a parametric curve broken into three separate functions, one for each coordinate: $Q(u) = (x(u), y(u), z(u))$. This is also known as a *space curve*. The derivative of a space curve is $\mathbf{Q}'(u) = (x'(u), y'(u), z'(u))$.

When curves are used for animation, the parameter u or t usually represents time, although the units used don't necessarily have any relationship to seconds. In our discussion we will often use u as the parameter to a normalized curve such that $Q(0)$ is the start of the curve and $Q(1)$ is the end. When we want to use a general parameterization, we will refer to the parameter t. In this case we usually set a time value t_i for each point P_i; we expect to end up at position P_i in space at time t_i. The sequence $t_0, t_1, \dots, t_n$ is sorted (as are the corresponding points) so that it is monotonically increasing.

The average *speed* r we travel along a curve is related to the distance d traveled along the curve and the time it takes to travel that distance, namely,

$$r = d/u$$

The instantaneous speed at a particular parameter u is the length of the derivative vector $\mathbf{Q}'(u)$.

For a given point P on a smooth curve $Q(u)$, we define a circle with first and second derivative vectors equal to those at P as the osculating[1] circle. If the radius of the osculating circle is ρ, the *curvature* κ at P is $1/\rho$. The curvature at any point is always nonnegative. The higher the curvature, the more the curve bends at that point; the curvature of a straight line is 0.

In general, it is not practical to construct a single, closed form polynomial that uses all of the sample points — most of the curves we will discuss use at most four points as their geometric foundation. Instead, we will create curve segments that each apply over a sequential subset of the points and join these segments together to create a function across the entire domain. How we create this joint determines the type of continuity we will have in our function.

Formally, we say that a function f is *continuous* at a value x_0 if

$$\lim_{x \to x_0} f(x) = f(x_0)$$

In addition, we say that a function $f(x)$ is continuous over an interval (a, b) if it is continuous for every value x in the interval. We can also say that the function has positional, or C^0, continuity over the interval (a, b). Informally, we can

1. So called because it "kisses" up to the point.

think of a continuous function as one that we can draw without ever lifting the pen from the page. When using curve segments, we can achieve C^0 continuity by ensuring that the end point of one curve segment is equal to the start point of the next segment.

This can be taken further: a function $f(x)$ has tangential, or C^1, continuity across an interval (a, b) if the first derivative $f'(x)$ of the function is continuous across the interval. We can achieve C^1 continuity when using curve segments by guaranteeing that tangent vectors are equal at the end of one segment and the start of the next segment. A related form of continuity is G^1 continuity, where the tangents at each segment are not necessarily equal but point in the same direction. In many cases G^1 continuity is good enough for our purposes.

Occasionally, we may be concerned with C^2 continuity, also known as curvature continuity. A function $f(x)$ has C^2 continuity across an interval (a, b) if the second derivative $f''(x)$ of the function is continuous across the interval. Higher orders of continuity are possible, but they are not relevant to the discussion that follows.

9.3 Linear Interpolation

9.3.1 Definition

The most basic parametric curve is a line passing through two points. By using the parameterized line equation based on the two points, we can generate any point along the line. This is known as *linear interpolation* and is the most commonly used form of interpolation in game programming, mainly because it is the fastest. From our familiar line equation

$$Q(u) = P_0 + u(P_1 - P_0)$$

we can rearrange to get

$$Q(u) = (1 - u)P_0 + uP_1$$

The value u is the factor we use to control our interpolation, or parameter. Recall that if u is 0, $Q(u)$ returns our starting point P_0, and if u is 1, then $Q(u)$ returns P_1, our end point. Values of u between 0 and 1 will return a point along the line segment $\overline{P_0P_1}$. When interpolating, we usually care only about values of u within the interval [0, 1] and, in fact, state that the interpolation is undefined outside of this interval.

It is common when creating parametric curves to represent them as matrix equations. As we'll see next, it makes it simple to set certain conditions for a curve and then solve for the equation we want. The standard matrix form is

$$Q(u) = \mathbf{U} \cdot \mathbf{M} \cdot \mathbf{G}$$

where **U** is a row matrix containing the polynomial interpolants we're using: 1, u, u^2, u^3, and so on; **M** is a matrix containing the coefficients necessary for the parametric curve; and **G** is a matrix containing the coordinates of the geometry that defines the curve. In the case of linear interpolation

$$\mathbf{U} = \begin{bmatrix} u & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \end{bmatrix}$$

With this formulation, the result **UMG** will be a 1×3 matrix:

$$\begin{aligned} \mathbf{UMG} &= \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} \\ &= \begin{bmatrix} (1-u)x_0 + ux_1 & (1-u)y_0 + uy_1 & (1-u)z_0 + uz_1 \end{bmatrix} \end{aligned}$$

This is counter to our standard convention of using column vectors. However, rather than write out **G** as individual coordinates, we can write **G** as a column matrix of n points, where for linear interpolation this is

$$\mathbf{G} = \begin{bmatrix} P_0 \\ P_1 \end{bmatrix}$$

Then, using block matrix multiplication, the result **UMG** becomes

$$\mathbf{UMG} = (1-u)P_0 + uP_1$$

This form allows us to use a convenient shorthand to represent a general parameterized curve without having to expand into three essentially similar functions.

Recall that in most cases we are given time values t_0 and t_1 that are associated with points P_0 and P_1, respectively. In other words, we want to start at point P_0 at time t_0 and end up at point P_1 at time t_1. These times are not necessarily 0 and 1, so we'll need to remap our time value t in the interval $[t_0, t_1]$ to a parameter u in the interval [0, 1], which we'll use in our original interpolation equation. If we want the percentage u that a time value t lies between t_0 and t_1, we can use the formula

$$u = \frac{t - t_0}{t_1 - t_0} \tag{9.1}$$

Using this parameter u with the linear interpolation will give us the effect we desire. We can use this approach to change any curve valid over the interval [0, 1] using u as a parameter to be valid over $[t_0, t_1]$ using t as a parameter.

9.3.2 Piecewise Linear Interpolation

Source Code
Demo
Linear

Pure linear interpolation works fine if we have only two values, but in most cases we will have many more than two. How do we interpolate among multiple points?

The simplest method is to linearly interpolate from the first point to the second, then from the second point to the third, and so on, until we get to the end. For each pair of points P_i and P_{i+1}, we use equation 9.1 to adjust the time range $[t_i, t_{i+1}]$ to [0, 1] so we can interpolate properly.

For a given time value t, we need to find the stored time values t_i and t_{i+1} such that $t_i \le t \le t_{i+1}$. From there we look up their corresponding P_i and P_{i+1} values and interpolate. If we start with $n + 1$ points, we will end up with a series of n segments labeled $Q_0, Q_1, \ldots, Q_{n-1}$. Each Q_i is defined by points P_i and P_{i+1} where

$$Q_i(u) = (1 - u)P_i + uP_{i+1}$$

and $Q_i(1) = Q_{i+1}(0)$. This last condition guarantees C^0 continuity.

Expressed as code:

```
IvVector3 EvaluatePiecewiseLinear( float t, unsigned int count,
                                   const IvVector3* positions,
                                   const float* times)
{
    // handle boundary conditions
    if ( t <= times[0] )
        return positions[0];
    else if ( t >= times[count-1] )
        return positions[count-1];

    // find segment and parameter
    unsigned int i;
    for ( i = 0; i < count-1; ++i )
    {
```

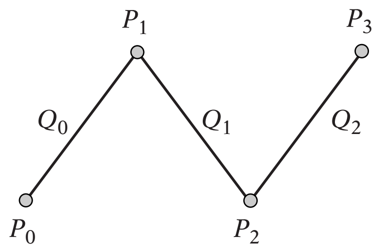


Figure 9.1 Piecewise linear interpolation.

```
        if ( t < times[i+1] )
            break;
    }
    float t0 = times[i];
    float t1 = times[i+1];
    float u = (t - t0)/(t1 - t0);

    //evaluate
    return (1-u)*positions[i] + u*positions[i+1];
}
```

In the pseudocode we found the subcurve by using a straight linear search. For large sets of points, using a binary search will be more efficient since we'll be storing the values in sorted order. We can also use temporal coherence: since our time values won't be varying wildly and will be increasing in value, we can first check whether we lie in the interval $[t_i, t_{i+1}]$ from the last frame and then check subsequent intervals.

This works reasonably well and is quite fast, but as Figure 9.1 demonstrates, will lead to sharp changes in direction. If we treat the piecewise interpolation of $n+1$ points as a single function $f(t)$ over $[t_0, t_n]$, we find that the derivative $f'(t)$ is discontinuous at the sample points, so $f(t)$ is not C^1 continuous. In animation this expresses itself as sudden changes in the speed and direction of motion, which may not be desirable. Despite this, because of its speed, piecewise linear interpolation is a reasonable choice if the slopes of the piecewise line segments are relatively close. If not, or if smoother motion is desired, other methods using higher order polynomials are necessary.

9.4 Lagrange Polynomials

Source Code
Demo
Lagrange

One way to create smoother motion is to generate a polynomial function that will pass through every point. So if we have three sample points, we will generate a quadratic function; if we have four sample points, a cubic function; and so on. The most common method to generate such a curve is to use a set of generalized functions known as *Lagrange polynomials*. They allow us to take a set of any $n+1$ points $P_0, \ldots, P_n$, along with their corresponding time values $t_0, \ldots, t_n$, and construct an n-degree polynomial. For example, if we have two points, the corresponding Lagrange polynomial is a first-degree polynomial or a line, as we expect. If we have three noncollinear points, we can create a quadratic equation that passes through all three points.

The general form of the Lagrange polynomial is

$$P(t) = \sum_{k=0}^{n} P_k L_{n,k}(t)$$

where

$$\begin{aligned} L_{n,k}(t) &= \frac{(t-t_0)(t-t_1)\cdots(t-t_{k-1})(t-t_{k+1})\cdots(t-t_n)}{(t_k-t_0)(t_k-t_1)\cdots(t_k-t_{k-1})(t_k-t_{k+1})\cdots(t_k-t_n)} \\ &= \prod_{i=0, i\neq k}^{n} \frac{(t-t_i)}{(t_k-t_i)} \end{aligned} \tag{9.2}$$

Equation 9.2 is known as the *Lagrange product*. Let's take a closer look. For the kth equation, if we substitute t_k for t, we get

$$\begin{aligned} L_{n,k}(t_k) &= \frac{(t_k-t_0)(t_k-t_1)\cdots(t_k-t_{k-1})(t_k-t_{k+1})\cdots(t_k-t_n)}{(t_k-t_0)(t_k-t_1)\cdots(t_k-t_{k-1})(t_k-t_{k+1})\cdots(t_k-t_n)} \\ &= 1 \end{aligned}$$

Otherwise, if we substitute t_k for t in any of the other Lagrange products, we get

$$\begin{aligned} L_{n,j}(t_k) &= \frac{(t_k-t_0)\cdots(t_k-t_k)\cdots(t_k-t_{j-1})(t-t_{j+1})\cdots(t-t_n)}{(t_j-t_0)\cdots(t_j-t_k)\cdots(t_j-t_{j-1})(t_j-t_{j+1})\cdots(t_j-t_n)} \\ &= 0 \end{aligned}$$

So for a given t_k, $P(t_k)$ returns P_k, which is what we expect.

If we have two points, the corresponding Lagrange polynomial is

$$P(t) = \frac{(t-t_1)}{(t_0-t_1)} P_0 + \frac{(t-t_0)}{(t_1-t_0)} P_1$$

If our two points are at time values $t_0 = 0$ and $t_1 = 1$, then

$$\begin{aligned} P(t) &= \frac{(t-1)}{-1} P_0 + \frac{(t-0)}{1} P_1 \\ &= (1-t)P_0 + tP_1 \end{aligned}$$

So the Lagrange polynomial with two points is our standard linear interpolation formula.

Three points gives us the following equation:

$$P(t) = \frac{(t-t_1)(t-t_2)}{(t_0-t_1)(t_0-t_2)}P_0 + \frac{(t-t_0)(t-t_2)}{(t_1-t_0)(t_1-t_2)}P_1 + \frac{(t-t_0)(t-t_1)}{(t_2-t_0)(t_2-t_1)}P_2 \qquad (9.3)$$

Substituting our time values for each point and simplifying the equation generates a quadratic equation that will interpolate from P_0 to P_1 to P_2.

This works fine for small numbers of points. But suppose we have a larger dataset of, say, 23 points and time values. Or a number of different datasets, each with different numbers of sample points and times. One possibility would be to generate the Lagrange equation for each data set and then simplify to a less complicated equation. If our data is fixed, this works fine, but usually animators will tweak their values throughout the development of a game. An animation may change in time value or in the number of sample points. If this happens, the entire Lagrange equation is invalid and would have to be recalculated.

Based on the assumption that our data is going to be changing frequently, we could use the generalized Lagrange equation, but that would involve at least 22 multiplications and subtractions per sample point, for 23 samples, or 506 multiplications and subtractions total. This is not very efficient and would grow worse with more points.

Lagrange polynomials have other issues that make them impractical for our purposes. An animator can't adjust the curve other than by moving points or adjusting t values, which is both unwieldy and inflexible. And when interpolating large numbers of points, the curve tends to oscillate in order to maintain continuity and pass through every point. Lagrange polynomials also run into numerical problems with larger and larger numbers of points. Because of this, they are fine for interpolating small datasets, but other methods are more useful for real animation data.

9.5 Hermite Curves

9.5.1 Definition

Source Code
Demo
Hermite

Clearly, trying to build a single parametric curve by using all of the points is not going to be a productive method. Instead, let's return to the idea of piecewise equations. But this time, instead of using piecewise linear equations, which give us discontinuities in the derivative at the sample points, we will use higher-order equations, in particular cubic curves. If we control the curve properly at each point, then we can smoothly transition from one point to the

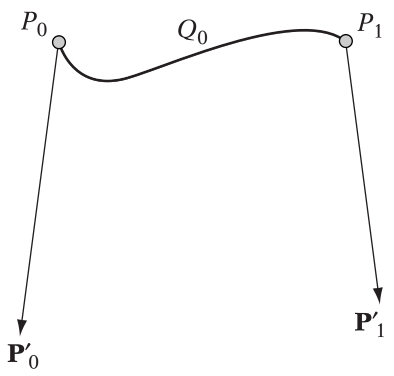


FIGURE 9.2 Hermite curve.

next, avoiding the obvious discontinuities. In particular, what we want to do is to set up our piecewise curves so that the tangent at the end of one curve matches the tangent at the start of the next curve. This will remove the first order discontinuity at each point — the derivative will be continuous over the entire time interval that we are concerned with.

Why a cubic curve and not a quadratic curve? Take a look at Figure 9.2. We have set two positions P_0 and P_1, and two tangents $\mathbf{P}'_0$ and $\mathbf{P}'_1$. Clearly, a line won't pass through the two points and also have a derivative at each point that matches its corresponding tangent vectors. The same is true for a parabola. The next order curve is cubic, which will satisfy these conditions. Intuitively, this makes sense. A line is constrained by two points, or one point and a vector. A parabola can be defined by three points, or by two points and a tangent. And a cubic curve can be defined by four points, or two points and two tangents.

Using our given constraints, or *boundary conditions*, let's derive our cubic equation. A generalized cubic function and corresponding derivative are

$$Q(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + D \tag{9.4}$$

$$\mathbf{Q}'(u) = 3\mathbf{a}u^2 + 2\mathbf{b}u + \mathbf{c} \tag{9.5}$$

We'll solve for our four unknowns $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and D by using our four boundary conditions. We'll assume that when $u = 0$, $Q(0) = P_0$ and $\mathbf{Q}'(0) = \mathbf{P}'_0$. Similarly, at $u = 1$, $Q(1) = P_1$ and $\mathbf{Q}'(1) = \mathbf{P}'_1$. Substituting these values into equations 9.4 and 9.5, we get

$$Q(0) = D = P_0 \tag{9.6}$$

$$Q(1) = \mathbf{a} + \mathbf{b} + \mathbf{c} + D = P_1 \tag{9.7}$$

$$\mathbf{Q}'(0) = \mathbf{c} = \mathbf{P}'_0 \tag{9.8}$$

$$\mathbf{Q}'(1) = 3\mathbf{a} + 2\mathbf{b} + \mathbf{c} = \mathbf{P}'_1 \tag{9.9}$$

We can see that equations 9.6 and 9.8 already determine that **c** and D are $\mathbf{P}'_0$ and P_0, respectively. Substituting these into equations 9.7 and 9.9 and solving for **a** and **b** gives

$$\mathbf{a} = 2(P_0 - P_1) + \mathbf{P}'_0 + \mathbf{P}_1$$
$$\mathbf{b} = 3(P_1 - P_0) - 2\mathbf{P}'_0 - \mathbf{P}'_1$$

Substituting our now known values for **a**, **b**, **c**, and D into equation 9.4 gives:

$$Q(u) = \left[2(P_0 - P_1) + \mathbf{P}'_0 + \mathbf{P}'_1\right]u^3 + \left[3(P_1 - P_0) - 2\mathbf{P}'_0 - \mathbf{P}'_1\right]u^2 + \mathbf{P}'_0 u + P_0$$

This can be rearranged in terms of the boundary conditions to produce our final equation:

$$Q(u) = (2u^3 - 3u^2 + 1)P_0 + (-2u^3 + 3u^2)P_1 + (u^3 - 2u^2 + u)\mathbf{P}'_0 + (u^3 - u^2)\mathbf{P}'_1$$

This is known as a *Hermite curve*. We can also represent this as the product of a matrix multiplication, just as we did with linear interpolation. In this case, the matrices are

$$\mathbf{U} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} P_0 \\ P_1 \\ \mathbf{P}'_0 \\ \mathbf{P}'_1 \end{bmatrix}$$

We can use either formulation to build piecewise curves just as we did for linear interpolation. As before, we can think of each segment as a separate function, valid over the interval [0, 1]. Then to create a C^1 continuous curve, two adjoining segments Q_i and Q_{i+1} would have to have matching positions such that

$$Q_i(1) = Q_{i+1}(0)$$

and matching tangent vectors such that

$$\mathbf{Q}'_i(1) = \mathbf{Q}'_{i+1}(0)$$

What we end up with is a set of sample positions $\{P_0, \ldots, P_n\}$, tangent vectors $\{\mathbf{P}'_0, \ldots, \mathbf{P}'_n\}$, and times $\{t_0, \ldots, t_n\}$. At a given point adjoining two curve segments Q_i and Q_{i+1}

$$Q_i(1) = Q_{i+1}(0) = P_{i+1}$$
$$\mathbf{Q}'_i(1) = \mathbf{Q}'_{i+1}(0) = \mathbf{P}'_{i+1}$$

Figure 9.3 shows this situation in the piecewise Hermite curve.

The tangent vectors are used for more than just maintaining first-derivative continuity across each sample point. Changing their magnitude also controls the speed at which we move through the point and consequently through the curve. They also affect the shape of the curve. Take a look at Figures 9.4a and 9.4b. The longer the vector, the faster we will move and the sharper the curvature. We can create a completely different curve through our sample points, simply by adjusting the tangent vectors.

There is, of course, no reason that the tangents $\mathbf{Q}'_i(1)$ and $\mathbf{Q}'_{i+1}(0)$ have to match. One possibility is to match the tangent directions but not the tangent magnitudes — this gives us G^1 continuity. The resulting function has a discontinuity in its derivative but usually still appears smooth. It also has the advantage that it allows us to control how our curve looks across each segment a little better. For example, it might be that we want to have the appearance of a continuous curve but also be able to have more freedom in how each individual segment is shaped. By maintaining the same direction but allowing for different magnitudes, this function provides for the kind of flexibility we need in this instance (Figure 9.5).

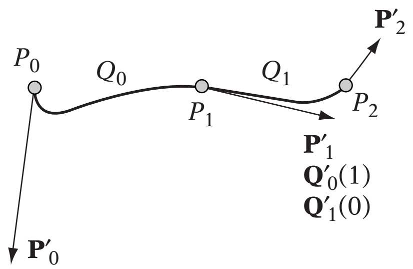


FIGURE 9.3 Piecewise Hermite curve. Tangents at *P*1 match direction and magnitude.

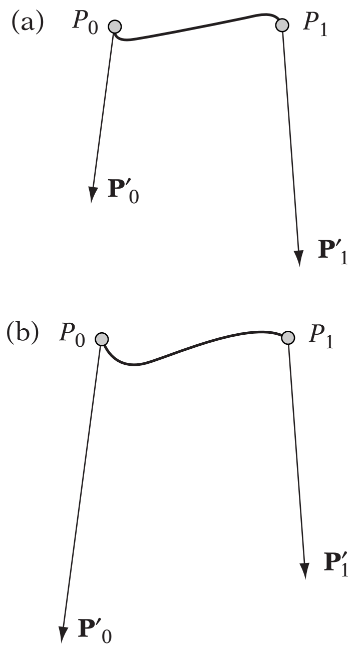


FIGURE 9.4 Hermite curve with (a) small tangent and low curvature (b) large tangent and higher curvature.

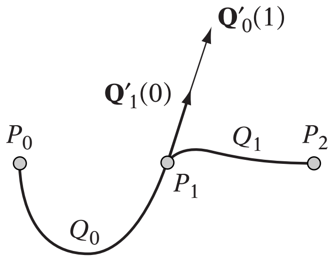


FIGURE 9.5 Piecewise Hermite curve. Tangents at P_1 have same direction but differing magnitudes.

Another possibility is that the tangent directions don't match at all. In this case we'll end up with a kink, or cusp, in the whole curve (Figure 9.6). While not physically realistic, it does allow for sudden changes in direction. The combination of all the possibilities at each sample point — equal tangents, equal tangent directions with non-equal magnitudes, and non-equal tangent directions — gives us a great deal of flexibility in creating our interpolating

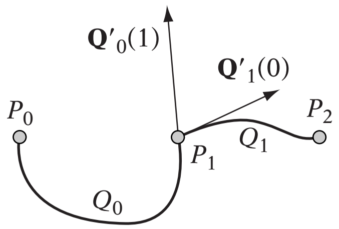


FIGURE 9.6 Piecewise Hermite curve. Tangents at P_1 have differing directions and magnitudes.

function across all the sample points. To allow for this level of control, we need to set two tangents at each internal sample point, which we'll express as $\mathbf{P}'_{i,1}$ and $\mathbf{P}'_{i+1,0}$. Alternatively, we can think of a curve segment as being defined by two points P_i and P_{i+1}, and two tangents $\mathbf{P}'_{i,0}$ (the "incoming" tangent) and $\mathbf{P}'_{i+1,1}$ (the "outgoing" tangent).

One question remains: how do we generate these tangents? One simple answer is that most existing tools that artists will use, such as Alias's Maya and Discreet's 3D Studio Max, provide ways to set up Hermite curves and their corresponding tangents. When exporting the sample points for subsequent animation, we export the tangents as well. Some tweaking may need to be done to guarantee that the curves generated in internal code match that in the artist program; information on a particular representation is usually available from the manufacturer.

Another common way of generating Hermite data is using in-house tools built for a specific purpose — for example, a tool for managing paths for cameras and other animated objects. In this case, an interface will have to be created to manage construction of the path. One possibility is to click to set the next sample position, and then drag the mouse away from the sample position to set tangent magnitude and direction. A line segment with an arrowhead can be drawn showing the outgoing tangent, and a corresponding line segment with a tail drawn showing the incoming tangent (Figure 9.7).

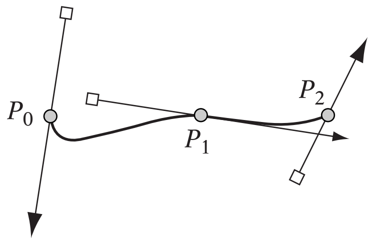


FIGURE 9.7 Possible interface for Hermite curves, showing in–out tangent vectors.

We will need to modify the tangents so that they can either have different magnitudes or different directions. Many drawing programs control this by allowing three different tangent types. For example, Jasc's Paint Shop Pro refers to them as symmetric, asymmetric, and cusp. With the symmetric node, clicking and dragging on one of the segment ends rotates both segments and changes their lengths equally, to maintain equal tangents. With an asymmetric node, clicking and dragging will rotate both segments to maintain equal direction but change only the length of the particular tangent clicked on. And with a cusp, clicking and dragging a segment end changes only the length and direction of that tangent. This allows for the full range of possibilities in continuity previously described.

9.5.2 Automatic Generation of Hermite Curves

But suppose we don't need the full control of generating tangents for each sample position. Instead, we just want to automatically generate a smooth curve that passes through all the sample points. To do this we'll need to have a method of creating reasonable tangents for each sample. One solution is to use Lagrange interpolation to generate a quadratic function using a given sample point and its two neighbors, and then take the derivative of the function to get a tangent value at the sample point. A similar possibility is to take, for a given point P_i, the weighted average of $(P_{i+1} - P_i)$ and $(P_i - P_{i-1})$. However, for both of these it will still be necessary to set a tangent for the two endpoints, since they have only one neighboring point.

Source Code
Demo
Auto Hermite

Another method creates tangents that maintain C^2 continuity at the interior sample points. To do this, we'll need to solve a system of linear equations, using our sample points as the known quantities and the tangents as our unknowns. We'll begin by computing the first derivative of the Hermite curve Q:

$$\mathbf{Q}_i'(u) = (6u^2 - 6u)P_i + (-6u^2 + 6u)P_{i+1} + (3u^2 - 4u + 1)\mathbf{P}_i' + (3u^2 - 2u)\mathbf{P}_{i+1}'$$

and from that the second derivative $\mathbf{Q}''$:

$$\mathbf{Q}_i''(u) = (12u - 6)P_i + (-12u + 6)P_{i+1} + (6u - 4)\mathbf{P}_i' + (6u - 2)\mathbf{P}_{i+1}'$$

At a given interior point P_{i+1}, we want the outgoing second derivative of curve Q_i to equal the incoming second derivative of curve Q_{i+1}. We'll assume that each curve segment has a valid parameterization from 0 to 1, so we want

$$\begin{gathered}\mathbf{Q}_i''(1) = \mathbf{Q}_{i+1}''(0)\\ 6P_i - 6P_{i+1} + 2\mathbf{P}_i' + 4\mathbf{P}_{i+1}' = -6P_{i+1} + 6P_{i+2} - 4\mathbf{P}_{i+1}' - 2\mathbf{P}_{i+2}'\end{gathered}$$

This can be rewritten to place our knowns on one side of the equation and unknowns on the other:

$$2\mathbf{P}'_i + 8\mathbf{P}'_{i+1} + 2\mathbf{P}'_{i+2} = 6[(P_{i+2} - P_{i+1}) + (P_{i+1} - P_i)]$$

This simplifies to

$$\mathbf{P}'_i + 4\mathbf{P}'_{i+1} + \mathbf{P}'_{i+2} = 3(P_{i+2} - P_i)$$

Applying this to all of our sample points $\{P_0, \ldots, P_n\}$ creates $n-1$ linear equations. This can be written as a matrix product as follows:

$$\begin{bmatrix} 1 & 4 & 1 & & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \end{bmatrix}$$

This means we have $n-1$ equations with $n+1$ unknowns. To solve this, we will need two more equations. We have already constrained our interior tangents by ensuring C^2 continuity; what remains is to set our two tangents at each extreme point. One possibility is to set them to given values $\mathbf{v}_0$ and $\mathbf{v}_1$, or

$$\mathbf{Q}'_0(0) = \mathbf{P}'_0 = \mathbf{v}_0 \tag{9.10}$$

$$\mathbf{Q}'_{n-1}(1) = \mathbf{P}'_n = \mathbf{v}_1 \tag{9.11}$$

This is known as a clamped end condition, and the resulting curve is a *clamped cubic spline*. Our final system of equations is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \\ \mathbf{v}_1 \end{bmatrix}$$

Solving this system of equations gives us the appropriate tangent vectors. This is not as bad as it might seem. Because this matrix (known as a *tridiagonal matrix*) is sparse and extremely structured, the system is very easy and efficient to solve.

For this discussion, we have assumed uniform time values (this is also known as a *normalized cubic spline*). However, as mentioned under linear interpolation, our time values may vary from t_i to t_{i+1} across each spline segment. One solution is to do the same thing we did for linear interpolation: if we know that a given value t lies between t_i and t_{i+1}, we can use equation 9.1 to normalize our time value to the range $0 \leq u \leq 1$, and use that as our parameter to curve segment Q_i. While not strictly correct, this provides a reasonable approximation. For those who require it, a full derivation for non-normalized splines can be found in [95].

9.5.3 Natural, Cyclic, and Acyclic End Conditions

Source Code
Demo
Auto Hermite

In the preceding examples, we generated splines assuming that the beginning and end tangents were clamped to values set by the programmer or the user. This may not be convenient; we may want to avoid specifying tangents at all. An alternative approach is to set conditions on the end tangents, just as we did with the internal tangents, to reduce the amount of input needed.

The first such possibility is to assume that the second derivative is 0 at the two extremes; that is, $\mathbf{Q}_0''(0) = \mathbf{Q}_{n-1}''(1) = 0$. This is known as a *relaxed* or *natural* end condition, and the spline created is known as a *natural spline*. As the name indicates, this produces a very smooth and natural looking curve at the endpoints, and in most cases this is the end condition we would want to use.

With a natural spline, we don't need to specify tangent information at all — we can compute the two unconstrained tangents from the clamped spline using the second derivative condition.

At point P_0, we know that

$$\begin{aligned} 0 &= \mathbf{Q}_0''(0) \\ &= -6P_0 + 6P_1 - 4\mathbf{P}_0' - 2\mathbf{P}_1' \end{aligned}$$

As before, we can rewrite this so that the unknowns are on the left side and the knowns on the right:

$$4\mathbf{P}_0' + 2\mathbf{P}_1' = 6P_1 - 6P_0$$

or

$$2\mathbf{P}_0' + \mathbf{P}_1' = 3(P_1 - P_0) \qquad (9.12)$$

Similarly, at point P_n, we know that

$$\begin{aligned} 0 &= \mathbf{Q}''_{n-1}(1) \\ &= 6P_{n-1} - 6P_n + 2\mathbf{P}'_{n-1} + 4\mathbf{P}'_n \end{aligned}$$

This can be rewritten as

$$\mathbf{P}'_{n-1} + 2\mathbf{P}'_n = 3(P_n - P_{n-1}) \tag{9.13}$$

We can substitute equations 9.12 and 9.13 for our first and last equations in the clamped case, to get the matrix product:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} 3(P_1 - P_0) \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \\ 3(P_n - P_{n-1}) \end{bmatrix}$$

Once again, by solving this system of linear equations or inverting the main matrix, we can find the values for our tangents.

Another possibility, known as the *cyclic* end condition, is to assume that the first and second derivatives at the endpoints are equal. Note that this doesn't necessarily mean that the positions of the two endpoints have to be equal. Neither does it mean that the resulting curve will be symmetric if they are equal (i.e., you can't guarantee an oval). You might use a curve of this type if you want to ensure that the animated object ends up moving in the same direction at the end of the curve as it does at the beginning.

We can represent the cyclic end condition as

$$\begin{aligned} \mathbf{Q}'_0(0) &= \mathbf{Q}'_{n-1}(1) \\ \mathbf{Q}''_0(0) &= \mathbf{Q}''_{n-1}(1) \end{aligned}$$

Expanding the first equation gives

$$\mathbf{P}'_0 = \mathbf{P}'_n$$

which is not all that surprising: the initial tangent is equal to the final tangent. Expanding the second gives

$$-6P_0 + 6P_1 - 4\mathbf{P}'_0 - 2\mathbf{P}'_1 = 6P_{n-1} - 6P_n + 2\mathbf{P}'_{n-1} + 4\mathbf{P}'_n$$

or

$$2\mathbf{P}'_0 + \mathbf{P}'_1 + \mathbf{P}'_{n-1} + 2\mathbf{P}'_n = 3(P_1 - P_0) + 3(P_n - P_{n-1})$$

We can substitute $\mathbf{P}'_0$ for $\mathbf{P}'_n$, since they are equal, to get the final constraint equation:

$$4\mathbf{P}'_0 + \mathbf{P}'_1 + \mathbf{P}'_{n-1} = 3(P_1 - P_0) + 3(P_n - P_{n-1})$$

As before, we can set this up as a series of linear equations. However, since $\mathbf{P}'_0 = \mathbf{P}'_n$, we have only $n-1$ unknowns, and so we need only $n-1$ equations. Our matrix ends up being

$$\begin{bmatrix} 4 & 1 & 0 & 0 & \cdots & 1 \\ 1 & 4 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 4 & 1 \\ 1 & 0 & \cdots & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \end{bmatrix} = \begin{bmatrix} 3(P_1 - P_0) + 3(P_n - P_{n-1}) \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \end{bmatrix}$$

The *acyclic* end condition is similar to the cyclic end condition, except that the first and second derivatives are negatives of each other. If the positions of the two endpoints are equal, this can produce a shape like the head of a tennis racket. You might use a curve of this type if you want to ensure that the animated object ends up moving in the opposite direction at the end of the curve as it does at the beginning.

We can represent the acyclic end condition as

$$\mathbf{Q}'_0(0) = -\mathbf{Q}'_n(1)$$
$$\mathbf{Q}''_0(0) = -\mathbf{Q}''_n(1)$$

Using a similar process to the cyclic end condition, we end up with the matrix equation for the acyclic end condition:

$$\begin{bmatrix} 4 & 1 & 0 & 0 & \cdots & -1 \\ 1 & 4 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1 & 4 & 1 \\ -1 & 0 & \cdots & 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \end{bmatrix} = \begin{bmatrix} 3(P_1 - P_0) - 3(P_n - P_{n-1}) \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \end{bmatrix}$$

9.6 Catmull-Rom Splines

Source Code
Demo
Catmull

An alternative for automatic generation of a parametric curve is the *Catmull-Rom spline*. This takes a similar approach to some of the initial methods we described for Hermite curves (tangent of parabola, weighted average), where tangents are generated based on the positions of the sample points. The standard Catmull-Rom splines create the tangent for a given sample point by taking the neighboring sample points, subtracting to create a vector, and halving the length. So, for sample P_i, the tangent $\mathbf{P}'_i$ is

$$\mathbf{P}'_i = \frac{1}{2}(P_{i+1} - P_{i-1})$$

If we substitute this into our matrix definition of a Hermite curve between P_i and P_{i+1}, this gives us

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_i \\ P_{i+1} \\ \frac{1}{2}(P_{i+1} - P_{i-1}) \\ \frac{1}{2}(P_{i+2} - P_i) \end{bmatrix}$$

We can rewrite this in terms of P_{i-1}, P_i, P_{i+1}, P_{i+2} to get

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

This provides a definition for curve segments Q_1 to Q_{n-2}, so it can be used to generate a C^1 curve from P_1 to P_{n-1}. However, since there is no P_{-1} or P_{n+1}, we once again have the problem that curves Q_0 and Q_{n-1} are not valid due to undefined tangents at the end points. And as before, these can either be provided by the artist or programmer, or automatically generated. Parent [87] presents one technique. For P_0, we can take the next two points, P_1 and P_2, and use them to generate a new phantom point, $P_1 + (P_1 - P_2)$. If we subtract P_0 from the phantom point and halve the length, this gives a reasonable tangent for the start of the curve (Figure 9.8). The tangent at P_n can be generated similarly.

Since our knowns for the outer curve segments are two points and a tangent, another possibility is to use a quadratic equation to generate these segments. We can derive this in a similar manner as the Hermite spline equation.

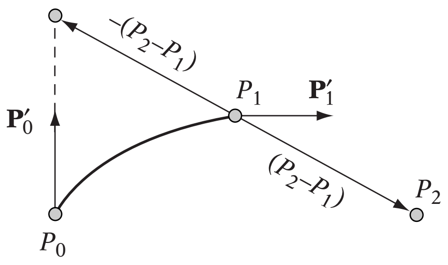


FIGURE 9.8 Automatic generation of tangent vector at $P0$, based on positions of $P1$ and $P2$.

The general quadratic equation will have the form:

$$Q(u) = \mathbf{a}u^2 + \mathbf{b}u + C \tag{9.14}$$

For the case of Q_0, we know that

$$\begin{aligned} Q_0(0) &= C = P_0 \\ Q_0(1) &= \mathbf{a} + \mathbf{b} + C = P_1 \\ \mathbf{Q}'_0(1) &= 2\mathbf{a} + \mathbf{b} = \mathbf{P}'_1 \\ &= \frac{1}{2}(P_2 - P_0) \end{aligned}$$

Solving for **a**, **b**, and C and substituting into equation 9.14, we get

$$Q_0(u) = \left(\frac{1}{2}P_0 - P_1 + \frac{1}{2}P_2\right)u^2 + \left(-\frac{3}{2}P_0 + 2P_1 - \frac{1}{2}P_2\right)u + P_0$$

Rewriting in terms of P_0, P_1, and P_2 gives

$$Q_0(u) = \left(\frac{1}{2}u^2 - \frac{3}{2}u + 1\right)P_0 + \left(-u^2 + 2u\right)P_1 + \left(\frac{1}{2}u^2 - \frac{1}{2}u\right)P_2$$

As before, we can write this in matrix form:

$$Q_0(u) = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -3 & 4 & -1 \\ 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

A similar process can be used to derive Q_{n-1}:

$$Q_{n-1}(u) = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix} \begin{bmatrix} P_{n-2} \\ P_{n-1} \\ P_n \end{bmatrix}$$

9.7 Bézier Curves

9.7.1 Definition

Source Code
Demo
Bézier

The previous techniques for generating curves from a set of points meet the functional requirements of controlling curvature and maintaining continuity. However, other than Hermite curves where the tangents are user-specified, they are not so good at providing a means of controlling the shape that is produced. It is not always clear how adjusting the position of a point will change the curve produced, and if we're using a particular type of curve and want to pass through a set of fixed points, there is usually only one possibility.

Bézier curves were created to meet this need. They were devised by Pierre Bézier for modeling car bodies for Renault and further refined by Forrest, Gordon, and Riesenfeld. A cubic Bézier curve uses four *control points*: two endpoints P_0 and P_3 that the curve interpolates, and two points P_1 and P_2 that the curve approximates. Their positions act, as their name suggests, to control the curve. The convex hull, or control polygon, formed by the control points bounds the curve (Figures 9.9a and 9.9b). Another way to think of it is that the curve mimics the shape of the control polygon. Note that the four points in this case do not have to be coplanar, which means that the curve generated will not necessarily lie on a plane either.

The tangent vector at point P_0 points in the same direction as the vector $P_1 - P_0$. Similarly, the tangent at P_3 has the same direction as $P_3 - P_2$. As we will see, there is a definite relationship between these vectors and the tangent vectors used in Hermite curves. For now we can think of the polygon edge between the interpolated end point and neighboring control point as giving us an intuitive sense of what the tangent is like at that point.

So far we've only shown cubic Bézier curves, but there is no reason why we couldn't use only three control points to produce a quadratic Bézier curve (Figure 9.10) or more control points to produce higher-order curves. A general Bézier curve is defined by the function

$$Q(u) = \sum_{i=0}^{n} P_i J_{n,i}(u)$$

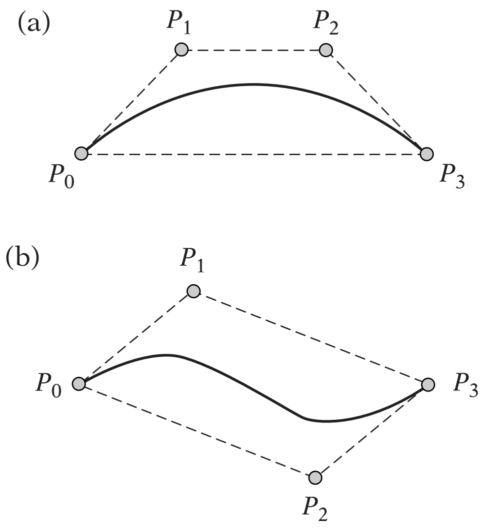


Figure 9.9 Example of cubic Bézier curve showing convex hull.

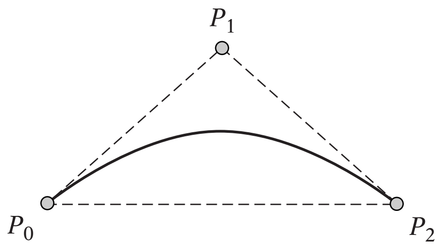


Figure 9.10 Example of quadratic Bézier curve showing convex hull.

where the set of P_i are the control points, and

$$J_{n,i}(u) = \begin{pmatrix} n \\ i \end{pmatrix} u^i \, (1-u)^{n-i}$$

where

$$\begin{pmatrix} n \\ i \end{pmatrix} = \frac{n!}{i!(n-i)!}$$

The polynomials generated by $J_{n,i}$ are also known as the Bernstein polynomials, or *Bernstein basis*.

In most cases, however, we will use only cubic Bézier curves. Higher order curves are more expensive and can lead to odd oscillations in the shape of the curve. Quadratic curves are useful when processing power is limited (the game Quake 3 used them, for example) but don't have quite the flexibility of cubic curves. For example, they don't allow for the familiar S shape in Figure 9.9b. To generate something similar with quadratic curves requires two piecewise curves, and hence more data.

The standard representation of an order n Bézier curve is to use an ordered list of points $P_0, \ldots, P_n$ as the control points. Using this representation, we can expand the general definition to get the formula for the cubic Bézier curve:

$$Q(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u)P_2 + u^3 P_3 \tag{9.15}$$

The matrix form looks like

$$Q(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

We can think of the curve as a set of affine combinations of the four points, where the weights are defined by the four basis functions $J_{3,i}$. We can see these basis functions graphed in Figure 9.11. At a given parameter value u, we grab the four basis values and use them to compute the affine combination.

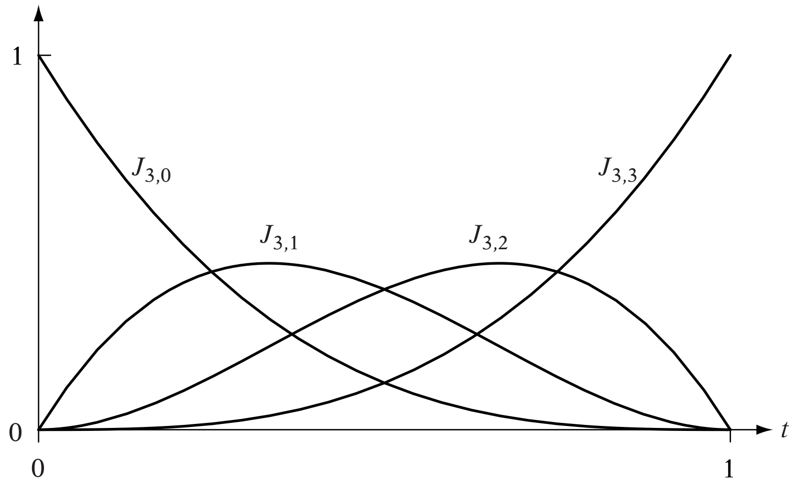


FIGURE 9.11 Cubic Bézier curve basis functions.

As hinted at, there is a relationship between cubic Bézier curves and Hermite curves. If we set our Hermite tangents to $3(P_1 - P_0)$ and $3(P_3 - P_2)$, substitute those values into our cubic Hermite equation, and simplify, we end up with the cubic Bézier equation.

9.7.2 Piecewise Bézier Curves

As with linear interpolation and Hermite curves, we can interpolate a curve through more than two points by creating curve segments between each neighboring pair of interpolation points. Many of the same principles apply with Bézier curves as did with Hermite curves. In order to maintain matching direction for our tangents, giving us G^1 continuity, each interpolating point and its neighboring control points need to be collinear. To obtain equal tangents, and therefore C^1 continuity, the control points need to be collinear with and equidistant to the shared interpolating point. Drawing a line segment through the three points gives a three-lobed barbell shape, seen in Figure 9.12.

The barbell makes another very good interface for managing our curves. If we set up our interpolating point as a pivot, then we can grab one neighboring control point and rotate it around to change the direction of the tangent. The other neighboring control point will rotate correspondingly to maintain collinearity and equal distance, and thereby C^1 continuity. If we drag the control point away from our interpolating point, that will increase the length of our tangent. We can leave the other control point at the original distance, if we like, to create different arrival/departure speeds while still maintaining G^1 continuity. Or, we can match its distance from the sample as well, to maintain C^1 continuity. And of course, we can move each neighboring control point independently to create a cusp at that interpolating point.

This seems very similar to our Hermite interface, so the question may be, why use Bézier curves? The main advantage of the Bézier interface over the Hermite interface is that, as mentioned, the control points act to bound the curve, and so give a much better idea of how the shape of the curve will

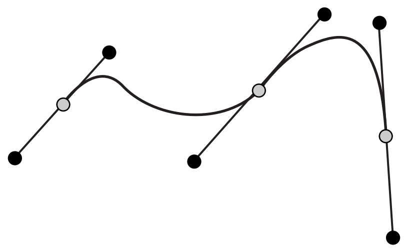


Figure 9.12 Example interface for Bézier curves.

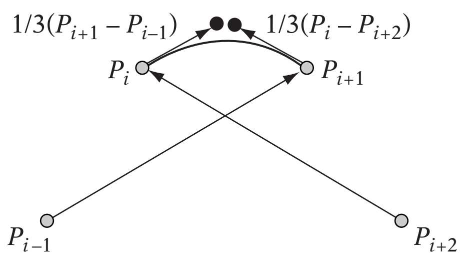


Figure 9.13 Automatic construction of approximating control points with Bézier curve.

change as we move the control points around. Because of this, many drawing packages use Bézier curves instead of Hermite curves.

While in most cases we will want to make use of user-created data with Bézier curves, it is sometimes convenient to automatically generate them, just as we did with Hermite curves. Parent [87] provides a method for automatically generating Bézier control points from a set of sample positions, as shown in Figure 9.13. Given four points P_{i-1}, P_i, P_{i+1}, and P_{i+2}, we want to compute the two control points between P_i and P_{i+1}. We compute the tangent vector at P_i by computing the difference between P_{i+1} and P_{i-1}. From that we can compute the first control point as $P_i + \frac{1}{3}(P_{i+1} - P_{i-1})$. The same can be done to create the second control point as $P_{i+1} - \frac{1}{3}(P_{i+2} - P_i)$. This is very similar to how we created the Catmull-Rom spline, but with tangents twice as large in magnitude.

9.8 B-Splines

Source Code
Demo
B-Spline

The first set of curves we looked at were interpolating curves, which pass through all the given points. With Bézier curves, the resulting curve interpolates two of the control points, while approximating the others. B-splines are a generalization of this — depending on the form of the B-spline, all or none of the points can be interpolated. Because of this, in a B-spline all of the control points can be used as approximating points (Figure 9.14). In fact, B-splines are so flexible they can be used to represent all of the curves we have described so far. However, with flexibility comes a great deal of complexity. Because of this, B-splines are not yet in common usage in games, either for animation or surface construction. Hence, this section is designed only to give an overview, with implementation details of a single commonly used B-spline.

The motivation for using B-splines is twofold. First of all, most of our previous solutions have only C^1 continuity. In some cases we may want to ensure

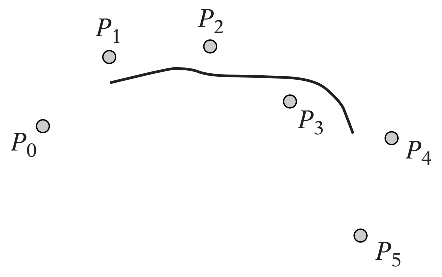


FIGURE 9.14 B-spline approximating curve.

that we have at least C^2 continuity (although, admittedly, such cases are rare in animation). While natural cubic splines provide that level of continuity, they also are subject to *global control*. That is, changing a single point affects the entire curve, which requires us to recalculate the entire thing.

B-splines, in comparison, provide what's called *local control*. Each point has influence only over a limited region of the curve. This is controlled by a set of basis functions (hence the *B* in B-spline) that are computed for each sample location and added up to give our final curve position. Piecewise Hermite and Bézier curves do allow us local control, but at the cost of having to adjust other control points or tangents to maintain continuity. B-splines can maintain continuity without such adjustments.

B-splines are computed similarly to Bézier curves. We set up a basis function for each control point in our curve, and then for each parameter value u we multiply the appropriate basis function by its point and add the results. In general, this can be represented by

$$Q(u) = \sum_{i=0}^{n} P_i B_i(u)$$

where each P_i is a point and B_i is a basis function for that point. The basis functions in this case are far more general than those described for Bézier curves, which gives B-splines their flexibility and their power.

Like our previous piecewise curves, B-splines are broken into smaller segments. The difference is that the number of segments is not necessarily dependent on the number of points, and the intermediary point between each segment is not necessarily one of our control points. These intermediary points are called *knots*. If the knots are spaced equally in time, the curve is known as a *uniform B-spline*. Otherwise it is a *nonuniform B-spline*.

The standard example of a uniform cubic B-spline has knots lying 1 unit apart in u — the knots are at $Q(0)$, $Q(1)$, $Q(2)$, and so on. So a given segment Q_i describes the curve between $Q(i)$ and $Q(i+1)$. Assuming that we're using the same convention we have before, where each segment is parameterized from 0 to 1, the partial basis functions

$$\begin{aligned}
B_{i-3}(u) &= \frac{1}{6}(-u^3 + 3u^2 - 3u + 1) \\
B_{i-2}(u) &= \frac{1}{6}(3u^3 - 6u^2 + 4) \\
B_{i-1}(u) &= \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\
B_i(u) &= \frac{1}{6}(u^3)
\end{aligned}$$

give us C^2 continuity at each knot. Figure 9.15 shows these bases graphed within the interval of one segment.

The matrix representation for a particular segment Q_i is

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix}$$

We stated that the B_is we set above were *partial* basis functions. For a given point P_i, its corresponding full basis function B_i forms a bell-shaped curve (Figure 9.16). Each corresponding B_{i-1}, B_{i+1} is a translation of this curve, with the peak centered over each knot parameter k. If we look at the complete

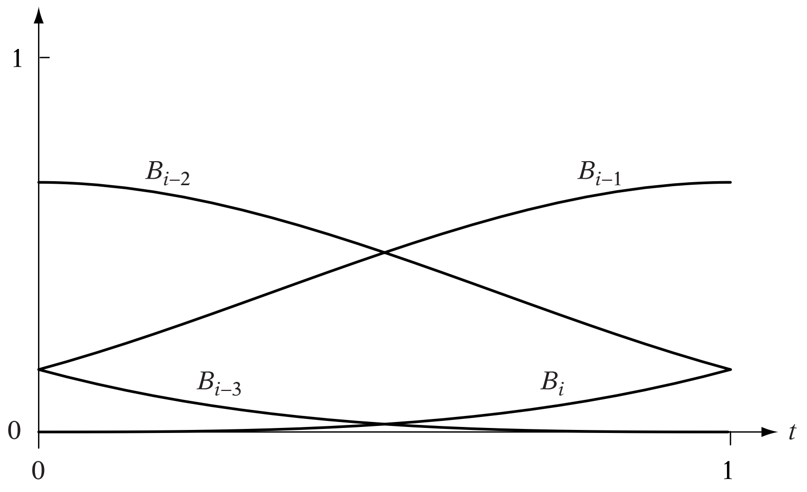


FIGURE 9.15 Uniform cubic B-spline basis functions.

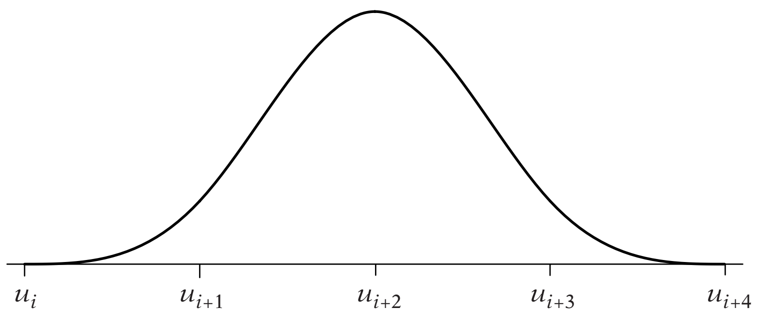


FIGURE 9.16 Single uniform B-spline basis function.

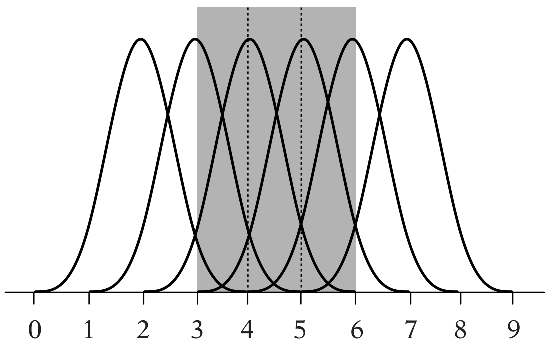


FIGURE 9.17 Overlapping basis functions for three B-spline segments.

basis functions for a series of segments joined together (Figure 9.17), we see that each basis affects up to four segments. Each segment is only controlled by four points, and each point controls no more than four segments. This demonstrates the local control properties of the B-spline.

Figure 9.14 shows an example of a uniform cubic B-spline generated using the preceding basis. Clearly, this is a pure approximating curve where the curve doesn't pass through any of the control points. As we've seen before with Catmull-Rom splines, the end segments are undefined since we don't have enough points to describe them. Usually, this isn't an issue since it is an approximating curve, but Bartels *et al.* [8] describe methods for creating end conditions for B-splines, much as we did with Hermite curves.

Finally, suppose we don't want to approximate all the points, but want to ensure that the curve passes through specific positions. With uniform B-splines, we copy the points we want to interpolate. Duplication will draw the curve closer to the point and triplicating it will cause the curve to pass through it. The one drawback is that we will end up with a kink in the curve at that point. Another possibility is to use nonuniform B-splines.

By triplicating knot values, we can cause the curve to pass through a point, again with a loss of continuity. The curve produced will not quite be the same curve as with the control point triplication method.

This is merely a taste of what is possible. As mentioned, B-splines are not often used for animation; they are more commonly used when building surface representations. A full description of the power and complexity of B-splines is out of the purview of this text, so for those who are interested, more information on B-splines and other curves can be found in [36], [94], or [8].

9.9 Rational Curves

The curves we have discussed so far have the property that any affine transformation on the set of points (or tangents, in the case of Hermite curves) generating the curve will transform the curve accordingly. So for example, if we want to transform a Bézier curve from the local frame to the view frame, all we need to do is transform the control points and then generate the curve in the view frame.

However, this will not work for a perspective transformation, due to the need for a reciprocal division at each point on the curve. The answer is to apply a process similar to the one we used when transforming points, by adding an additional parameterized function $w(u)$ that we divide by when generating the points along the curve.

We create a rational curve by first considering a curve $Q(u)$ in $\mathbb{R}P^3$, similar to our space curve in $\mathbb{R}^3$ but with the $w(u)$ function added:

$$Q(u) = (x(u), y(u), z(u), w(u))$$

The corresponding rational curve $R(u)$ projects this homogeneous curve $Q(u)$ into $\mathbb{R}^3$ as

$$R(u) = \left(\frac{x(u)}{w(u)}, \frac{y(u)}{w(u)}, \frac{z(u)}{w(u)}\right)$$

So for example, we can define $Q(u)$ as a Bézier curve:

$$Q(u) = \sum_{i=0}^{3} P_i J_{3,i}(u)$$

but now each P_i is a point in $\mathbb{R}P^3$, or

$$P_i = (w_i x_i, w_i y_i, w_i z_i, w_i)$$

Note that $w(u)$ in this case is just another Bézier function:

$$w(u) = \sum_{i=0}^{3} w_i J_{3,i}(u)$$

The corresponding rational curve is

$$R(u) = \left(\frac{\sum w_i x_i J_{3,i}(u)}{\sum w_i J_{3,i}(u)}, \frac{\sum w_i y_i J_{3,i}(u)}{\sum w_i J_{3,i}(u)}, \frac{\sum w_i z_i J_{3,i}(u)}{\sum w_i J_{3,i}(u)} \right)$$

or

$$R(u) = \frac{\sum w_i P_i J_{3,i}(u)}{\sum w_i J_{3,i}(u)}$$

where each P_i is one of our standard control points in $\mathbb{R}^3$.

We can create a rational curve from a nonrational one by implicitly setting the w_is to 1, so rational curves encapsulate nonrational curves. In the previous case, $R(u)$ collapses to the standard cubic Bézier definition.

To apply a perspective projection to a curve, we transform the control points as we normally would with our projection matrix, but defer the division by w until we actually generate the points along the curve. This is much more efficient than the alternative, where we generate the curve points in world space and apply the full perspective transformation to every single point generated.

There are a number of uses for rational curves. The first has already been stated: we can use it as a more efficient method for projecting curves. But it also allows us to set weights w_i for the control points so that we can direct the curve to pass closer to one point or another. Figure 9.18 shows the effect

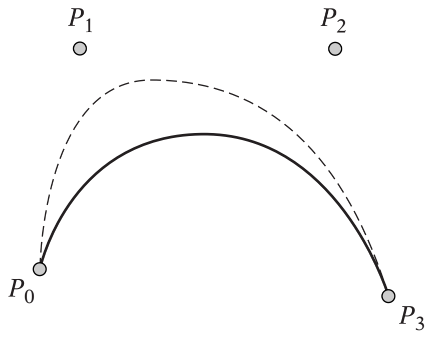


FIGURE 9.18 Dotted line shows effect of giving vertex P1 greater weight in a rational Bézier curve.

of such weighting being applied to control point P_1 on a Bézier curve. The higher the relative weight, the more the curve tends towards that point.

Another use of rational curves is to create conic section curves, such as circles and ellipses. Nonrational curves, since they are polynomials, can only approximate conic sections. As an example, we can construct a quarter-circle in $\mathbb{R}^2$ with a rational quadratic Bézier curve in $\mathbb{R}P^2$ (i.e., coordinates are (x, y, w)), where the Bézier function is

$$Q(u) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

and the control points are

$$\begin{aligned} P_0 &= (0, 1, 1) \\ P_1 &= \left(\sqrt{2}/2, \sqrt{2}/2, \sqrt{2}/2\right) \\ P_2 &= (1, 0, 1) \end{aligned}$$

The entire circle can be exactly duplicated by a piecewise curve made up of four such curve segments, with control points in the appropriate quadrants.

In the examples thus far, we have used rational Bézier curves, but the most commonly used of the rational curves are nonuniform rational B-splines, or NURBS. Since they can produce conic as well as general curves and surfaces, they are extremely useful in CAD systems and modeling for computer animation. Like B-splines, rational curves and particularly NURBS are not yet used much in games because of their relative performance cost and because of concern by artists about lack of control.

9.10 Rendering Curves

9.10.1 Forward Differencing

Source Code
Library
IvCurves
Filename
IvHermite

Given a parametric curve, it is only natural that we might want to render it at some point. The main purpose would be to allow artists to see, and thus more accurately control, the animation paths that they are creating. We may also want to render a curve for debugging, to allow engineers testing the animation code to ensure that the path taken is the one intended. We may want rendered curves for other reasons as well: game interface components, for example.

In most cases we will be using a cubic curve. The simplest rendering method is to take the general function for our curve or curve segment $Q(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + D$, evaluate it at $n+1$ values of u, and then use those $n+1$ points to create n line segments, which we render with our standard line drawing algorithm. Assuming that we're generating points in $\mathbb{R}^3$, this will take

11 multiplies and 9 adds per point (we save three multiplies by computing u^3 as $u \cdot u^2$).

An alternative which is slightly faster is to use Horner's rule, which expresses the same cubic curve as

$$Q(u) = ((\mathbf{a}u + \mathbf{b})u + \mathbf{c})u + D$$

This will take only 9 multiplies and 9 adds per point. In addition, it can actually improve our floating point accuracy under certain circumstances.

This assumes that there is no pattern to how we evaluate our curve. Suppose we can instead sample our curve at even intervals of u, say at a time step of every h. This gives us a list of $n + 1$ parameter values: $0, h, 2h, \ldots, nh$. In such a situation, we can use a technique called *forward differencing*.

For the time being, let's consider computing only the x values for our points. For a given value x_i, located at parameter u, we can compute the next value x_{i+1} at parameter $u + h$. Subtracting x_i from x_{i+1}:

$$x_{i+1} - x_i = x(u + h) - x(u)$$

We'll label this difference between x_{i+1} and x_i as $\Delta x_1(u)$. For a cubic curve this equals

$$\begin{aligned}
\Delta x_1(u) &= a(u+h)^3 + b(u+h)^2 + c(u+h) + d - (au^3 + bu^2 + cu + d) \\
&= a(u^3 + 3hu^2 + 3h^2u + h^3) + b(u^2 + 2hu + h^2) + c(u+h) + d \\
&\quad - au^3 - bu^2 - cu - d \\
&= au^3 + 3ahu^2 + 3ah^2u + ah^3 + bu^2 + 2bhu + bh^2 + cu + ch + d \\
&\quad - au^3 - bu^2 - cu - d \\
&= 3ahu^2 + 3ah^2u + ah^3 + 2bhu + bh^2 + ch \\
&= (3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)
\end{aligned}$$

Pseudocode to compute the set of values might look like

```
u = 0;
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
for ( i = 1; i <= n; i++ )
{
    u += h;
    x += dx1;
```

```
        output(x);
        dx1 = (3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch);
    }
```

While we have removed the cubic equation, we have introduced evaluation of a quadratic equation $\Delta x_1(u)$. Fortunately, we can perform the same process to simplify this equation. Computing the difference between $\Delta x_1(u+h)$ and $\Delta x_1(u)$ as $\Delta x_2(u)$:

$$\begin{aligned}\Delta x_2(u) &= \Delta x_1(u+h) - \Delta x_1(u)\\ &= (3ah)(u+h)^2 + (3ah^2 + 2bh)(u+h) + (ah^3 + bh^2 + ch)\\ &\quad - [(3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)]\\ &= 3ahu^2 + 6ah^2u + 3ah^3 + (3ah^2 + 2bh)u + 3ah^3 + 2bh^2 + (ah^3 + bh^2 + ch)\\ &\quad - [(3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)]\\ &= 6ah^2u + (6ah^3 + 2bh^2)\end{aligned}$$

This changes our pseudocode to

```
u = 0;
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
dx2 = 6ah^3 + 2bh^2;
for ( i = 1; i <= n; i++)
{
    u += h;
    x += dx1;
    output(x);
    dx1 += dx2;
    dx2 = 6ah^2u + (6ah^3 + 2bh^2);
}
```

We can carry this one final step further to remove the linear equation for Δx_2. Computing the difference between $\Delta x_2(u+h)$ and $\Delta x_2(u)$ as $\Delta x_3(u)$:

$$\begin{aligned}\Delta x_3(u) &= \Delta x_2(u+h) - \Delta x_2(u)\\ &= 6ah^2(u+h) + (6ah^3 + 2bh^2)\\ &\quad - 6ah^2u + (6ah^3 + 2bh^2)\end{aligned}$$

$$\begin{aligned}
&= 6ah^2u + 6ah^3 + (6ah^3 + 2bh^2) \\
&\quad - 6ah^2u + (6ah^3 + 2bh^2) \\
&= 6ah^3
\end{aligned}$$

Our final code for forward differencing becomes

```
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
dx2 = 6ah^3 + 2bh^2;
dx3 = 6ah^3;
for ( i = 1; i <=n; i++ )
{
    x += dx1;
    output(x);
    dx1 += dx2;
    dx2 += dx3;
}
```

We have simplified our evaluation from 3 multiplies and 3 adds, down to 3 adds. We'll have to perform similar calculations for y and z, with differing deltas and a, b, c, and d values for each coordinate, giving a total of 9 adds for each point.

Note that forward differencing is only possible if the time steps between each point are equal. Because of this, we can't use it for animating along a curve, as time between frames may vary from frame to frame. In this case Horner's rule for our degree polynomial is the most efficient solution.

9.10.2 Midpoint Subdivision

Source Code
Library
IvCurves
Filename
IvBezier

An alternative method for generating points along a curve is to recursively subdivide the curve until we have a set of subcurves, each of which can be approximated by a line segment. This subdivision usually stops at pixel resolution if necessary. This may end up with a more accurate and more efficient representation of the curve than forward differencing since more curve segments will be generated in areas with high curvature (areas that we might cut across with forward differencing), and fewer in areas with lower curvature.

We can perform this subdivision by taking a curve $Q(u)$ and breaking it into two new curves $L(s)$ and $R(t)$, usually at the midpoint $Q(1/2)$. In this case, $L(s)$ is the subcurve of $Q(u)$ where $0 \leq u \leq 1/2$, and $R(t)$ is the subcurve where $1/2 \leq u \leq 1$. The parameters s and t are related to u by

$$s = 2u$$
$$t = 2u - 1$$

Each subcurve is then tested for relative "straightness" — if it can be approximated by a line segment, we stop subdividing, otherwise we keep going. The general algorithm looks like

```
void
RenderCurve( Q )
{
    if ( Straight( Q ) )
        DrawLine( Q(0), Q(1) );
    else
    {
        MidpointSubdivide( Q, &L, &R );
        RenderCurve( L );
        RenderCurve( R );
    }
}
```

There are a few ways of testing how straight a curve is. The most accurate is to measure the length of the curve and compare it to the length of the line segment between the curve's two extreme points. If the two lengths are within a certain tolerance ϵ, then we can say the curve is relatively straight. This assumes that we have an efficient method for computing the arc length of a curve. We discuss some ways of calculating this next.

Another method is to use the two endpoints and the midpoint (Figure 9.19a). If the distance between the midpoint and the line segment formed by the two endpoints is close to 0, then we can usually say that the curve is relatively close to a line segment. The one exception is when the curve crosses the line segment between the two endpoints (Figure 9.19b), which will

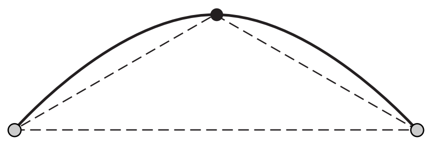


Figure 9.19a Midpoint test for curve straightness. Total distance from endpoints to midpoint (block dot) is compared to distance between endpoints.

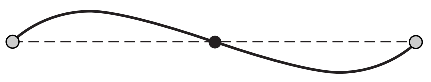


FIGURE 9.19b Midpoint test for curve straightness. Example of midpoint test failure.

result in a false positive when clearly the curve is not straight. To avoid the worst examples of this case, Parent [87] recommends performing forward differencing down to a certain level and only then adaptively subdividing.

The convex hull properties of the Bézier curve lead to a particularly efficient method for testing straightness, with no need of calculating a midpoint. If the interior control points are incident with the line segment formed by the two exterior control points, the area of the convex hull is 0, and the curve generated is itself a line segment. So for a cubic Bézier curve, we can test distance squared between the line segment formed by P_0 and P_3 and the two control points P_1 and P_2 (Figure 9.20). If both squared distances are less than some tolerance value, then we can say that the curve is relatively straight.

How we subdivide the curve if it fails the test depends on the type of curve. The simplest curves to subdivide are Bézier curves. To achieve this, we will generate new control points for each subcurve from our existing control points. So for a cubic curve, we will compute new control points L_1, L_2, L_3, and L_4 for curve L, and new control points R_1, R_2, R_3, and R_4 for curve R. These can be built by using a technique devised by de Casteljau. This method—known as *de Casteljau's method*—geometrically evaluates a Bézier curve at a given parameter u, and as a side effect creates the new control points needed to subdivide the curve at that point.

Figure 9.21 shows the construction for a cubic Bézier curve. L_0 and R_3 are already known: they are the original control points P_0 and P_3, respectively.

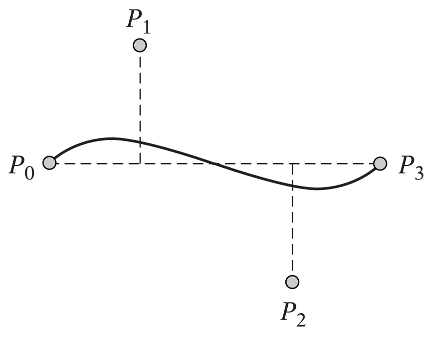


FIGURE 9.20 Test of straightness for Bézier curve. Measure distance of P_1 and P_2 to line segment P_0P_3.

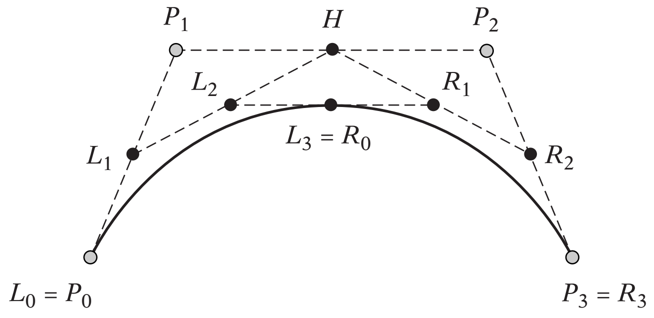


Figure 9.21 de Casteljau's method for subdividing Bézier curves.

Point L_1 lies on segment $\overline{P_0P_1}$ at position $(1-u)P_0 + uP_1$. Similarly, point H lies on segment $\overline{P_1P_2}$ at $(1-u)P_1 + uP_2$, and point R_2 at $(1-u)P_2 + uP_3$. We then linearly interpolate along the newly formed line segments $\overline{L_1H}$ and $\overline{HR_2}$ to form $L_2 = (1-u)L_1 + uH$ and $R_1 = (1-u)H + uR_2$. Finally, we split segment $\overline{L_2R_1}$ to find $Q(u) = L_3 = R_1 = (1-u)L_2 + uR_1$.

Using the midpoint to subdivide is particularly efficient in this case. It takes only 6 adds and 6 multiplies (to perform the division by 2):

```
L0 = P0;
R3 = P3;
L1 = (P0 + P1)*0.5f;
 H = (P1 + P2)*0.5f;
R2 = (P2 + P3)*0.5f;
L2 = (L1 + H)*0.5f;
R1 = (H + R2)*0.5f;
L3 = R0 = (L2 + R1)*0.5f;
```

Subdividing other types of curves, in particular B-splines, can be handled by using an extension of this method devised by Böhm [15]. More information on Böhm subdivision and knot insertion can be found in Bartels *et al.* [8].

9.10.3 Using OpenGL

Source Code
Library
IvCurves
Filename
IvUniformBSpline

If we're using OpenGL as our graphics API, we can take advantage of an interface that assists in the rendering of parametric curves, in particular those that can be emulated by the Bernstein basis. If a curve can be converted to a Bézier curve, then we can render it using this interface. Fortunately, this applies to any of the curves that we have discussed. The interface consists of two parts: setting up a Bézier evaluator for the curve, and then evaluating it at increasing parameter values to create the appropriate OpenGL rendering calls.

The first part is done by using one of the routines `glMap1f()` or `glMap1d()`. This sets up the data for an evaluator function of one parameter, which we can use to generate the curve to be rendered. There can be only one such evaluator at a time: calling `glMap1f()` or `glMap1d()` a second time will overwrite the previously defined values. The arguments are as follows:

```
glMap1{fd}(GLenum target, TYPE u1, TYPE u2, GLint stride,
           GLint order, const TYPE* points );
```

The `TYPE` in this case is either `float` or `double`, depending on whether we use `glMap1f()` or `glMap1d()`. The `target` argument indicates what kind of rendering data we want to create: positions, colors, normals, or texture coordinates. The standard for rendering a curve is to use `GL_MAP1_VERTEX_3`. Arguments `u1` and `u2` represent the minimum and maximum *u* values on the curve, respectively. The value of `stride` is the offset (in number of floating-point values) between each control point in the array `points`. The `order` of the curve is the degree of the curve plus 1, so a cubic curve has order 4. Finally, the array `points` are the control points for the curve.

An example of using `glMap1f()` to set up a simple Bézier curve is

```
IvVector3 controlPoints[] =
{
    IvVector3(0.0f, 0.0f, 0.0f),
    IvVector3(1.0f, 1.0f, 0.0f),
    IvVector3(2.0f, -1.0f, 0.0f),
    IvVector3(4.0f, 0.0f, 0.0f)
};

glMap1f( GL_MAP1_VERTEX_3, 0.0f, 1.0f, 3, 4,
         (float*) &controlPoints[0].x );
glEnable( GL_MAP1_VERTEX_3 );
```

Note that we call `glEnable()` to activate the evaluator so we can use it. To render the curve we need to evaluate it at increasing parameter values. We can do this in one of two ways: manually, which allows us more control over where the curve is actually evaluated, or automatically through OpenGL.

The manual method uses the routine `glEvalCoord1f()`. It takes a single argument `u`, and evaluates the curve at that parameter. Then, depending on the target value set in glMap1f(), it will make an OpenGL call for that particular data value. So if target equals GL_MAP1_VERTEX_3, it will internally call glVertex3(); for colors it will call glColor(); and so forth. How this is used depends on the graphics primitive set by glBegin(). So for example, to render a curve using line segments we might do

```
glBegin(GL_LINE_STRIP);
for (unsigned int i = 0; i < 32; ++i)
{
  glEvalCoord1f( (float)i/32.0f );
}
glEnd();
```

An alternative to this is to pass in an array of pregenerated parameter values, or

```
float params[32];
for (unsigned int i = 0; i < 32; ++i)
{
  params[i] = (float)i/32.0f;
}
...
glBegin(GL_LINE_STRIP);
glEvalCoord1fv( params );
glEnd();
```

Rather than generate the parameter values ourselves, we could let OpenGL do it for us. This requires a two part interface: one part that generates a set of equally spaced values, and one that uses the stored data to actually render the curve. The first has the format

```
void glMapGrid1{fd}(GLint n, TYPE u1, TYPE u2)
```

This will generate $n + 1$ equally spaced parameter values starting at $u1$ and at subsequent values of $i \cdot (u2 - u1)/n$. Like glMap1f() and glMap1d(), this can be set up once and reused over subsequent rendering passes but will be overwritten by the next call of glMapGrid1f() or glMapGrid1d().

To render using this set of parameters, we use the routine glEvalMesh1(), which has arguments

```
void glEvalMesh1(GLenum mode, GLint p1, GLint p2)
```

This will render parameters in the array from index p1 to p2 ($0 \leq p1 \leq p2 \leq n$), using primitive mode. The mode can be either GL_POINT or GL_LINE. The equivalent of applying both of these routines in sequence is

```
glBegin(mode);
for (unsigned int i = 0; i < n; ++i)
{
  glEvalCoord1f( u1 + (float)i*(u2 - u1)/(float)n );
}
glEnd();
```

All of these interfaces evaluate at parameters specified by the user, so they save only in cost of evaluation (potentially) and ease of interface. Designed particularly for uniformly spaced subdivision, they are not nearly as useful if we want to employ an adaptive subdivision method.

9.11 Controlling Speed Along a Curve

9.11.1 Moving at Constant Speed

Source Code
Demo
Speed Control

One common requirement for animation is that the object animated move at a constant speed along a curve. However, in most interesting cases, using a given curve directly will not achieve this. The problem is that in order to achieve variety in curvature, the first derivative must vary as well, and hence the distance we travel in a constant time will vary depending on where we start on the curve. For example, Figure 9.22 shows a curve subdivided at equal intervals of the parameter u. The lengths of the subcurves generated vary greatly from one to another.

Ideally, given a constant rate of travel r and time of travel t, we'll want to cover a distance of $s = rt$. So given a starting parameter u_1 on the curve, we

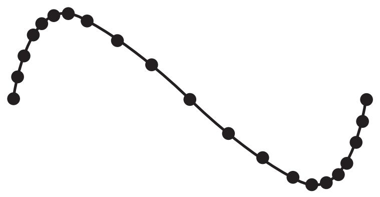


Figure 9.22 Parameter-based subdivision of curve, showing non-equal segment lengths.

want to find the parameter u_2 such that the distance along the curve, or *arc length*, between $Q(u_1)$ and $Q(u_2)$ equals s.

We'll discuss how to compute the arc length of a curve in the next section, but for now suppose we somehow have a function $G(u)$ that returns the length s from $Q(0)$ to $Q(u)$. So for the case where $u_1 = 0$, we can use the inverse function $G^{-1}(s)$ to determine the parameter u_2, given an input length s. This is known as a reparameterization by arc length. Unfortunately, in general the arc length function for a parameterized curve is impossible to invert in terms of a finite number of elementary functions, so numerical methods are used instead.

One way is to note that finding u_2 is equivalent to the problem of finding the solution u of the equation

$$s - \text{length}(u_1, u) = 0 \tag{9.16}$$

A method that allows us to solve this is Newton-Raphson root finding. Burden and Faires [17] present a derivation for this using the Taylor series expansion.

Suppose we have a function $f(x)$ where we want to find p such that $f(p) = 0$. We begin with a guess for p, which we'll call $\bar{x}$, such that $f'(\bar{x}) \neq 0$ and $|p - \bar{x}|$ is relatively small. In other words, $\bar{x}$ may not quite be p but it's a pretty good guess. If we use $\bar{x}$ as a basis for the Taylor series polynomial:

$$f(x) = f(\bar{x}) + (x - \bar{x})f'(\bar{x}) + \frac{1}{2}(x - \bar{x})^2 f''(\xi(x))$$

We assume that $\xi(x)$ is bounded by x and $\bar{x}$, so we can ignore the remainder of the terms. If we substitute p for x, then $f(p) = 0$ and

$$0 = f(\bar{x}) + (p - \bar{x})f'(\bar{x}) + \frac{1}{2}(p - \bar{x})^2 f''(\xi(x))$$

Since $|p - \bar{x}|$ is relatively small, we assume that $(p - \bar{x})^2$ is small enough that we can ignore it, and so

$$0 \approx f(\bar{x}) + (p - \bar{x})f'(x)$$

Solving for p gives

$$p \approx \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})} \tag{9.17}$$

This gives us our method. We make an initial guess $\bar{x}$ at the solution and use the result of equation 9.17 to get a more accurate result p. If p still isn't close enough, then we feed it back into the equation as $\bar{x}$ to get a still more accurate result, and so on until we reach a solution of sufficient accuracy or after a given number of iterations is performed.

For our initial guess in solving equation 9.16, Eberly [27] recommends taking the ratio of our traveled length to the total arc length of the curve and map it to our parameter space. Assuming our curve is normalized so that u is in [0, 1], then pseudocode for our root-finding method will look like

```
float FindParameterByDistance( float u1, float s )
{
    // ensure that we remain within valid parameter space
    if (s > ArcLength(u1,1.0f))
        return 1.0f;

    // get total length of curve
    float len = ArcLength(0.0f,1.0f);

    // make first guess
    float p = u1 + s/len;

    for (int i = 0; i < MAX_ITER; ++i)
    {
        // compute function value and test against zero
        float func = ArcLength(u1,p) - s;
        if ( fabsf(func) < EPSILON )
        {
            return p;
        }

         // perform Newton-Raphson iteration step
        p -= func/Length(Derivative(p));
    }

    // done iterating, return last guess
return p;
}
```

The first test ensures that the distance we wish to travel is not greater than the remaining length of the curve. In this case we assume that this is the last segment of a piecewise curve and just jump to the end. A more robust implementation should subtract the remaining length from the distance and restart at the beginning of the next segment.

A few other implementation notes are in order at this point. As we'll see, computing `ArcLength()` can be a nontrivial operation. Because of this, if we're going to be calling `FindParameterByDistance()` many times for a fixed curve, it is more efficient to precompute `ArcLength(0.0f,1.0f)` and use this stored value instead of recomputing it each time. Also, the constants `MAX_ITER` and `EPSILON` will need to be tuned depending on the type of curve and the number of iterations we can feasibly calculate due to performance constraints. Reasonable starting values for this tuning process are 32 for `MAX_ITER` and `1.0e-06f` for `EPSILON`.

There are two pieces missing in order to solve this completely: a derivative for the curve, and a function that computes arc length between two parameters. The first is easily derived from the definition of the curve, as we did for clamped and natural splines. The second is discussed in the next section.

9.11.2 Computing Arc Length

The most accurate method of computing the length of a smooth curve (see Appendix B) $Q(u)$ from $Q(a)$ to $Q(b)$ is to directly compute the line integral

$$s = \int_a^b \left\| Q'(u) \right\| du$$

Unfortunately, for most cubic polynomial curves, it is not possible to find an analytic solution to this integration. For quadratic curves, there is a closed form solution, but evaluating the resulting functions is more expensive than using a numerical method that gives similar accuracy. In any case, if we wish to vary our curve types, we would have to redo the calculation and so it is not always practical.

The usual approach is to use a numerical method to solve the integral. There are many methods, which Burden and Faires [17] cover in some detail. In this case the most efficient for its accuracy is Gaussian quadrature, since it attempts to minimize the number of function evaluations, which can be expensive. It approximates a definite integral from -1 to 1 by a weighted sum of unevenly spaced function evaluations, or

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} c_i f(x_i)$$

The actual c_i and x_i values depend on n and are carefully selected to give the best approximation to the integral. Appendix B tabulates values up to $n = 5$, and Burden and Faires [17] describe in detail how these are derived for arbitrary values of n.

The restriction that we have to integrate over $[-1, 1]$ is not a serious obstacle. For a general definite integral over $[a, b]$, we can remap to $[-1, 1]$ by

$$\int_a^b f(x)dx = \int_{-1}^{1} f\left(\frac{(b-a)t+b+a}{2}\right)\frac{b-a}{2}dt$$

Guenter and Parent [51] describe a method that uses Gaussian quadrature in combination with adaptive subdivision to get very efficient results when computing arc length. Similar to using adaptive subdivision for rendering, we cut the current curve segment in half. We use Gaussian quadrature to measure the length of each half, and compare their sum to the length of the entire curve, again computed using Gaussian quadrature. If the results are close enough, we stop and return the sum of lengths of the two halves. Otherwise, we recursively compute their lengths via subdivision.

There are other arc length methods that don't involve computing the integral in this manner. One is to subdivide the curve as we would for rendering, and use the sums of the lengths of the line segments created to approximate arc lengths at each of the subdivision points. We can create a sorted table of pairs (u_i, s_i), where u_i is the parameter for each subdivision, and s_i is the corresponding length at the point $Q(u_i)$. Since both u and len are monotonically increasing, we can sort by either parameter. An example of such a table can be seen in Table 9.1.

To find the length from the start of the curve for a given u, we search through the table to find the two neighboring entries with parameters u_k and u_{k+1} such that $u_k \leq u \leq u_{k+1}$. Since the entries are sorted, this can be handled efficiently by a binary search. The length can then be approximated by linearly

Table 9.1 Mapping Parameter Value to Arc Length

u	s
0.0	0.0
0.1	0.2
0.15	0.3
0.29	0.7
0.35	0.9
0.56	1.1
0.72	1.6
0.89	1.8
1.00	1.9

interpolating between the two entries:

$$s \approx \frac{u_{k+1} - u}{u_{k+1} - u_k} s_k + \frac{u - u_k}{u_{k+1} - u_k} s_{k+1}$$

A higher-order curve can be used to get a better approximation.

To find the length between two parameters a and b where $a \leq b$, we compute the length for each and subtract one from the other, or

$$\text{length}(Q, a, b) = \text{length}(Q, b) - \text{length}(Q, a)$$

We can also use Table 9.1 to solve our original reparameterization problem, which is to find u given a length s. In this case we invert the process and search for the two neighboring entries with lengths s_j and s_{j+1} such that $s_j \leq s \leq s_{j+1}$. Again, we can use linear interpolation to approximate the parameter u which gives us length s as

$$u \approx \frac{s_{j+1} - s}{s_{j+1} - s_j} u_j + \frac{s - s_j}{s_{j+1} - s_j} u_{j+1}$$

To find the parameter b given a starting parameter a and a length s, we compute the length at a and add that to s. We then use the preceding process with the total length to find parameter b.

The obvious disadvantage of this scheme is that it takes additional memory for each curve. However, it is simple to implement, somewhat fast, and does avoid the Newton-Raphson iteration needed with other methods.

If we are using cubic Bézier curves, we can use a method described by Gravesen [49]. First of all, given a parameter u we can subdivide the curve (using de Casteljau's method) to be the subcurve from $[0, u]$. The new control points for this new subcurve can be used to calculate bounds on the length. The length of the curve is bounded by the length of the chord $\overline{P_0P_3}$ as the minimum, and the sum of the lengths of the line segments $\overline{P_0P_1}$, $\overline{P_1P_2}$ and $\overline{P_2P_3}$ as the maximum. We can approximate the arc length by the average of the two, or

$$\begin{aligned}
L_{min} &= \|P_3 - P_0\| \\
L_{max} &= \|P_1 - P_0\| + \|P_2 - P_1\| + \|P_3 - P_2\| \\
L &\approx \frac{1}{2}(L_{min} + L_{max})
\end{aligned}$$

The error can be estimated by the square of the difference between the minimum and maximum:

$$\xi = (L_{max} - L_{min})^2$$

If the error is judged to be too large, then the curve can be subdivided and the length becomes the sum of the lengths of the two halves. Gravesen [49] states that for m subdivisions the error drops to 0 as 2^{-4m}.

9.11.3 Ease-In and Ease-Out

In our original equation for computing the desired distance to travel, $s = rt$, we assumed that we were traveling at a constant rate of speed. However, it is often convenient to have an adjustable rate of speed over the length of the curve. We can represent this by a general distance-time function $s(t)$, which maps a time value t to the total distance traveled from t_0. As an example, Figure 9.23 shows $s(t) = rt$ as a distance-time graph.

Other than traveling at a constant rate, the most common distance-time function is known as *ease-in/ease-out*. Here, we start at a zero rate of speed, accelerate up to a constant nonzero rate of speed in the middle, and then decelerate down again to a stop. This feels natural, as it approximates the need to accelerate a physical camera, move it, and slow it down to a stop. Figure 9.24 shows the distance-time graph for one such function.

Parent [87] describes two methods for constructing ease-in/ease-out distance-time functions. One is to use sinusoidal pieces for the acceleration/deceleration areas of the function and a constant velocity in the middle. The pieces are carefully chosen to ensure C^1 continuity over the entire function. The user specifies percentages of the interval that are used for acceleration and deceleration, represented by k_1 and k_2. If the curve is normalized over the interval [0, 1], then an object that moves along the curve will accelerate from

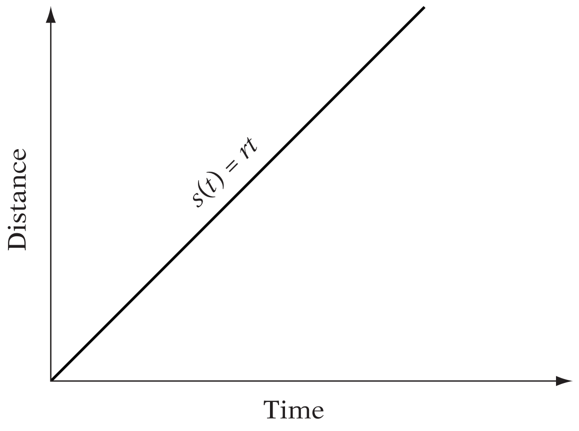


Figure 9.23 Example of distance-time graph: moving at constant speed.

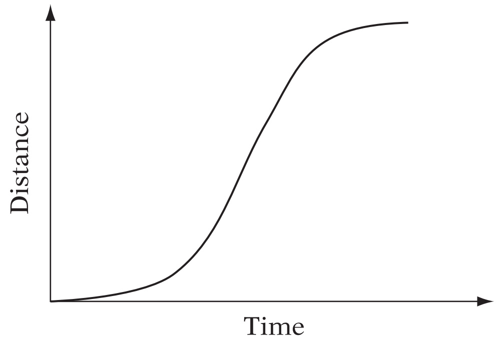


FIGURE 9.24 Example of distance-time graph. Ease-in/ease-out function.

$Q(0)$ to $Q(k_1)$, move at constant velocity until $Q(k_2)$, and then decelerate until the end at $Q(1)$. The piecewise function constructed is

$$\text{ease(t)} = \begin{cases} \left[k_1 \frac{2}{\pi}\left(\sin\left(\frac{t}{k_1}\frac{\pi}{2} - \frac{\pi}{2}\right) + 1\right)\right] / f & 0 \le t \le k_1 \\ \left[k_1 \frac{2}{\pi} + t - k_1\right] / f & k_1 \le t \le k_2 \\ \left[k_1 \frac{2}{\pi} + k_2 - k_1 + \left((1 - k_2)\frac{2}{\pi}\right)\sin\left(\frac{t-k_2}{1-k_2}\frac{\pi}{2}\right)\right] / f & k_2 \le t \le 1 \end{cases}$$

where $f = k_1 \frac{2}{\pi} + k_2 - k_1 + (1 - k_2)\frac{2}{\pi}$.

The second method involves setting a maximum velocity that we wish to attain in the center part of the function, and assumes that we move with constant acceleration in the opening and closing ease-in/ease-out areas. This gives a velocity-time curve as in Figure 9.25. By integrating this, we get a distance-time curve. By assuming that we start at the beginning of the curve,

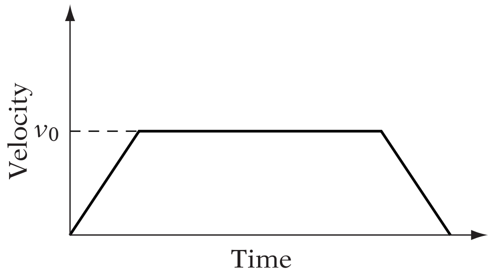


FIGURE 9.25 Example of velocity-time function. Ease-in/ease-out with constant acceleration/deceleration.

this gives us a piecewise curve with parabolic acceleration and deceleration:

$$\text{ease(t)} = \begin{cases} v_0 \frac{t^2}{2k_1} & 0 \le t \le k_1 \\ v_0 \frac{k_1}{2} + v_0(t - k_1) & k_1 \le t \le k_2 \\ v_0 \frac{k_1}{2} + v_0(k_2 - k_1) + \left[v_0 - \frac{1}{2}\left(v_0 \frac{t-k_2}{1-k_2} \right) \right](t - k_2) & k_2 \le t \le 1 \end{cases}$$

Which one we use depends on the needs of the application. The sinusoidal implementation has fewer parameters for the user to manage, but provides no control over the velocity reached during the constant velocity section.

9.12 Camera Control

Source Code
Demo
Camera Control

One common use for a parametric curve is as a path for controlling the motion of a virtual camera. In games this comes into play most often when setting up in-game cinematics, where we want to play a series of scripted events in engine while giving it a cinematic feel via the clever use of camera control. For example, we might want to have a camera track around a pair of characters as they dance about a room. Or, we might want to simulate a crane shot zooming from a far point of view right down into a close-up. While either of these could be done programmatically, it would be better to provide external control to the artist, who will most likely be setting up the shot. The artist sets the path for the camera — all the programmer needs to do is provide code to move the camera along the given path.

Determining the position of the camera isn't a problem. Given the start time t_s for the camera and the current time t_c, we compute the parameter $t = t_c - t_s$ and then use our time controls together with our curve description to determine the current position at $Q(t)$.

Computing orientation is another matter. The most basic option is to set a fixed orientation for the entire path. This might be appropriate if we are trying to create the effect of a panning shot but is rather limiting and somewhat static. Another way would be to set orientations at each sample time as well as positions, and interpolate orientations. Techniques for handling this situation are discussed in Chapter 10, but for now we'll assume that we don't have such sample orientations available.

A further possibility is to use the Frenet frame for the curve. This is an orthonormal frame with an origin of the current position on the curve, and a basis $\{\hat{\mathbf{u}}, \hat{\mathbf{v}}, \hat{\mathbf{w}}\}$ where $\hat{\mathbf{u}}$ points in the direction of the first derivative, $\hat{\mathbf{v}}$ points roughly in the direction of the second derivative, and $\hat{\mathbf{w}}$ is the cross product of the first two. The vector $\hat{\mathbf{u}}$ acts as our view direction vector, $\hat{\mathbf{v}}$ acts as our view side vector, and $\hat{\mathbf{w}}$ acts as our view up vector.

For any curve specified by the matrix form $Q(u) = \mathbf{UMG}$, we can easily compute the first derivative by using the form $\mathbf{Q}'(u) = \mathbf{U}'\mathbf{MG}$, where for a cubic curve

$$\mathbf{U}' = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix}$$

Similarly, we can compute the second derivative as $\mathbf{Q}''(u) = \mathbf{U}''\mathbf{MG}$ where

$$\mathbf{U}'' = \begin{bmatrix} 6u & 2 & 0 & 0 \end{bmatrix}$$

Setting $\mathbf{u} = \mathbf{Q}'(u)$, we can compute $\mathbf{v}$ using Gram-Schmidt orthogonalization:

$$\mathbf{v} = \mathbf{Q}''(u) - \frac{\mathbf{u} \cdot \mathbf{Q}''(u)}{\mathbf{u} \cdot \mathbf{u}}\mathbf{u}$$

Finally, $\mathbf{w}$ is the cross product of these two:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v}$$

Normalizing $\mathbf{u}$,$\mathbf{v}$, and $\mathbf{w}$ gives us our orthonormal basis.

Parent [87] describes a few flaws with using the Frenet frame directly. First of all, the second derivative may be $\mathbf{0}$. We can handle this situation by interpolating between two frames on either side of our current location. Since the second derivative is zero, or near zero, the first derivative won't be changing much, so we're really interpolating between two frames in $\mathbb{R}^2$. This consists of finding the angle between them and interpolating along that angle (Figure 9.26). The one flaw with this is that when finding these frames we're still using $\mathbf{Q}''$, which may be near zero and hence lead to floating-point issues. In particular, if we are moving with linear motion, there will be no valid neighboring values for estimating $\mathbf{Q}''$.

Then, too, it assumes that the second derivative exists for all values of t, namely, that $Q(t)$ is C^2 continuous. Many of the curves we've discussed,

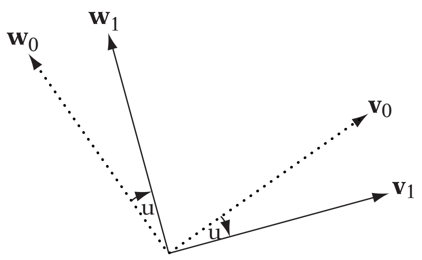


FIGURE 9.26 Interpolating between two path frames.

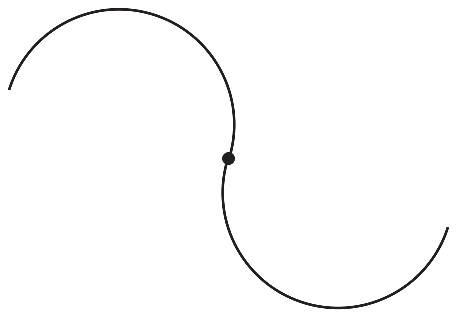


FIGURE 9.27 Frame interpolation issues. Discontinuity of second derivative at point.

in particular the piecewise curves, do not meet this criterion. In such cases the camera will rather jarringly change orientation. For example, suppose we have two curve segments as seen in Figure 9.27, where the second derivative instantly changes to the opposite direction at the join between the segments. In the Frenet frame for the first segment, the **w** vector points out of the page. In the second segment, it points into the page. As the camera crosses the join, it will instantaneously flip upside down. This is probably not what the animator had in mind.

Finally, we may not want to use the second derivative at all. For example, if we have a path that heads up and then down, like a hill on a roller coaster, the direction of the second derivative points generally down along that section of path. This means that our view up vector will end up parallel to the ground for that section of curve — again, probably not the intention of the animator.

One solution is to adopt the technique from Chapter 5 and use the first derivative as our view direction vector, computing the view up vector from this and the world up vector. The view side vector is the cross product of these two. This solves the problem, but does mean that if we have a fixed up-vector we can't roll our camera through a banking turn — its up vector will remain relatively aligned with the given up-vector.

A refinement of this is to allow user-specified up vectors at each sample position, which default to the world up-vector. The program would interpolate between these up vectors just as it interpolates between the positions. Alternatively, the user could set a path $U(t)$ that is used to calculate the up vector: $\mathbf{v}_{up} = U(t) - Q(t)$. The danger here is that the user may specify two up vectors of opposing directions that end up interpolating to $\mathbf{0}$, or an up vector that aligns with the view direction vector, which would lead to a cross product of $\mathbf{0}$. If the user is allowed this kind of flexibility, recovery cases and some sort of error message will be needed.

We can take this one step further; separate our view direction from the Frenet frame and use our familar look-at point method, again from Chapter 5.

The choice of what we use as our look-at point can depend on the camera effect desired. For example, we might pick a fixed point on the ground and then perform a fly-by. We could use the position of an object, or the centroid of positions for a set of objects. We could set an additional path, and use the position along that path at our current time, to give the effect of a moving point of view without tying it to a particular object.

Another possibility is to look ahead along our current path a few steps in time, as if we were following an object a few seconds ahead of us. So if we're at position $Q(t)$, we use as our look-at point the position $Q(t + \delta t)$. In this situation, we have to be sure to reparameterize the curve based on arc length, because otherwise the distance $\|Q(t) - Q(t + \delta t)\|$ may change depending on where we are on the curve, which may lead to odd changes in the view direction.

An issue with this technique is that it may make the camera seem clairvoyant, which can ruin the drama in some situations. Also, if our curve is particularly twisty, looking ahead may lead to sudden changes in direction. We can smooth this by averaging a set of points ahead of our position on the curve. How separated the points are makes a difference: too separated and our view direction may not change much. Too close together and the smoothing effect will be nullified. It's usually best to make the amount of separation another setting available to the animator so that he or she can control the effect desired.

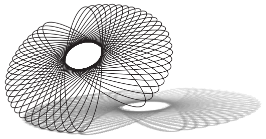
9.13 Chapter Summary

In this chapter we have touched on some of the issues involved with using parametric curves to aid in animation. We have discussed the most commonly used of the many possible curve types and how to render and subdivide these curves. Possible interfaces have been presented that allow animators and designers to create curves that can be used in the games they create. We have also covered some of the most common animation tasks: controlling travel speed along curves and maintaining a logical camera orientation.

For further reading, Rogers and Adams [95] and Bartels, Beatty, and Barsky [8] present much of this material in greater detail, in particular focusing on B-splines. Parent [87] covers the use of splines in animation, as well as additional animation techniques. Burden and Faires [17] have a chapter on interpolation and explain some of the numerical methods used with curves, in particular integration techniques and the Newton-Raphson method.

We have not discussed parametric surfaces, but many of the same principles apply: surfaces are approximated or interpolated by a grid of points and are usually rendered using a subdivision method. Rogers [94] is an excellent resource for understanding how NURBS surfaces, the most commonly used parametric surfaces, are created and used.

Chapter 10 Orientation Representation



10.1 Introduction

So far in our exploration of animation we've considered only interpolation of position. For a coordinate frame, this means only translating the frame in space, without considering rotation. This is fine for moving an object along a path, assuming we wanted it to remain oriented in the same manner as its base frame — generally, we don't. One possibility that we mentioned in the previous chapter is to align the forward vector of the object to the tangent vector of the curve, and use either the second derivative vector or an up vector to build a frame. This will work in general for airplanes and missiles, which tend to orient along their direction of travel. But suppose we want to interpolate a camera so that it travels sideways along a section of curve, or we're trying to model a helicopter, which can face in one direction while moving in another?

Another reason we want to interpolate orientation is for the purpose of animating a character. Usually characters are broken into a scene-graph–like data structure, called the *skeleton*, where each level, or *bone*, is stored at a constant translation from its parent, and only relative rotation is changed to move a particular node (Figure 10.1). So to move a forearm, for example, we rotate it relative to an upper arm (Figure 10.2). Accordingly, we can generate a set of keyframes for an animated character by storing a set of poses generated by setting rotations at each bone. To animate the character, we interpolate from one keyframe rotation to another.

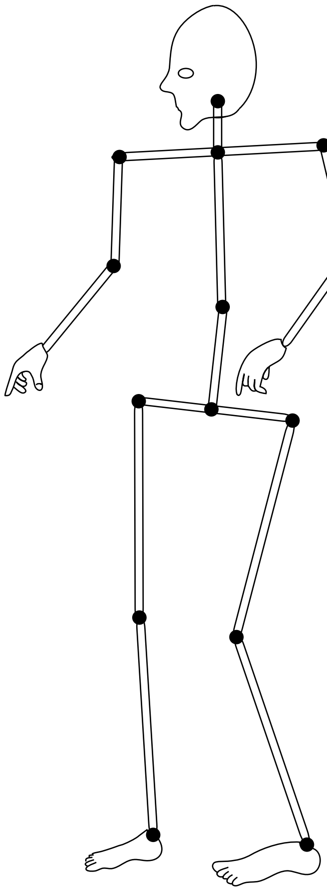


FIGURE 10.1 Example of skeleton showing relationship between bones.

As we shall see, when interpolating orientation we can't quite use the same techniques as we did with position. Rotational space doesn't behave in the same way as positional space; we'll be more concerned with interpolating along the surface of a sphere instead of along a line.

Before covering interpolation of orientation, we'll look at four different orientation formats and compare them on the basis of the following criteria:

- Represents orientation/rotation with a small number of values
- Can be concatenated efficiently to form new orientations/rotations
- Rotates points and vectors efficiently

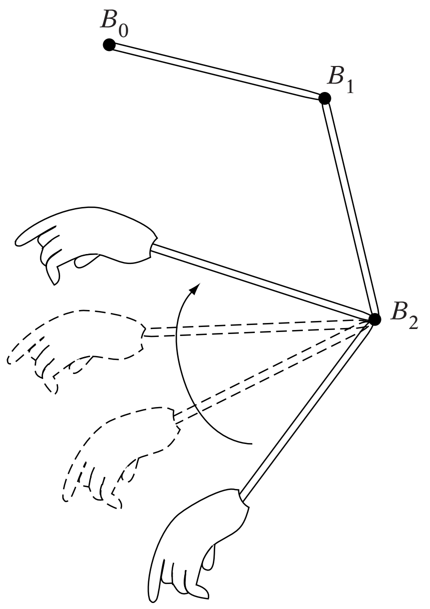


Figure 10.2 Relative bone poses for bending arm.

The first item is important if memory usage is an issue, either because we are working with a memory-limited machine such as a console, or because we want to store a large number of animations. In either case, any reduction in representation size means that we have freed-up memory that can be used for more animations, for more animation frames (leading to a smoother result), or for some other aspect of the game. Rotating points and vectors efficiently may seem like an obvious requirement, but one that merits mentioning; not all representations are good at this. Similarly, for some representations concatenation is not possible.

Once we've presented these different representations, we'll discuss interpolation, as well as the pros and cons of each representation for handling that task. As we'll see, there is no one choice that meets all of our requirements; each has its strengths and weaknesses in each area, depending on our implementation needs.

10.2 Rotation Matrices

Since we have been using matrices as our primary orientation/rotation representation, it is natural to begin our discussion with them.

For our first desired property, memory usage, matrices do not fare well. Euler's law of rotations states that the minimum number of values needed to represent a rotation in three dimensions is 3. The smallest possible rotation matrix requires 9 values, or 3 orthonormal basis vectors. It is possible to compress a rotation matrix, but in most cases this is not done unless we're sending data across a network. Even then it is better to convert to one of the more compact representations that we will present in the following sections, rather than compress the matrix.

However, for the second two properties, matrices do quite well. Concatenation is done through a matrix-matrix multiplication, which for two 3×3 matrices takes 27 multiplies and 18 additions, or 45 total operations. Rotating a vector is done through a matrix-vector multiply, which for a matrix and 3-vector takes 9 multiplies and 6 additions, or 15 total operations. On a SIMD processor, which can perform matrix and vector operations in parallel, both of these operations can be performed even faster. One such parallel processor can do matrix-vector multiplication in 3 instructions, and matrix-matrix multiplication in 9 instructions. Most graphics hardware has built-in circuitry that performs similarly. And as we've seen, 4×4 matrices can be useful for more than just rotation. Because of all these reasons, matrices continue to be useful despite their memory footprint.

10.3 Fixed and Euler Angles

10.3.1 Definition

We've just stated that the minimum number of values needed to represent a rotation in three-dimensional space is 3. As it happens, these 3 values can be the angles of three sequential rotations around a set of orthogonal axes. In Chapter 3, we used this as one means of building a generalized rotation matrix. Our chosen sequence of axes in this case was z-y-x, so the values $(0, \pi/4, \pi/2)$ represent a rotation of 0 radians around the z-axis, followed by a rotation of $\pi/4$ radians (or 45 degrees) around the y-axis, and concluding with a rotation of $\pi/2$ radians (90 degrees) around the x-axis. Angles can be less than 0 or greater than 2π, to represent reversed rotations and multiple rotations around a given axis. Note that we are using radians rather than degrees to represent our angles; either convention is acceptable, but the trigonometric functions used in C or C++ expect radians.

The order we've given is somewhat arbitrary, as there is no standard order that is used for the three axes. We could have used the sequence x-y-z, or z-x-y just as well. We can even duplicate one axis, so long as it is not the same axis in a row, so y-z-y is a valid sequence, while an axis rotation sequence such as z-y-y is not permitted. This is because duplicating an axis is redundant and doesn't add an additional degree of freedom.

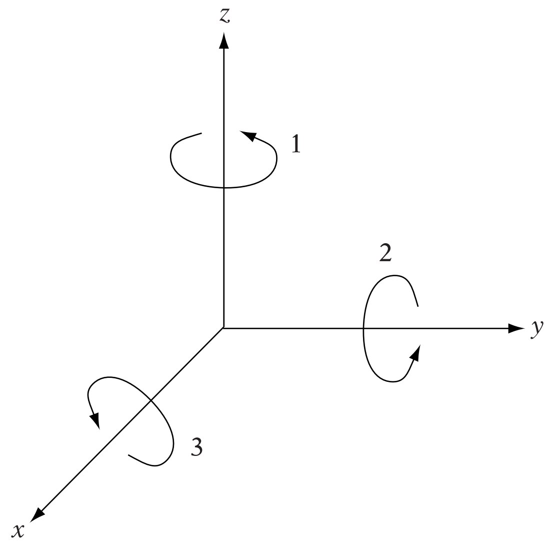


FIGURE 10.3 Order and direction of rotation for z-y-x fixed angles.

These rotations are performed around either the world axes or the object's local axes. When the angles represent world axis rotations, they are usually called *fixed angles* (Figure 10.3). The most convenient way to use fixed angles is to create an x-, y-, or z-rotation matrix for each angle and apply it in turn to our set of vertices. So an x-y-x fixed angle representation can be concatenated into a single matrix $\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_x$ in matrix form.

A sequence of local axis rotations, in turn, is said to consist of *Euler angles*[1]. The three Euler angles are commonly known as *roll*, *pitch*, and *heading*, after the three axes in a ship or an airplane. Heading is also sometimes referred to as *yaw*. Roll represents rotation around the forward axis, pitch rotation around a side axis, and heading rotation around the up axis (Figure 10.4). Whether a given roll, pitch, or heading rotation is around x, y, or z depends on how we've defined our coordinate frame. Suppose we are using a coordinate system where the z-axis represents up, the x-axis represents forward, and the y-axis represents left. Then heading is rotation around the z-axis, pitch is rotation around the y-axis, and roll is rotation around the x-axis. They are commonly applied in the order roll-pitch-heading, so the corresponding Euler angles for our case are x-y-z.

1. Just to be confusing, sometimes (a sequence of) rotations around world space axes are also referred to as Euler angles. Context should tell you which one the author means.

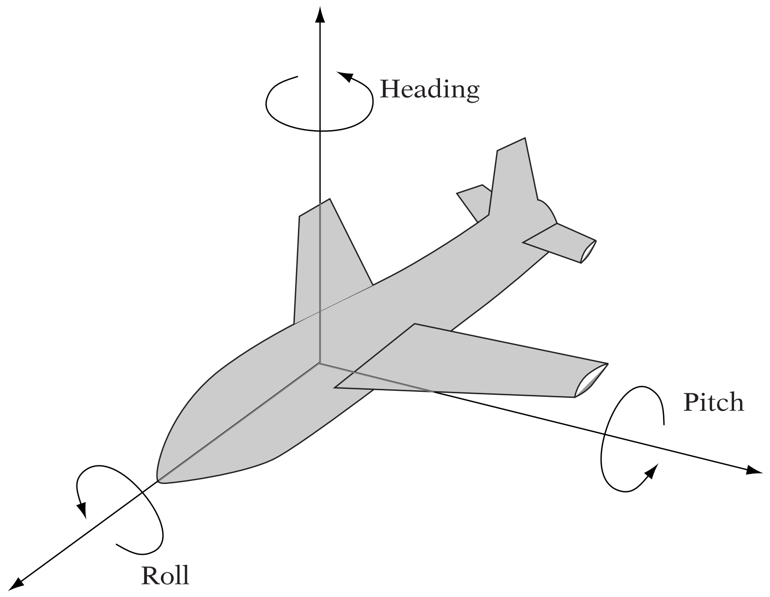


Figure 10.4 Roll, pitch, and yaw rotations relative to the local coordinate axes.

To create a rotation matrix which applies Euler angles, we concatenate in the reverse order of fixed angles. To see why, let's take our set of x-y-z Euler angles. We begin by applying the $\mathbf{R}_x$ matrix, to give us a rotation around x. We then want to apply a rotation around the object's local y-axis. However, because of the x rotation, the y-axis has been transformed to a new orientation. So if we concatenate as we normally would, our rotation will be about the transformed y-axis, which is not what we want. To avoid this, we transform by $\mathbf{R}_y$ first, then by $\mathbf{R}_x$, giving $\mathbf{R}_x\mathbf{R}_y$. The same is true for the z rotation: we need to rotate around z first to ensure we rotate around the local z-axis, not the transformed one. The resulting matrix is

$$\mathbf{R}_{Euler} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z$$

So x-y-z Euler angles are the same as z-y-x fixed angles.

10.3.2 Format Conversion

By concatenating three general axis rotation matrices and expanding out the terms, we can create a generalized rotation matrix. The particular matrix will depend on which axis rotations we're using and whether they are fixed or

Euler angles. For z-y-x fixed angles, or x-y-z Euler angles, the matrix looks like

$$\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z = \begin{bmatrix} CyCz & -CySz & Sy \\ SxSyCz + CxSz & -SxSySz + CxCz & -SxCy \\ -CxSyCz + SxSz & CxSySz + SxCz & CxCy \end{bmatrix}$$

where

$$\begin{aligned} Cx &= \cos\theta_x & Sx &= \sin\theta_x \\ Cy &= \cos\theta_y & Sy &= \sin\theta_y \\ Cz &= \cos\theta_z & Sz &= \sin\theta_z \end{aligned}$$

This should look familiar from Chapter 3. By combining terms appropriately, this takes 6 transcendentals, 12 multiplies, and 4 adds to compute.

When possible, we can save some instructions by computing each sine and cosine using a single `sincos()` call. This function is not supported on all processors, or even in all math libraries, so we have provided a wrapper function `IvSinCosf()` (accessible by including `IvMath.h`) that will calculate it depending on the platform. In any case, because we can't be guaranteed of its availability, we will assume that the function doesn't exist when computing our instruction count.

We can convert from a matrix back to a possible set of fixed angles by inverting this process. Note that since we'll be using inverse trigonometric functions there are multiple resulting angles. We'll also be taking a square root, the result of which could be positive or negative. Hence, there are multiple possibilities of Euler or fixed angles for a given matrix—the best we can do is find one. Assuming we're using z-y-x fixed angles, we can see that $\sin\theta_y$ is equal to $\mathbf{R}_{02}$. Finding $\cos\theta_y$ can be done by using the identity $\cos\theta_y = \sqrt{1 - \sin^2\theta_y}$. The rest falls out from dividing quantities out of the first row and last column of the matrix, so

$$\begin{aligned} \sin\theta_y &= \mathbf{R}_{02} \\ \cos\theta_y &= \sqrt{1 - \sin^2\theta_y} \\ \sin\theta_x &= -\mathbf{R}_{12}/\cos\theta_y \\ \cos\theta_x &= \mathbf{R}_{22}/\cos\theta_y \\ \sin\theta_z &= -\mathbf{R}_{01}/\cos\theta_y \\ \cos\theta_z &= \mathbf{R}_{00}/\cos\theta_y \end{aligned}$$

Note that we have no idea whether $\cos\theta_y$ should be positive or negative, so we assume that it's positive. Also, if $\cos\theta_y = 0$, then the x and z axes have become aligned (see Section 10.3.5) and we can't distinguish between rotations around x and rotations around z. One possibility is to assume that rotation around z is 0, so

$$\begin{aligned}\sin\theta_z &= 0\\ \cos\theta_z &= 1\\ \sin\theta_x &= \mathbf{R}_{21}\\ \cos\theta_x &= \mathbf{R}_{11}\end{aligned}$$

Calling `arctan2()` for each sin/cos pair will return a possible angle in radians, generally in the range $[0, 2\pi)$. Note that we have lost one of the few benefits of fixed/Euler angles, which is that it can represent multiple rotations around an axis by using angles greater than 2π radians, or 360 degrees. We have also lost any notion of "negative" rotation.

Assuming that $\cos\theta_y$ is not 0, this will take 2 additions, 5 multiplies, 1 divide, and 4 transcendental functions. If it is 0, this takes 1 addition, 1 multiply, and 4 transcendentals.

10.3.3 Concatenation

Clearly, fixed and Euler angles meet our first criteria for a good orientation representation: they use the minimum number of values. However, they don't really meet the remainder of our requirements. First of all, they don't concatenate well. Adding angles doesn't work: applying $(\pi/2, \pi/2, \pi/2)$ twice doesn't end up at the same orientation as (π, π, π). The most straightforward method for concatenating two Euler or fixed angle triples is to convert each sequence of angles to a matrix, concatenate the matrix, and then convert the matrix back to Euler or fixed angles. In the worst case, this will take 24 additions, 34 multiplies, and 10 transcendentals, and will only give an approximate result, due to the ill-formed nature of the matrix to fixed/Euler conversion.

10.3.4 Vector Rotation

Euler and fixed angles also aren't the most efficient method for rotating vectors. Recall that to rotate a vector around z uses the formula

$$R_z(x, y, \theta) = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$$

So using the angles directly means for each axis, we compute a sine and cosine (2 transcendental calls) and then apply the preceding formula (4 multiplies and 2 adds). This is a total of 6 transcendental operations, 12 multiplies, and 6 adds. Even if we cache the sine and cosine values for a set of vectors, this is still more expensive than the 9 multiplies and 6 adds of a matrix multiply. So when rotating multiple vectors (the break-even point is 5 vectors), it's more efficient to convert to matrix format.

10.3.5 Other Issues

As if all of these disadvantages are not enough, the fatal blow is that in certain cases fixed or Euler angles can lose one degree of freedom. We can think of this as a mathematical form of *gimbal lock*. In aeronautic navigational systems, there is often a set of gyroscopes, or gimbals, which control the orientation of an airplane or rocket. Gimbal lock is a mechanical failure where one gimbal is rotated to the end of its physical range and it can't be rotated any further, thereby losing one degree of freedom. While in the virtual world, we don't have mechanical gyroscopes to worry about, a similar situation can arise.

Suppose we are using x-y-z fixed angles and we consider the case where, no matter what we use for the x and z angles, we will always rotate around the y-axis by 90 degrees. This rotates the original world x-axis — the axis we first rotate around — to be aligned with the world negative z-axis (Figure 10.5). Now any rotation we do with θ_z will subtract from any rotation to which we

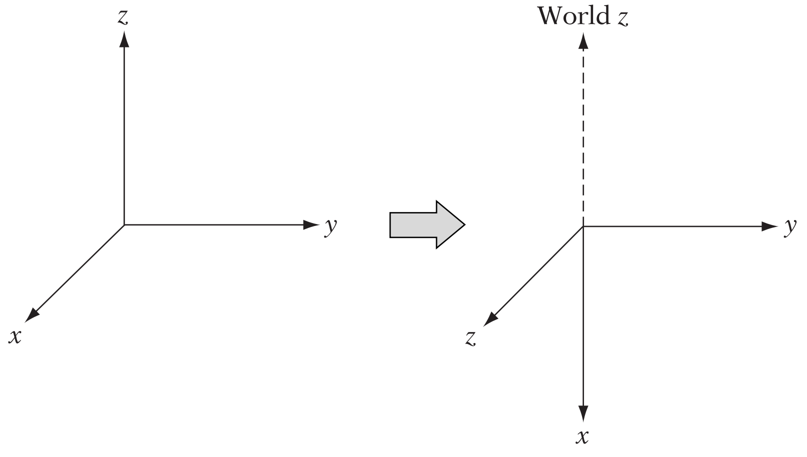


Figure 10.5 Demonstration of mathematical gimbal lock. A rotation of 90 degrees around y will lead to the local x-axis aligning with the -z world axis, and a loss of a degree of freedom.

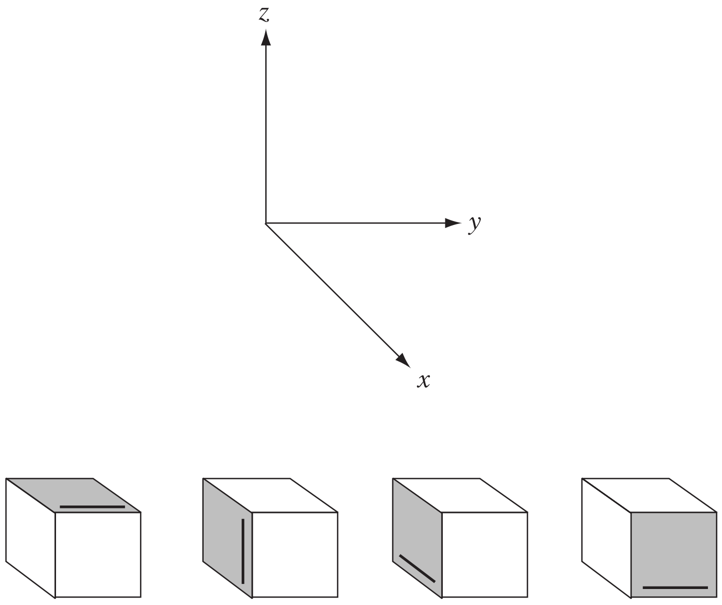


Figure 10.6 Effect of gimbal lock. Rotating the box around the world x axis, then world y axis, then the world z axis ends up having the same effect as rotating the box around just the y axis.

have applied θ_x. The combination of x- and z-rotations can be represented by one value $\theta_x - \theta_z$, applied as the initial x-axis rotation. For example in Figure 10.6, applying the fixed angles $(\pi/2, \pi/2, \pi/2)$ gets us back to our original $(0, \pi/2, 0)$. Instead of using $(\theta_x, \pi/2, \theta_z)$, we could just as well use $(\theta_x - \theta_z, \pi/2, 0)$ or $(0, \pi/2, \theta_z - \theta_x)$. We have effectively lost one degree of freedom.

To try this for yourself, take an object whose orientation can be clearly distinguished, like a book or CD case. From your point of view, rotate the object clockwise 90 degrees around an axis pointing forward (roll). Now rotate the new top of the object away from you by 90 degrees (pitch). Now rotate the object counterclockwise 90 degrees around an axis pointing up (heading). The result is the same as pitching the object downward 90 degrees (see Figure 10.6).

Still, in some cases fixed or Euler angles do provide an intuitive representation for orientation. For example, in a hierarchical system it is very intuitive to define rotations at each joint as a set of Euler angles and to constrain certain axes to remain fixed. An elbow or knee joint, for instance, could be considered a set of Euler angles with two constraints and only one axis available for applying rotation. It's also easy to set a range of angles so that the joint doesn't bend too far one way or the other. However, these limited advantages are not enough to outweigh the problems with fixed/Euler angles. So in most cases, fixed/Euler angles are used as a means to semi-intuitively set other representations (being aware of the dangers of gimbal lock, of course), and our library will be no exception.

10.4 Axis-Angle Representation

10.4.1 Definition

Recall from Chapter 3 that we can represent a general rotation in $\mathbb{R}^3$ by an axis of rotation, and the amount we rotate around this axis by an angle of rotation. Therefore, we can represent rotations in two parts: a 3-vector **r** that lies along the axis of rotation, and a scalar θ which corresponds to a counterclockwise rotation around the axis, if the axis is pointing towards us. Usually, a normalized vector $\hat{\mathbf{r}}$ is used instead, which constrains the four values to three degrees of freedom, corresponding to the three degrees of freedom necessary for 3D rotations.

Generating the axis-angle rotation that takes us from one normalized vector $\hat{\mathbf{v}}$ to another vector $\hat{\mathbf{w}}$ is straightforward (Figure 10.7). The angle of rotation is the angle between the two vectors:

$$\theta = \arccos(\hat{\mathbf{v}} \cdot \hat{\mathbf{w}}) \tag{10.1}$$

The two vectors lie in the plane of rotation, and so the axis of rotation is perpendicular to both of them:

$$\mathbf{r} = \hat{\mathbf{v}} \times \hat{\mathbf{w}} \tag{10.2}$$

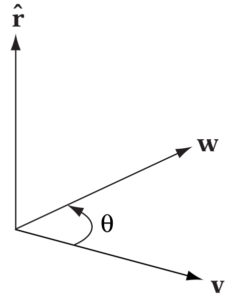


Figure 10.7 Axis-angle representation. Rotation around **r** by angle θ rotates **v** into **w**.

Normalizing **r** gives us $\hat{\mathbf{r}}$. Near-parallel vectors may cause us some problems either because the dot product is near 0, or normalizing the cross product ends up dividing by a near-zero value. In those cases, we set θ to 0, and $\hat{\mathbf{r}}$ to any arbitrary, normalized vector.

10.4.2 Format Conversion

To convert an axis-angle represention to a matrix, we can use the derivation from Chapter 3:

$$\mathbf{R}_{\hat{\mathbf{r}}\theta} = \begin{bmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{bmatrix} \tag{10.3}$$

where

$$\begin{aligned} \hat{\mathbf{r}} &= (x, y, z) \\ c &= \cos\theta \\ s &= \sin\theta \\ t &= 1 - \cos\theta \end{aligned}$$

This will take 12 multiplies, 10 adds, and 2 transcendental evaluations.

Converting from a matrix to the axis-angle format has similar issues as the fixed angle format, since opposing vectors $\hat{\mathbf{r}}$ and $-\hat{\mathbf{r}}$ can be used to generate the same rotation by rotating in opposite directions, and multiple angles (0 and 2π, for example) applied to the same axis can rotate to the same orientation. The following method is from Eberly [29].

We begin by computing the angle. The sum of the diagonal elements, or *trace* of a rotation matrix **R**, is equal to $2\cos\theta + 1$, where θ is our angle of rotation. This gives us an easy method for computing θ:

$$\theta = \arccos\left(\frac{1}{2}(\text{trace}(\mathbf{R}) - 1)\right)$$

There are three possibilities for θ. If θ is 0, then we can use any arbitrary unit vector as our axis. If θ lies in the range $(0, \pi)$, then we can compute the axis by using the formula

$$\mathbf{R} - \mathbf{R}^T = 2\sin\theta\mathbf{S} \tag{10.4}$$

where $\mathbf{S}$ is a skew symmetric matrix of the form

$$\mathbf{S} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix}$$

The values x, y, and z in this case are the components of our axis vector $\hat{\mathbf{r}}$. So we can compute $\mathbf{r}$ as $(R_{21}-R_{12}, R_{02}-R_{20}, R_{10}-R_{01})$, and normalize to get $\hat{\mathbf{r}}$.

If θ equals π, then $\mathbf{R} - \mathbf{R}^T = \mathbf{0}$, which doesn't help us at all. In this case, we can use another formulation for the rotation matrix, which only holds if $\theta = \pi$:

$$\mathbf{R} = \mathbf{I} + 2\mathbf{S}^2 = \begin{bmatrix} 1-2y^2-2z^2 & 2xy & 2xz \\ 2xy & 1-2x^2-2z^2 & 2yz \\ 2xz & 2yz & 1-2x^2-2y^2 \end{bmatrix}$$

The idea is that we can use the diagonal elements to compute the three axis values. By subtracting appropriately, we can solve for one term, and then use that value to solve for the other two. For example, $R_{00}-R_{11}-R_{22}+1$ expands to

$$\begin{aligned} R_{00} - R_{11} - R_{22} + 1 &= 1 - 2y^2 - 2z^2 - 1 + 2x^2 + 2z^2 - 1 + 2x^2 + 2y^2 + 1 \\ &= 4x^2 \end{aligned}$$

So

$$x = \frac{1}{2}\sqrt{R_{00} - R_{11} - R_{22} + 1} \tag{10.5}$$

and consequently,

$$\begin{aligned} y &= \frac{R_{01}}{2x} \\ z &= \frac{R_{02}}{2x} \end{aligned}$$

To avoid problems with numeric precision and square roots of negative numbers, we'll choose the largest diagonal element as the term that we'll solve for. So if R_{00} is the largest diagonal element, we'll use the preceding equations. If R_{11} is the largest, then

$$\begin{aligned} y &= \frac{1}{2}\sqrt{R_{11} - R_{00} - R_{22} + 1} \\ x &= \frac{R_{01}}{2y} \end{aligned}$$

$$z = \frac{R_{12}}{2y}$$

Finally, if R_{22} is the largest element we use

$$z = \frac{1}{2}\sqrt{R_{22} - R_{00} - R_{11} + 1}$$

$$x = \frac{R_{02}}{2z}$$

$$y = \frac{R_{12}}{2z}$$

Computing the angle takes 1 multiply, 3 additions, and 1 `arccos()`. If θ is 0, then we're done. If $0 < \theta < 2\pi$, then computing the axis takes an additional 6 multiplies, 5 adds, 1 divide, and 1 transcendental (we can save the divide if we have an `InvSquareRoot()` function available), for a total of 7 multiplies, 8 additions, 1 divide, and 2 transcendentals. For $\theta = 2\pi$, the total is 3 multiplies, 6 additions, 1 divide, and 2 transcendentals.

10.4.3 Concatenation

Concatenating two axis-angle representations is not straightforward. One method is to convert them to matrices, multiply, and then convert back to the axis-angle format. Converting the pair of axis-angle rotations to matrices takes 24 multiplies, 20 adds, and 4 transcendental functions. Added to that operation count is the matrix multiplication, which takes 27 multiplies and 18 adds. Finally, in the worst case converting back takes 7 multiplies, 8 additions, 1 divide, and 2 transcendentals, for a total of 58 multiplies, 46 additions, 1 division, and 6 transcendentals.

10.4.4 Vector Rotation

For the rotation of a vector $\mathbf{v}$ by the axis-angle representation $(\hat{\mathbf{r}}, \theta)$, we can use the Rodrigues formula that we derived in Chapter 3:

$$R\mathbf{v} = \cos\theta\mathbf{v} + [1 - \cos\theta](\mathbf{v}\cdot\hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v})$$

If we precompute $\cos\theta$ and $\sin\theta$ and reuse intermediary values, we can compute this in 19 multiplies and 12 additions, or 31 operations. We can improve this slightly by using the identity

$$\begin{aligned}\hat{\mathbf{r}} \times (\hat{\mathbf{r}} \times \mathbf{v}) &= (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - (\hat{\mathbf{r}} \cdot \hat{\mathbf{r}})\mathbf{v} \\ &= (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - \mathbf{v}\end{aligned}$$

and substituting to get an alternate Rodrigues formula:

$$R\mathbf{v} = \mathbf{v} + (1 - \cos\theta)[\hat{\mathbf{r}} \times (\hat{\mathbf{r}} \times \mathbf{v})] + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v})$$

This will require only 18 multiplies and 12 additions, assuming that $(1 - \cos\theta)$ and $\sin\theta$ are precomputed. In both these cases, the trade-off is whether to store the results of the transcendental functions and thereby use more memory, or compute them every time and lose speed. The answer will depend on the needs of the implementation.

When rotating two or more vectors, it is more efficient to convert the axis-angle format to a matrix and then multiply. Assuming that we haven't pregenerated the sine and cosine values, this takes $12 + 9x$ multiplies, $10 + 6x$ adds, and 2 transcendental evaluations, where x is the number of vectors we're transforming. The break-even point is two vectors, so if you're only transforming one vector, don't bother converting; otherwise, use a matrix.

10.4.5 Section Summary

While being a useful way of thinking about rotation, the axis-angle format still has some problems. Concatenating two axis-angle representations is extremely expensive. And unless we store two additional values, rotating vectors requires computing transcendental functions, which is not very efficient either. Our next representation encapsulates some of the useful properties of the axis-angle format, while providing a more efficient method for concatenation. It precomputes the transcendental functions and uses them to rotate vectors in nearly equivalent time to the axis-angle method. Because of this, we have not explicitly provided an implementation for the axis-angle format.

10.5 Quaternions

10.5.1 Definition

Source Code
Library
IvMath
Filename
IvQuat

The final orientation representation we'll consider could be considered a variant of the axis-angle representation, and in fact it's often simplest to think of it that way. It is called the *quaternion* and was created by the Irish mathematician Sir William Hamilton [54] in the 19th century and introduced to computer graphics by Ken Shoemake [98] in the 1980s. Quaternions require only four values, they don't have problems of gimbal lock, the mathematics for concatenation is relatively simple, and if properly constructed they can be used to rotate vectors in a reasonably efficient manner.

Hamilton's general formula for a quaternion **q** is as follows:

$$\mathbf{q} = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

The quantities ***i***, ***j***, and ***k*** can be thought of as the standard basis for all quaternions, so it is common to write a quaternion as just

$$\mathbf{q} = (w, x, y, z)$$

The $x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ part of the quaternion is akin to a vector in $\mathbb{R}^3$, so a quaternion can also be written as

$$\mathbf{q} = (w, \mathbf{v})$$

where w is called the scalar part and **v** is called the vector part.

Frequently, we'll want to use vectors in combination with quaternions. To do so, we'll zero out the scalar part and set the vector part equal to our original vector. So the quaternion corresponding to a vector **u** is

$$\mathbf{q}_\mathbf{u} = (0, \mathbf{u})$$

Other than terminology, we aren't that concerned about Hamilton's intentions for generalized quaternions, because we are only going to consider a specialized case discovered by Arthur Cayley [18]. He determined that if you took a quaternion with four values (as just described), treated it like a fourth-dimensional vector and normalized it, it can be used to describe pure rotations. Later on, Courant and Hilbert [21] determined the relationship between normalized quaternions and the axis and angle representation.

10.5.2 Rotation Quaternions

Since we want to represent rotations, we will be normalizing all of our quaternions. In a normalized quaternion, w can be thought of as representing the angle of rotation θ. More specifically, $w = \cos(\theta/2)$. The vector **v** represents the axis of rotation, but normalized and scaled by $\sin(\theta/2)$. So $\mathbf{v} = \sin(\theta/2)\hat{\mathbf{r}}$. For example, suppose we wanted to rotate by 90 degrees around the z-axis.

Our axis is $(0, 0, 1)$ and half our angle is $\pi/4$ (in radians). The corresponding quaternion components are

$$
\begin{aligned}
w &= \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \\
x &= 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0 \\
y &= 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0 \\
z &= 1 \cdot \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}
\end{aligned}
$$

giving us a final quaternion of

$$\mathbf{q} = \left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right)$$

So why reformat our previously simple axis and angle to this somewhat strange representation? As we'll see shortly, pre-cooking the data in this way allows us to concatenate, rotate vectors, and interpolate with ease.

As with the axis-angle format, it is often useful to create a quaternion that rotates a vector $\mathbf{v}_1$ into another vector $\mathbf{v}_2$, although in this case we'll use a different approach. Melax [76] provides a method that uses trigonometric identities for efficiency's sake, and also avoids some issues with numerical error when $\mathbf{v}_1$ and $\mathbf{v}_2$ are nearly collinear.

We begin by normalizing $\mathbf{v}_1$ and $\mathbf{v}_2$. We'll define $\mathbf{r}$ as $\hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2$, and d as $\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2$. We know that $\|\mathbf{r}\| = \sin\theta$ and $d = \cos\theta$, but what we want is $\sin(\theta/2)$ and $\cos(\theta/2)$. From half-angle trigonometric identities, we know that

$$
\begin{aligned}
\cos\left(\frac{\theta}{2}\right) &= \sqrt{\frac{1+\cos\theta}{2}} \\
\sin\left(\frac{\theta}{2}\right) &= \sqrt{\frac{1-\cos\theta}{2}}
\end{aligned}
$$

We could use these to compute w, and then normalize $\mathbf{r}$ and multiply by $\sin(\theta/2)$. However, by normalizing and then re-scaling by $\sin(\theta/2)$, we are actually scaling by

$$
\begin{aligned}
\frac{\sin\left(\frac{\theta}{2}\right)}{\sin\theta} &= \frac{\sqrt{\frac{1-\cos\theta}{2}}}{\sqrt{1-\cos^2\theta}} \\
&= \sqrt{\frac{1-\cos\theta}{2(1-\cos^2\theta)}}
\end{aligned}
$$

$$
\begin{aligned}
&= \sqrt{\frac{1-\cos\theta}{2(1+\cos\theta)(1-\cos\theta)}} \\
&= \sqrt{\frac{1}{2(1+\cos\theta)}} \\
&= \frac{1}{\sqrt{2(1+\cos\theta)}}
\end{aligned}
$$

So we can precompute s, where $s = \sqrt{2(1+\cos\theta)}$, and scale $\mathbf{r}$ by $1/s$ to compute $\mathbf{v}$ directly. And as it happens $w = s/2$, since

$$
\begin{aligned}
s/2 &= \frac{\sqrt{2(1+\cos\theta)}}{2} \\
&= \sqrt{\frac{2(1+\cos\theta)}{4}} \\
&= \sqrt{\frac{(1+\cos\theta)}{2}} \\
&= \cos\left(\frac{\theta}{2}\right) \\
&= w
\end{aligned}
$$

The final formulas for computing the quaternion are

$$
\begin{aligned}
\mathbf{r} &= \hat{\mathbf{v}}_1 \times \hat{\mathbf{v}}_2 \\
s &= \sqrt{2(1+\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2)} \\
\mathbf{q} &= (s/2, \mathbf{r}/s)
\end{aligned}
$$

Our class implementation for quaternions looks like

```
class IvQuat
{
public:
    // constructor/destructor
    inline IvQuat() {}
    inline IvQuat( float_w, float _x, float _y, float _z )  :
        w(_w), x(_x), y(_y), z(_z)
    {
    }
    IvQuat(const IvVector3& axis, float angle);
    IvQuat(const IvVector3& v1, const IvVector3& v2);
```

```
        explicit IvQuat(const IvVector3& vector);
        inline ~IvQuat() {}

        // member variables
        float x, y, z, w;
    };
```

Much of this follows from what we've already discussed. We can set our quaternion values directly, use an axis-angle format, compute rotation from two vectors, or explicitly use a vector. Recall that in this last case, we use the vector to set our x, y, and z terms, and set w to 0.

10.5.3 Format Conversion

Converting from axis-angle format to a quaternion takes 1 multiply for the half-angle, 2 function calls for the sine and cosine, and 3 multiplies to scale the axis vector. To convert back, we take the arccos of w to get half the angle, and then use $\sqrt{1 - w^2}$ to get the length of $\mathbf{v}$ so we can normalize it. The full conversion is

$$\begin{aligned}\theta &= 2\arccos(w)\\ \|\mathbf{v}\| &= \sqrt{1-w^2}\\ \hat{\mathbf{r}} &= \mathbf{v}/\|\mathbf{v}\|\end{aligned}$$

This takes 1 addition, 5 multiplies, 1 divide, and 2 transcendental functions.

Converting a normalized quaternion to a 3×3 rotation matrix takes the following form:

$$\mathbf{M_q} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix} \tag{10.6}$$

If the quaternion is not normalized, we need to scale the matrix by

$$\frac{1}{w^2+x^2+y^2+z^2}$$

There is a lot of duplication of terms here, so on a serial processor this can be done with 12 multiplies and 12 adds if normalized, plus an additional 3 adds, 4 multiplies, and a floating-point divide if not normalized. The following is derived from Shoemake [99]:

```
IvMatrix33&
IvMatrix33::Rotation( const IvQuat& q )
{
    float s, xs, ys, zs, wx, wy, wz, xx, xy, xz, yy, yz, zz;

    // if q is normalized, s = 2.0f
    s = 2.0f/( q.x*q.x + q.y*q.y + q.z*q.z + q.w*q.w );

    xs = s*q.x;     ys = s*q.y;     zs = s*q.z;
    wx = q.w*xs;    wy = q.w*ys;    wz = q.w*zs;
    xx = q.x*xs;    xy = q.x*ys;    xz = q.x*zs;
    yy = q.y*ys;    yz = q.y*zs;    zz = q.z*zs;

    mV[0] = 1.0f - (yy + zz);
    mV[3] = xy - wz;
    mV[6] = xz + wy;

    mV[1] = xy + wz;
    mV[4] = 1.0f - (xx + zz);
    mV[7] = yz - wx;

    mV[2] = xz - wy;
    mV[5] = yz + wx;
    mV[8] = 1.0f - (xx + yy);

    return *this;

}   // End of Rotation()
```

If we have a parallel vector processor that can perform fast matrix multiplication, another way of doing this is to generate two 4 × 4 matrices and multiply them together:

$$\mathbf{M_q} = \begin{bmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{bmatrix} \begin{bmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{bmatrix}$$

If the quaternion is normalized, the product will be the homogeneous rotation matrix corresponding to the quaternion.

To convert a matrix to a quaternion, we can use an approach that combines our matrix to axis-angle conversion with our method of creating a quaternion from two vectors. Recall that the trace of a rotation matrix is $2\cos\theta + 1$, where θ is our angle of rotation. Assuming that the trace is greater than 0, if we add 1 to this and take the square root, we get the same s as when we rotated one vector into another:

$$s = \sqrt{2(\cos\theta + 1)}$$

so

$$w = s/2$$

as before. From equation 10.4, we know that the vector $\mathbf{r} = (R_{21} - R_{12}, R_{02} - R_{20}, R_{10} - R_{01})$ will have length $2\sin\theta$. The value s is equal to $\sin\theta/\sin(\theta/2)$, so we need to scale $\mathbf{r}$ by $1/(2s)$ to give it length $\sin(\theta/2)$, or

$$x = (R_{21} - R_{12})/(2s)$$
$$y = (R_{02} - R_{20})/(2s)$$
$$z = (R_{10} - R_{01})/(2s)$$

If the trace of the matrix is less than zero, then this will not work. We'll need to use an approach similar to when we extracted the axis from a rotation matrix. By taking the largest diagonal element and subtracting the elements from it, we can derive an equation to solve for a single axis component (e.g., equation 10.5). Using that value as before, we can then compute the other quaternion components from the elements of the matrix.

So if the largest diagonal element is R_{00}:

$$x = \frac{1}{2}\sqrt{R_{00} - R_{11} - R_{22} + 1}$$
$$y = \frac{R_{01} + R_{10}}{4x}$$
$$z = \frac{R_{02} + R_{20}}{4x}$$
$$w = \frac{R_{21} - R_{12}}{4x}$$

If the largest diagonal element is R_{11}:

$$
\begin{aligned}
y &= \frac{1}{2}\sqrt{R_{11} - R_{00} - R_{22} + 1} \\
x &= \frac{R_{01} + R_{10}}{4y} \\
z &= \frac{R_{12} + R_{21}}{4y} \\
w &= \frac{R_{02} - R_{20}}{4y}
\end{aligned}
$$

And if the largest diagonal element is R_{22}:

$$
\begin{aligned}
z &= \frac{1}{2}\sqrt{R_{22} - R_{00} - R_{11} + 1} \\
x &= \frac{R_{02} + R_{20}}{4z} \\
y &= \frac{R_{21} + R_{12}}{4z} \\
w &= \frac{R_{10} - R_{01}}{4z}
\end{aligned}
$$

Converting from a fixed angle format to a quaternion requires creating a quaternion for each rotation around a coordinate axis, and then concatenating them together. For the z-y-x fixed angle format, the result is

$$
\begin{aligned}
w &= \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\cos\frac{\theta_z}{2} - \sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
x &= \sin\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\cos\frac{\theta_z}{2} + \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
y &= \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\cos\frac{\theta_z}{2} - \sin\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
z &= \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\sin\frac{\theta_z}{2} + \sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\cos\frac{\theta_z}{2}
\end{aligned}
$$

Converting a quaternion to fixed or Euler angles is, quite frankly, an awful thing to do. If it's truly necessary (e.g., for an interface) the simplest method is to convert the quaternion to a matrix, and extract the Euler angles from the matrix.

10.5.4 Addition and Scalar Multiplication

Like vectors, quaternions can be scaled and added componentwise. For both operations a quaternion acts just like a 4-vector, so

$$(w_1, x_1, y_1, z_1) + (w_2, x_2, y_2, z_2) = (w_1 + w_2, x_1 + x_2, y_1 + y_2, z_1 + z_2)$$
$$a(w, x, y, z) = (aw, ax, ay, az)$$

The algebraic rules for addition and scalar multiplication that apply to vectors and matrices apply here, so like them, the set of all quaternions is also a vector space. However, the set of normalized quaternions is not, since neither operation maintains unit length. Therefore, if we use one of these operations, we'll need to normalize afterwards to ensure that we're using a proper rotation quaternion.

We'll use scale primarily for normalization purposes, and addition will be used together with scale for linear interpolation. We'll also see another use for addition when we discuss using quaternions in physical simulation. The implementation of these operations is similar to that for vectors.

10.5.5 Negation

Negation is a subset of scale, but it's worth discussing separately. One would expect that negating a normalized quaternion would produce a quaternion that applies a rotation in the opposite direction—it would be the inverse. However, while it does rotate in the opposite direction, it also rotates around the negative axis. The end result is that a vector rotated by either quaternion ends up in the same place, but if one quaternion rotates by θ radians around $\hat{\mathbf{r}}$, its negation rotates $2\pi - \theta$ radians around $-\hat{\mathbf{r}}$. Figure 10.8 shows what this looks like on the rotation plane. The negated quaternion can be thought of as "taking the other way around," but both quaternions rotate the vector to the same orientation. This will cause some issues when we get to interpolation but can be handled by adjusting our values appropriately, which we'll discuss next. Otherwise, we can use $\mathbf{q}$ and $-\mathbf{q}$ interchangeably.

10.5.6 Magnitude and Normalization

As mentioned, we will normalize quaternions as if we were using 4-vectors. The magnitude of a quaternion is therefore as follows:

$$\|\mathbf{q}\| = \sqrt{(w^2 + x^2 + y^2 + z^2)}$$

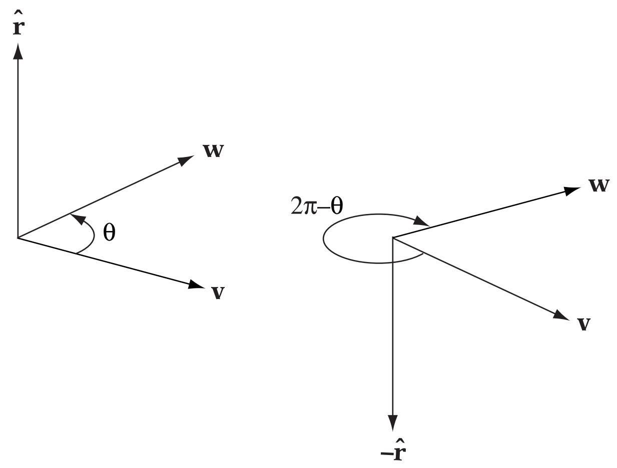


FIGURE 10.8 Comparing rotation performed by a normalized quaternion (left) with its negation (right).

A normalized quaternion $\hat{\mathbf{q}}$ is

$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{\|\mathbf{q}\|}$$

Since we're assuming that our quaternions are normalized, we'll forgo the use of the notation $\hat{\mathbf{q}}$ to keep our equations from being too cluttered.

10.5.7 Dot Product

The dot product of two quaternions should also look familiar:

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

As with vectors, this is still equal to the cosine of the angle between the quaternions, except that our "angle" is in four dimensions instead of the usual three. What this gives us is a way of measuring how "different" two quaternions are. If $\mathbf{q}_1 \cdot \mathbf{q}_2$ is close to 1 (remember that they're normalized), then they apply very similar rotations. Also, since we know that the negation of a quaternion performs the same rotation as the original, if the dot product is close to -1 the two still apply very similar rotations. So parallel normalized quaternions ($|\mathbf{q}_1 \cdot \mathbf{q}_2| \approx 1$) are similar. Correspondingly, orthogonal normalized quaternions ($\mathbf{q}_1 \cdot \mathbf{q}_2 = 0$) produce extremely different rotations.

10.5.8 Concatenation

As with matrices, if we wish to concatenate the transformations performed by two quaternions, we multiply them together to get a new quaternion. Expanding out the terms of the multiplication produces the following result:

$$\begin{aligned}&(w_2 + x_2\boldsymbol{i} + y_2\boldsymbol{j} + z_2\boldsymbol{k})(w_1 + x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k}) \\ &\quad = w_2w_1 + w_2x_1\boldsymbol{i} + w_2y_1\boldsymbol{j} + w_2z_1\boldsymbol{k} \\ &\qquad + x_2w_1\boldsymbol{i} + x_2x_1\boldsymbol{i}^2 + x_2y_1\boldsymbol{ij} + x_2z_1\boldsymbol{ik} \\ &\qquad + y_2w_1\boldsymbol{j} + y_2x_1\boldsymbol{ji} + y_2y_1\boldsymbol{j}^2 + y_2z_1\boldsymbol{jk} \\ &\qquad + z_2w_1\boldsymbol{k} + z_2x_1\boldsymbol{ki} + z_2y_1\boldsymbol{kj} + z_2z_1\boldsymbol{k}^2\end{aligned} \tag{10.7}$$

We define the products of the $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ quantities as follows:

$$\begin{matrix}\boldsymbol{ij} = \boldsymbol{k} & \boldsymbol{jk} = \boldsymbol{i} & \boldsymbol{ki} = \boldsymbol{j} \\ \boldsymbol{ji} = -\boldsymbol{k} & \boldsymbol{kj} = -\boldsymbol{i} & \boldsymbol{ik} = -\boldsymbol{j}\end{matrix}$$

and

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$$

Note that order does matter.

We can use these properties and well-known vector operations to simplify the product to

$$\mathbf{q}_2 \cdot \mathbf{q}_1 = (w_1w_2 - \mathbf{v}_1 \bullet \mathbf{v}_2, w_1\mathbf{v}_2 + w_2\mathbf{v}_1 + \mathbf{v}_2 \times \mathbf{v}_1)$$

Note that we've expressed this in a right-to-left order, like our matrices. This is because the rotation defined by $\mathbf{q}_1$ will be applied first, followed by the rotation defined by $\mathbf{q}_2$. We'll see this more clearly when we look at how we use quaternions to transform vectors. Also note the cross product; due to this, quaternion multiplication is also not commutative. This is what we expect with rotations; applying two rotations in one order does not necessarily provide the same result as applying them in the reverse order.

Multiplying two normalized quaternions does produce a normalized quaternion. However, due to floating-point error, it is wise to renormalize the result — if not after every multiplication, at least often and definitely before using the quaternion to rotate vectors.

A straightforward implementation of quaternion multiplication might look like

```
IvQuat operator*(IvQuat q2, IvQuat q1)
{
    IvVector3 v1(q1.x, q1.y, q1.z);
    IvVector3 v2(q2.x, q2.y, q2.z);

    float w = q1.w*q2.w - v1.Dot(v2);
    IvVector3 v = q1.w*v2 + q2.w*v1 + v2.Cross(v1);
    IvQuat q(w, v);

    return q;
}
```

Alternatively, we can unroll the operations to get

```
IvQuat operator*(IvQuat q2, IvQuat q1)
{
    w = q2.w*q1.w - q2.x*q1.x
        - q2.y*q1.y - q2.z*q1.z;
    x = q2.y* q1.z - q2.z*q1.y
        + q2.w*q1.x + q1.w*q2.x;
    y = q2.z*q1.x - q2.x*q1.z
        + q2.w*q1.y + q1.w*q2.y;
    z = q2.x*q1.y - q2.y*q1.x
        + q2.w*q1.z + q1.w*q2.z;
    return IvQuat(w,x,y,z);
}
```

This takes 16 multiplies and 12 additions, so concatenating two quaternions is actually faster than multiplying two matrices together.

An example of concatenating quaternions is the conversion from z-y-x fixed-angle format to a quaternion. The corresponding quaternions for each axis are

$$\mathbf{q}_z = \left(\cos\frac{\theta_z}{2}, 0, 0, \sin\frac{\theta_z}{2}\right)$$

$$\mathbf{q}_y = \left(\cos\frac{\theta_y}{2}, 0, \sin\frac{\theta_y}{2}, 0\right)$$

$$\mathbf{q}_x = \left(\cos\frac{\theta_x}{2}, \sin\frac{\theta_x}{2}, 0, 0\right)$$

Multiplying these together in the order $\mathbf{q}_x\mathbf{q}_y\mathbf{q}_z$ gives the result in Section 10.5.3.

10.5.9 Identity and Inverse

As with matrix products, there is an identity quaternion and, subsequently, there are multiplicative inverses. The identity quaternion is (1, 0, 0, 0), or $(1, \mathbf{0})$. Multiplying this by any quaternion $\mathbf{q} = (w, \mathbf{v})$ gives

$$\begin{aligned}\mathbf{q} \cdot (1, \mathbf{0}) &= (1 \cdot w - \mathbf{0} \bullet \mathbf{v}, 1\mathbf{v} + w\mathbf{0} + \mathbf{v} \times \mathbf{0}) \\ &= (w, \mathbf{v})\end{aligned}$$

In this case multiplication is commutative, so $\mathbf{q} \cdot (1, \mathbf{0}) = (1, \mathbf{0}) \cdot \mathbf{q} = \mathbf{q}$.

As with matrices, the inverse $\mathbf{q}^{-1}$ of a quaternion $\mathbf{q}$ is one such that $\mathbf{q}^{-1}\mathbf{q} = \mathbf{q}\mathbf{q}^{-1} = (1, \mathbf{0})$. If we consider a quaternion as rotating θ degrees counterclockwise around an axis $\hat{\mathbf{r}}$, then to undo the rotation we should rotate θ degrees clockwise around the same axis. This is the same as rotating $-\theta$ degrees counterclockwise: to create the inverse we negate the angle (Figure 10.9a). So if

$$(w, \mathbf{v}) = \left(\cos\left(\frac{\theta}{2}\right), \hat{\mathbf{r}} \sin\left(\frac{\theta}{2}\right)\right)$$

then

$$\begin{aligned}(w, \mathbf{v})^{-1} &= \left(\cos\left(-\frac{\theta}{2}\right), \hat{\mathbf{r}} \sin\left(-\frac{\theta}{2}\right)\right) \\ &= \left(\cos\left(\frac{\theta}{2}\right), -\hat{\mathbf{r}} \sin\left(\frac{\theta}{2}\right)\right) \\ (w, \mathbf{v})^{-1} &= (w, -\mathbf{v}) \qquad (10.8)\end{aligned}$$

At first glance, negating the vector part of the quaternion to reverse the rotation is counterintuitive. But after some thought this still makes

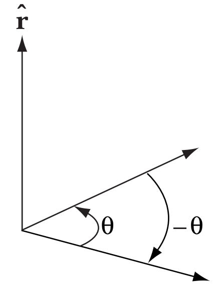


Figure 10.9a Relationship between quaternion and its inverse. Inverse rotates around same axis but negative angle.

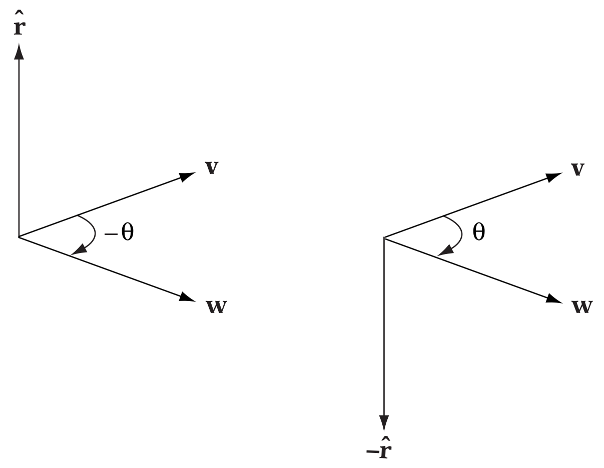


Figure 10.9b Rotation direction around axis by negative angle is same as rotation direction around negative axis by positive angle.

sense geometrically. A clockwise rotation around an axis turns in the same direction as a counterclockwise rotation around the negative of the axis (Figure 10.9b).

Equation 10.8 only holds if our quaternion is normalized. While it should be since we're working with rotation quaternions, if it is not then we need to scale by one over the length squared, or

$$\mathbf{q}^{-1} = \frac{1}{\|\mathbf{q}\|^2}(w, -\mathbf{v}) \tag{10.9}$$

Avoiding the floating-point divide in this case is another good reason to keep our quaternions normalized.

It bears repeating that the negative of a quaternion, where both w and $\mathbf{v}$ are negated, is not the same as the inverse. When applied to vectors, the negative actually rotates the vector to the same orientation but taking the other way around the axis.

10.5.10 Vector Rotation

If **qr** is used to concatenate two quaternions **q** and **r**, then for a vector **p** we might expect **qp** to rotate the vector by the quaternion, just as it does for a matrix. Unfortunately for intuition, this is not the case. For one thing, the result of this multiplication is not a vector (w will not be 0). The actual formula for rotating a vector by a quaternion is

$$R_{\mathbf{q}}\mathbf{p} = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \tag{10.10}$$

It may look like the effect of the operation is to perform the rotation and then undo it, but this is not the case. Remember that quaternion multiplication is not commutative, so if **q** is not the identity:

$$\mathbf{q}\mathbf{p}\mathbf{q}^{-1} \neq \mathbf{q}\mathbf{q}^{-1}\mathbf{p} = \mathbf{p}$$

We can use our rotation formula for axis and angle to show that equation 10.10 does rotate a vector. We begin by breaking it out into its component vector operations. Assuming that our quaternion is normalized, if we expand the full multiplication and combine terms, we get

$$R_{\mathbf{q}}\mathbf{p} = (2w^2 - 1)\mathbf{p} + 2(\mathbf{v} \cdot \mathbf{p})\mathbf{v} + 2w(\mathbf{v} \times \mathbf{p}) \tag{10.11}$$

Substituting $\cos(\theta/2)$ for w, and $\hat{\mathbf{r}}\sin(\theta/2)$ for $\mathbf{v}$:

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) = \left(2\cos^2\left(\frac{\theta}{2}\right) - 1\right)\mathbf{p} \;+\; 2\left(\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right) \cdot \mathbf{p}\right)\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right) \\ +\; 2\cos\left(\frac{\theta}{2}\right)\left(\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right) \times \mathbf{p}\right) \end{aligned}$$

Reducing terms and using the appropriate trigonometric identities, we end up with

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(\cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)\right)\mathbf{p} + 2\sin^2\left(\frac{\theta}{2}\right)(\hat{\mathbf{r}} \cdot \mathbf{p})\hat{\mathbf{r}} + 2\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\theta}{2}\right)(\hat{\mathbf{r}} \times \mathbf{p}) \\ &= \cos\theta\mathbf{p} + [1 - \cos\theta](\hat{\mathbf{r}} \cdot \mathbf{p})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}} \times \mathbf{p}) \end{aligned} \tag{10.12}$$

We see that equation 3.13 is equal to equation 10.12, so our quaternion multiplication—odd as it may look—does rotate a vector around an axis by a given angle.

In our code, we won't want to use the $\mathbf{q}\mathbf{p}\mathbf{q}^{-1}$ form, since performing both quaternion multiplications isn't very efficient. Instead, we'll use equation 10.11:

```
IvVector3
IvQuat::Rotate( const IvVector3& vector ) const
{
    ASSERT( IsUnit() );
    float vMult = 2.0f*(x*vector.x + y*vector.y + z*vector.z);
    float crossMult = 2.0f*w;
    float pMult = crossMult*w - 1.0f;

    return IvVector3( pMult*vector.x + vMult*x + crossMult*(y*vector.z - z*vector.y),
                      pMult*vector.y + vMult*y + crossMult*(z*vector.x - x*vector.z),
                      pMult*vector.z + vMult*z + crossMult*(x*vector.y - y*vector.x) );

}   // End of IvQuat::Rotate()
```

The operation count is 21 multiplications and 12 additions, which is still more than the 9 multiplications and 6 additions of matrix multiplication, but comparable to the 18 multiplications and 12 additions of Rodrigues' formula for axis-angle.

An alternate version:

$$R_{\mathbf{q}}\mathbf{p} = (\mathbf{v} \cdot \mathbf{p})\mathbf{v} + w^2\mathbf{p} + 2w(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p})$$

is useful for processors that have fast cross-product operations.

Neither of these formulas is as efficient as matrix multiplication, but for a single vector it is more efficient to perform these operations rather than convert the quaternion to a matrix and then multiply. However, if we need to rotate multiple vectors by the same quaternion, matrix conversion becomes worthwhile.

To see how concatenation of rotations works, suppose we apply a rotation from one quaternion followed by a second rotation from another quaternion. We can rearrange parentheses to get

$$\mathbf{q}(\mathbf{r}\mathbf{p}\mathbf{r}^{-1})\mathbf{q}^{-1} = (\mathbf{q}\mathbf{r})\mathbf{p}(\mathbf{q}\mathbf{r})^{-1}$$

As we see, concatenated quaternions will apply their rotation, one after the other. The order is right-to-left, as we have stated.

If we substitute $-\mathbf{q}$ in place of $\mathbf{q}$ in equation 10.10, we can see in another way how negating the quaternion doesn't affect rotation. By equation 10.8, $(-\mathbf{q})^{-1} = -\mathbf{q}^{-1}$, so

$$\begin{aligned} R_{-\mathbf{q}}(\mathbf{p}) &= -\mathbf{q}\mathbf{p}(-\mathbf{q})^{-1} \\ &= \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \end{aligned}$$

The two negatives cancel, and we're back with our familar result.

10.5.11 Quaternions and Transformations

Source Code Demo Transform

While quaternions are good for rotations, they don't help us much when performing translation and scale. Fortunately, we already have a transformation format that quaternions fit right into. Recall that in Chapter 3, instead of using a generalized 4×4 matrix for affine transformations, we used a single scale factor s, a 3×3 rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{t}$. Our formula for transformation was

$$\mathbf{p}' = \mathbf{R}(s\mathbf{p}) + \mathbf{t}$$

We can easily replace our matrix $\mathbf{R}$ with an equivalent quaternion $\mathbf{r}$, which gives us

$$\mathbf{p}' = \mathbf{r}(s\mathbf{p})\mathbf{r}^{-1} + \mathbf{t}$$

Concatenation using the quaternion is similar to concatenation with our original separated format, except that we replace multiplication by the rotation matrix with quaternion operations:

$$\begin{aligned} s' &= s_1 s_0 \\ \mathbf{r}' &= \mathbf{r}_1\mathbf{r}_0 \\ \mathbf{t}' &= \mathbf{t}_1 + \mathbf{r}_1(s_1\mathbf{t}_0)\mathbf{r}_1^{-1} \end{aligned}$$

Again, to add the translations, we first need to scale $\mathbf{t}_0$ by s_1 and then rotate by the quaternion $\mathbf{r}_1$.

As with lone quaternions, concatenation on a serial processor can be much cheaper in this format than using a 4×4 matrix. However, transformation of points is more expensive. As was the case with simple rotation, for multiple points it will be better to convert the quaternion to a matrix and transform them that way.

10.6 Interpolation

Our interpolation problem for position was to find a space curve — a function given a time parameter that returns a position — that passes through our sample points and maintains our desired curvature at each sample point. The same is true of interpolating orientation, except that our curve doesn't pass through a series of positions, but a series of orientations.

We can think of this as wanting to interpolate from one coordinate frame to another. If we were simply interpolating two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, we could find the rotation between them via the axis-angle representation $(\theta, \hat{\mathbf{r}})$, and then interpolate by rotating $\mathbf{v}_1$ as

$$\mathbf{v}(t) = R(t\theta, \hat{\mathbf{r}})\mathbf{v}_1$$

In other words, we linearly interpolate the angle from 0 to θ and continually apply the newly generated rotation to $\mathbf{v}_1$ to get our interpolated orientations. But for a coordinate frame, we need to interpolate three vectors simultaneously. We could use the same process for all three basis vectors, but it's not guaranteed that they will remain orthogonal. What we would need to do is find the overall rotation in axis-angle form from one coordinate frame to another, and then apply the process described. This is not a simple thing to do, and as it turns out there are better ways.

However, for fixed angles and axis-angle formats, we can use this to interpolate simple cases of rotation around a single axis. For instance, if we're interpolating from (90, 0, 0) to (180, 0, 0), we can linearly interpolate the first angle from 90 degrees to 180 degrees. Or, with an axis-angle format, if the rotation is from the reference orientation to another orientation, again we only need to interpolate the angle. Using this method also allows for interpolations over angles greater than 360 degrees. Suppose we want to rotate twice around the z-axis and represent this as only two values; we could interpolate between the two x-y-z fixed angles (0,0,0) and (0,0,4π). As we interpolate from 0 to 1, our object will rotate twice. More sample orientations are needed to do this with matrices and quaternions.

Source Code
Demo
Euler

But extending this to more complex cases does not work. Suppose we take as our starting orientation (0,90,0) and our ending orientation (90, 45, 90); if we linearly interpolate the angles to find a value halfway between them, we get (45, 67.5, 45). But this is wrong. One possible value which is correct is (90, 22.5, 90). The consequence of interpolating linearly from one sequence of Euler angles to another is that the object tends to sidle along, rotating around mostly one axis and then switching to rotations around mostly another axis, instead of rotating around a single axis, directly from one orientation to another.

We can mitigate this problem by defining Hermite or higher-order splines to better control the interpolation, and some 3D modeling packages provide output to do just that. However, you may not want to dedicate the space for the intermediary keyframes or the processing power to perform the spline interpolation, and it's still an approximation. For more complex cases, the only two formats that are practical are matrices and quaternions, and as we'll see this is where quaternions truly shine.

There are generally two approaches used when interpolating matrices and quaternions in games: linear interpolation and spherical linear interpolation. Both methods are usually applied piecewise between each orientation sample pair, and even though this will generate discontinuities at the sample points, the artifacts are rarely noticeable. While we will mention some ways of computing cubic curves, they generally are just too expensive for the small gain in visual quality.

10.6.1 Linear Interpolation

Source Code
Demo
LerpSlerp

By using the scalar multiplication and addition operations, we can linearly interpolate rotation matrices and quaternions just as we did vectors. Let's look at a matrix example first. Consider two orientations: one represented as the identity matrix and the other by a rotation of 90 degrees around the z-axis. Using linear interpolation to find the orientation halfway between the start and end orientations:

$$\frac{1}{2}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The result is not a well-formed rotation matrix. The basis vectors are indeed perpendicular, but they are not unit length. In order to restore this, we need to perform Gram-Schmidt orthogonalization, which is a rather expensive operation to perform every time we want to perform an interpolation.

With quaternions we run into some problems similar to those encountered with matrices. Suppose we perform the same interpolation, from the identity quaternion to a rotation of 90 degrees around z. This second quaternion is $(\sqrt{2}/2, 0, 0, \sqrt{2}/2)$. The resulting interpolated quaternion when $t = 1/2$ is

$$\begin{aligned} \mathbf{r} &= \frac{1}{2}(1, 0, 0, 0) + \frac{1}{2}\left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right) \\ &= \left(\frac{2+\sqrt{2}}{4}, 0, 0, \frac{\sqrt{2}}{4}\right) \end{aligned}$$

The length of $\mathbf{r}$ is 0.9239—clearly, not 1. Just as with matrices, we had to reorthogonalize after performing linear interpolation, with quaternions we will have to renormalize. Fortunately, this is a cheaper operation than orthogonalization, so quaternions have the advantage here.

In both cases, this happens because linear interpolation has the effect of cutting across the arc of rotation. If we compare a vector in one orientation with its equivalent in the other, we can get some sense of this. In the ideal case, as we rotate from one vector to another, the tips of the interpolated vectors trace an arc across the surface of a sphere (Figure 10.10). But as we can see in Figure 10.11, the linear interpolation is following a line segment

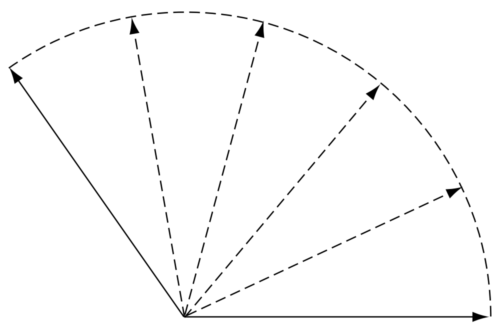


Figure 10.10 Ideal orientation interpolation, showing intermediate vectors tracing path along arc.

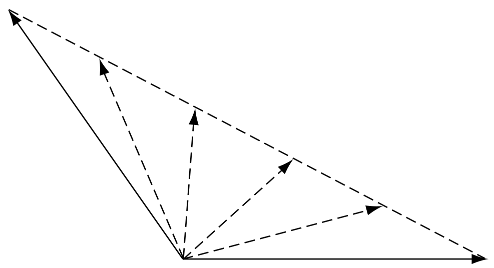


Figure 10.11 Linear orientation interpolation, showing intermediate vectors tracing path along line.

between the two tips of the vectors, which causes the interpolated vectors to shrink to a length of $\sqrt{2}/2$ at the halfway point, and then back up to 1.

Another problem with linear interpolation is that it doesn't move at a constant rate of rotation. Let's divide our interpolation at the t values 0, 1/4, 1/2, 3/4, and 1. In the ideal case, we'll travel one quarter of the arc length to get from orientation to orientation.

However, when we use linear interpolation, the t value doesn't interpolate along the arc, but along that chord which passes between the start and end orientations. When we divide the chord into four equal parts, the corresponding arcs on the surface of the sphere are no longer equal in length (Figure 10.12). Those closest to the center of interpolation are longer. The effect is that instead of moving at a constant rate of rotation throughout the interpolation, we will move at a slower rate at the endpoints and faster in the middle. This is particularly noticeable for large angles, as the figure shows. What we really want is a constant change in rotation angle as we apply a constant change in t.

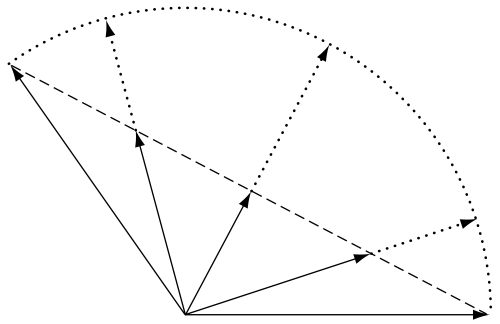


FIGURE 10.12 Effect of linear orientation interpolation on arc length when interpolating over 1/4 intervals.

One way to solve both of these issues is to insert one or two additional sample orientations and use quadratic or cubic interpolation. However, these are still only approximations to the spherical curve, and they involve storing additional orientation keyframes.

And even if you are willing to deal with nonconstant rotation speed, and eat the cost of orthogonalization, linear interpolation does create other problems. Suppose we use linear interpolation to find the orientation midway between these two matrices:

$$\frac{1}{2}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{10.13}$$

This is clearly not a rotation matrix, and no amount of orthogonalization will help us. The problem is that our two rotations (a rotation of $\pi/2$ around y and a rotation of $-\pi/2$ around y, respectively) produce opposing orientations—they're 180 degrees apart. As we interpolate between the pairs of transformed **i** and **k** basis vectors, we end up passing through the origin.

Quaternions are no less susceptible to this. Suppose we have a rotation of π radians counterclockwise around the y-axis, and a rotation of π radians clockwise around y. Interpolating the equivalent quaternions gives us

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(0, 0, 1, 0) + \frac{1}{2}(0, 0, -1, 0) \\ &= (0, 0, 0, 0)\end{aligned}$$

And again, no amount of normalization will turn this into a unit quaternion. The problem here is that we are trying to interpolate between two quaternions that are negatives of each other. They represent two rotations in the opposite direction that rotate to the *same* orientation. Rotating a vector 180 degrees counterclockwise around y will end up in the same place as rotating the same vector 180 degrees clockwise (or -180 degrees counterclockwise) around y. Even if we considered this an interpolation that runs entirely around the sphere, it is not clear which path to take—there are infinitely many.

This problem with negated quaternions shows up in other ways. Let's look at our first example again, interpolating from the identity quaternion to a rotation of $\pi/2$ around z. Recall that our result with $t = 1/2$ was $(2 + \sqrt{2}/4, 0, 0, \sqrt{2}/4)$. This time we'll negate the second quaternion, giving us a rotation of $-3\pi/2$ around z. We get the result

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(1, 0, 0, 0) + \frac{1}{2}\left(-\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}\right)\\ &= \left(\frac{2-\sqrt{2}}{4}, 0, 0, -\frac{\sqrt{2}}{4}\right)\end{aligned}$$

This new result is not the negation of the original result, nor is it the inverse. What is happening is that instead of interpolating along the shortest arc along the sphere, we're interpolating all the way around the other way, via the longest arc. This will happen when the dot product between the two quaternions is negative, so the angle between them is greater than 90 degrees.

This may be the desired result, but usually it's not. What we can do to counteract it is to negate the first quaternion and reinterpolate. In our example, we end up with

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(-1, 0, 0, 0) + \frac{1}{2}\left(-\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}\right)\\ &= \left(-\frac{2+\sqrt{2}}{4}, 0, 0, -\frac{\sqrt{2}}{4}\right)\end{aligned}$$

This gives us the negation of our original result, but this isn't a problem as it will rotate to the same orientation.

This also takes care of the case of interpolating from a quaternion to its negative, so for example, interpolating from $(0, 0, 1, 0)$ to $(0, 0, -1, 0)$:

$$\begin{aligned}\mathbf{r} &= -\frac{1}{2}(0, 0, 1, 0) + \frac{1}{2}(0, 0, -1, 0)\\ &= (0, 0, -1, 0)\end{aligned}$$

Negating the first one ends up interpolating to and from the same quaternion, which is a waste of processing power, but won't give us invalid results. Note that we will have to do this even if we are using spherical linear interpolation,

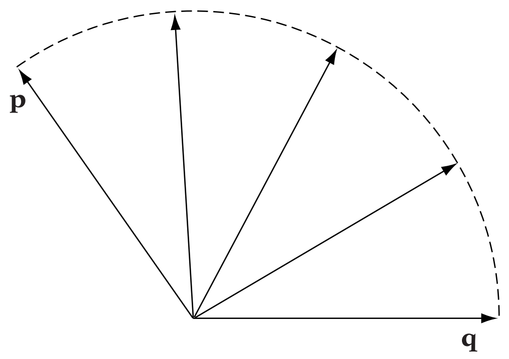


Figure 10.13 Effect of spherical linear interpolation when interpolating at quarter intervals. Interpolates equally along arc and angle.

which we will address next. All in all, it is better to avoid such cases by culling them out of our data beforehand.

10.6.2 Spherical Linear Interpolation

Source Code
Demo
LerpSlerp

To better solve the nonconstant rotation speed and normalization issues, we need an interpolation method known as *spherical linear interpolation* (usually abbreviated as *slerp*[2]). Slerp is similar to linear interpolation except that instead of interpolating along a line, we're interpolating along an arc on the surface of a sphere. Figure 10.13 shows the desired result. When using spherical interpolation at quarter intervals of t, we travel one quarter of the arc length to get from orientation to orientation. We can also think of slerp as interpolating along the angle, or in this case dividing the angle between the orientations into quarter intervals.

It can be shown that for two rotations P and Q, the slerp function is computed as follows:

$$\text{slerp}\,(P, Q, t) = P(P^{-1}Q)^t$$

For matrices, the question is how to take a matrix $\mathbf{R}$ to a power t. We can use a method provided by Eberly [29] as follows. Since we know that $\mathbf{R}$ is a rotation matrix, we can pull out the axis $\mathbf{v}$ and angle θ of rotation for the matrix as we've described, multiply θ by t to get a percentage of the rotation, and convert back to a matrix to get $\mathbf{R}^t$. This is an extraordinarily expensive operation, taking 77 multiplies, 58 additions, 1 division, and 6 transcendental functions.

2. As Shoemake [98] says, because it's fun.

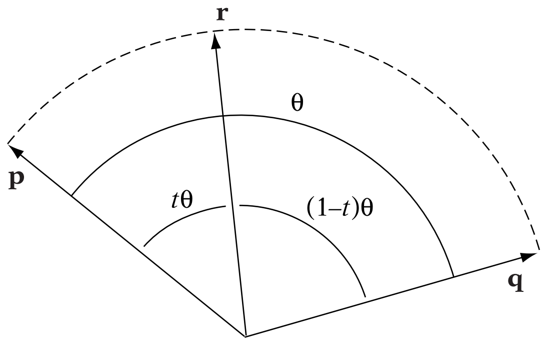


FIGURE 10.14 Construction for quaternion slerp. Angle θ is divided by interpolant t into subangles $t\theta$ and $(1-t)\theta$.

However, if we want to use matrices, it does give us the result we want of interpolating smoothly along arc length from one orientation to another.

For quaternions, we can derive slerp in another way, as demonstrated by Eberly [30]. Figure 10.14 shows the situation. We have two quaternions **p** and **q**, and an interpolated quaternion **r**. The angle between **p** and **q** is θ, calculated as $\theta = \arccos(\mathbf{p} \cdot \mathbf{q})$. Since slerp interpolates the angle, the angle between **p** and **r** will be a fraction of θ as determined by t, or $t\theta$. Similarly, the angle between **r** and **q** will be $(1-t)\theta$.

The general interpolation of **p** and **q** can be represented as

$$\mathbf{r} = a(t)\mathbf{p} + b(t)\mathbf{q} \tag{10.14}$$

The goal is to find two interpolating functions $a(t)$ and $b(t)$ so that they meet the criteria for slerp.

We determine these as follows. If we take the dot product of **p** with equation 10.14 we get

$$\begin{aligned} \mathbf{p} \cdot \mathbf{r} &= a(t)\mathbf{p} \cdot \mathbf{p} + b(t)\mathbf{p} \cdot \mathbf{q} \\ \cos(t\theta) &= a(t) + b(t)\cos\theta \end{aligned}$$

Similarly, if we take the dot product of **q** with equation 10.14 we get

$$\cos((1-t)\theta) = a(t)\cos\theta + b(t)$$

We have two equations and two unknowns. Solving for $a(t)$ and $b(t)$ gives us

$$a(t) = \frac{\cos(t\theta) - \cos((1-t)\theta)\cos\theta}{(1 + \cos^2\theta)}$$

$$b(t) = \frac{\cos((1-t)\theta) - \cos(t\theta)\cos\theta}{(1+\cos^2\theta)}$$

Using trigonometric identities, these simplify to

$$a(t) = \frac{\sin((1-t)\theta)}{\sin\theta}$$
$$b(t) = \frac{\sin(t\theta)}{\sin\theta}$$

Our final slerp equation is

$$\text{slerp}(\mathbf{p}, \mathbf{q}, t) = \frac{\sin((1-t)\theta)\mathbf{p} + \sin(t\theta)\mathbf{q}}{\sin\theta} \tag{10.15}$$

As we can see, this still is an expensive operation, consisting of three sines and a floating-point divide, not to mention the precalculation of the arccosine. But at 16 multiplications, 8 additions, 1 divide, and 4 transcendentals, it is much cheaper than the matrix method. It is clearly preferable to use quaternions versus matrices (or any other form) if you want to interpolate orientation.

One thing to notice is that as θ approaches 0—as **p** and **q** become close to equal—$\sin\theta$ and thus the denominator of the slerp function approaches 0. Testing for equality is not enough to catch this case, because of finite floating-point precision. Instead, we should test $\cos\theta$ before proceeding. If it's close to 1 ($> (1-\epsilon)$, say), then we use linear interpolation or *lerp* instead, since it's reasonably accurate for small angles and avoids the undesirable case of dividing by a very small number. It also has the nice benefit of helping our performance; lerp is much cheaper. In fact, it's generally best only to use slerp in the cases where it is obvious that rotation speed is changing.

Just as we do with linear interpolation, if we want to make sure that our path is taking the shortest route on the sphere and to avoid problems with opposing quaternions, we also need to test $\cos\theta$ to ensure that it is greater than 0 and negate the start quaternion if necessary. While slerp does maintain unit length for quaternions, it's still useful to normalize afterwards to handle any variation due to floating-point error.

Cubic Methods

Just as with lerp, if we do piecewise slerp we will have discontinuities at the sample orientations, which may lead to visible changes in orientation rather than the smooth curve we want. And just as we had available when interpolating points, there are cubic methods for interpolating quaternions.

One such method is *squad*, which uses the formula

$$\text{squad}(\mathbf{p}, \mathbf{a}, \mathbf{b}, \mathbf{q}, t) = \text{slerp}(\text{slerp}(\mathbf{p}, \mathbf{q}, t), \text{slerp}(\mathbf{a}, \mathbf{b}, t), 2(1-t)t) \qquad (10.16)$$

This is a modification of a technique of using linear interpolation to do Bezier curves, described by Böhm [16]. It performs a Bezier interpolation from **p** to **q**, using **a** and **b** as additional control points (or control orientations, to be more precise).

We can use similar techniques for other curve types, such as B-splines and Catmull-Rom curves. However, these methods usually are not used in games. They are more expensive than slerp (which is expensive enough), and most of the time the data being interpolated has been generated by an animation package or exists as samples from motion capture. Both of these tend to smooth the data out and insert additional samples at places where orientation is changing sharply, so smoothing the curve isn't that necessary. For those who are interested, Shoemake ([98],[99]) covers some of these spline methods in more detail.

10.6.3 Performance Improvements

Source Code
Demo
SlerpApprox

As we've seen, using slerp for interpolation, even when using quaternions, can take quite a bit of time — something we don't usually have. A typical character can have 20+ bones, all of which are being interpolated once a frame. If we have a team of characters in a room, there can be up to 20 characters being rendered at one time. The less time we spend interpolating, the better.

The simplest speedup is to use lerp all the time. It's very fast: ignoring the setup time (checking angles and adjusting quaternions) and normalization, only 12 basic floating-point operations are necessary on a serial processor, and on a vector processor this drops to 3. We do have the problems with inconsistent rotational speeds, but if our angles are small enough, or we're willing to live with it, lerp is a fine solution.

However, if we want better quality, then we need to try something else. One solution is to improve the speed of slerp. If we assume that we're dealing with a set of stored quaternions for keyframed animation, there are some things we can do here. First of all, we can precompute θ and $1/\sin\theta$ for each quaternion pair and store them with the rest of our animation data. In fact, if we're willing to give up the space, we could pre-scale **p** and **q** by $1/\sin\theta$ and store those values instead. This would mean storing up to two copies for each quaternion: one as the starting orientation of an interpolation and one as the ending orientation. Finally, if t is changing at a constant rate, we can use forward differencing to reduce our operations further. Shoemake [99] states that this can be done in 8 multiplies, 6 adds, and 2 table lookups for the two remaining sines.

If memory is plentiful and our frame rate is constant, then this approach can work well. However, neither of these is typically the case. Animation data usually takes up enough of our memory budget without nearly doubling its size, and frame rates can be variable, depending on what is being rendered or simulated. One possibility that doesn't have these restrictions is to approximate the most expensive operations – $1/\sin\theta$, $\sin(t\theta)$, and $\sin((1-t)\theta)$ — by splines. This can provide reasonable accuracy for less cost than the standard evaluation.

An alternate method is proposed by Jonathan Blow [14]. His idea is that instead of trying to change our interpolation method to fix our variable rotation speeds, we adjust our t values to counteract the variations. So in the section where an object would normally rotate faster with a constantly increasing t, we slow t down. Similarly, in the section where an object would rotate slower, we speed t up. Blow uses a cubic spline to perform this adjustment:

$$t' = 2kt^3 - 3kt^2 + (1+k)t$$

where

$$k = 0.5069269(1 - 0.7878088\cos\theta)^2$$

and $\cos\theta$ is the dot product between the two quaternions. This technique tends to diverge from the slerp result when $t > 0.5$, so Blow recommends detecting this case and swapping the two quaternions (i.e., interpolate from **q** to **p** instead of from **p** to **q**). In this way our interpolant always lies between 0 and 0.5.

The nice thing about this method is that it requires very few floating-point operations, doesn't involve any transcendental functions or floating-point divides, and fits in nicely with our existing lerp functions. It gives us slerp interpolation quality with close to lerp speed, which can considerably speed up our animation system.

10.7 Chapter Summary

In this chapter we've discussed four different representations for orientation and rotation: matrices, fixed/Euler angles, axis and angle, and quaternions. In the introduction we gave three criteria for our format: it may be informative to compare them along with their usefulness in interpolation.

As far as size, matrices are the worst at 9 values, and fixed/Euler angles the best at 3 values. However, quaternions and axis-angle representation are close to fixed/Euler angles at 4 values, and they avoid the problems engendered by gimbal lock.

For concatenation, quaternions take the fewest number of operations, followed closely by matrices, and then by axis-angle and fixed/Euler representations. The last two are hampered by not having low-cost methods for direct concatenation and so the majority of their expense is tied up in converting to a more favorable format.

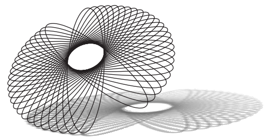
When transforming vectors, matrices are the clear winner. Assuming pre-cached sine and cosine data, fixed/Euler angles are close behind, while axis-angle and quaternions take a bit longer. However, if we don't pre-cache our data, the sine and cosine computations will probably take longer, and quaternions come in second.

Finally, fixed/Euler and axis-angle formats interpolate well only under simple circumstances. Matrices can be interpolated, but at significantly greater cost than quaternions. If you need to interpolate orientation, the clear choice is to use quaternions.

For further reading about quaternions, the best place to start is the writings of Shoemake, in particular [98]. Hamilton's original series of articles on quaternions [54] is in the public domain, and can be found by searching online. Courant and Hilbert [21] cover applications of quaternions, in particular to represent rotations. Finally, Eberly has an article [29] comparing orientation formats, and an entire chapter in his latest book [30] on quaternions, with additional material by Shoemake.

Part IV
Simulation

Chapter 11 Intersection Testing



11.1 Introduction

In the previous chapters, we have been primarily focused on manipulating and displaying our game objects in isolation. Whether we are rendering an object or animating it, we haven't been concerned with how it might be interacting with other objects in our scene. This is neither realistic nor interesting. For example, you are manipulating an object right now: this book. You can hold it in your hand, turn its pages, or drop it on the floor. In the latter case it stops reacting to you and starts reacting to the floor. If good gameplay derives from interesting interactions, then we need some way to detect when two game objects should be affecting one another and respond accordingly.

In this chapter we'll be concerned with a very straightforward question: how do we tell when two geometric entities are intersecting? This knowledge proves useful in many cases throughout a game engine. The most obvious is collision detection and response. Rather than have game objects pass through each other, we want them to push against each other and respond realistically. In the real world, this is a simple problem. Solid objects are solid; due to their physical properties, they just don't interpenetrate. But in the virtual world, we have to create these constraints ourselves. Despite the fact that we have completely defined the geometry of our game objects, we still need to provide methods to detect when they interpenetrate. Only when we have a way to handle this can we write the code to perform the proper response.

Another time when we want to detect when two geometric entities interpenetrate is when we want to cast a ray and see what objects it intersects.

One example of this we have seen already: detecting the object we've clicked on by generating a pick ray from a screen space mouse click, and determining the first object we hit with that ray. Another way this is used is in artifical intelligence. In order to simulate whether one AI agent can see another, we cast a ray from the first to the second and see if it intersects any objects. If not, then we can say that the first agent's target is in sight.

We have also mentioned a third use of object intersection before: determining which objects are visible in a view frustum so that we can do quick visibility culling. If they interpenetrate or are inside the frustum, then we go ahead to the rendering step; otherwise they get skipped. This can considerably speed up our rendering.

Due to the variety of shapes and primitives used in a standard game engine, finding intersections between all of the cases can get quite complex; a single chapter is not enough to cover everything. Instead, we'll cover five basic objects, some methods for improving performance and accuracy, and directions for improvement. We will also briefly discuss how to use these methods in a simple collision detection system, and how we can apply similar techniques to our ray casting and frustum culling problems. Details on more complex systems can be found in the recommended reading in the "Chapter Summary."

11.2 Closest Point and Distance Tests

As we'll find, object intersection tests can often be described more easily in terms of a distance computation between two primitives, such as a point and a line. In particular, we'll often want to know if the distance between two primitives is less than some value, such as a radius. So before we begin our discussion of determining intersections between bounding objects, we will cover a selection of useful methods for testing distances between certain geometric primitives.

Related to that topic is determining the closest points of approach between those same primitives; if we can find the closest points, the distance between the two primitives is the distance between those points. Because of this, we'll first consider closest point problems followed by how to calculate the distance between the same two primitives.

11.2.1 Closest Point on Line to Point

Source Code
Library
IvMath
Filename
IvLine3

Our first problem is illustrated in Figure 11.1: given a point Q, and a line defined by a point P and a vector $\mathbf{v}$, how do we find the point on the line Q' that is closest to Q? We approach this by examining the geometric relationships between the point and line. In particular, we notice that the dotted

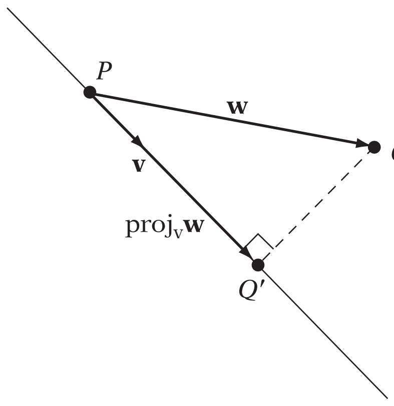


Figure 11.1 Closest point line.

line segment between Q and Q' is orthogonal to the line. This line segment corresponds to a line of projection: to find Q', we need to project Q onto the line.

To do this, we begin by computing the difference vector $\mathbf{w}$ between Q and P, or $\mathbf{w} = Q - P$. Then we project this onto $\mathbf{v}$, to get the component of $\mathbf{w}$ that points along $\mathbf{v}$. Recall that this is

$$\text{proj}_\mathbf{v}\mathbf{w} = \frac{\mathbf{w} \cdot \mathbf{v}}{\|\mathbf{v}\|^2}\mathbf{v}$$

We add this to the line point P to get our projected point Q', or

$$Q' = P + \frac{\mathbf{w} \cdot \mathbf{v}}{\|\mathbf{v}\|^2}\mathbf{v}$$

The equivalent code is

```
IvVector3 IvLine3::ClosestPoint(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;
    float vsq = mDirection.Dot(mDirection);
    float proj = w.Dot(mDirection);

    return mOrigin + (proj/vsq)*mDirection;
}
```

11.2.2 Line-Point Distance

Source Code
Library
IvMath
Filename
IvLine3

As before, we're given a point Q and a line defined by a point P and a vector $\mathbf{v}$. In this case, we want to find the distance between the point and the line. One way is to compute the closest point on the line and compute the distance between that and Q. A more direct approach is to use the Pythagorean theorem (Figure 11.2).

We note that $\mathbf{w} = Q - P$ can be represented as the sum of two vectors, one parallel to $\mathbf{v}$ ($\mathbf{w}_{\|}$) and one perpendicular ($\mathbf{w}_{\perp}$). These form a right triangle, so from Pythagoras, $\|\mathbf{w}\|^2 = \|\mathbf{w}_{\|}\|^2 + \|\mathbf{w}_{\perp}\|^2$. We want to know the length of $\mathbf{w}_{\perp}$, so we can rewrite this as

$$\begin{aligned}
\|\mathbf{w}_{\perp}\|^2 &= \|\mathbf{w}\|^2 - \|\mathbf{w}_{\|}\|^2 \\
&= \mathbf{w} \cdot \mathbf{w} - \left\| \frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \mathbf{v} \right\|^2 \\
&= \mathbf{w} \cdot \mathbf{w} - \left(\frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \right)^2 \mathbf{v} \cdot \mathbf{v} \\
&= \mathbf{w} \cdot \mathbf{w} - \frac{(\mathbf{w} \cdot \mathbf{v})^2}{\mathbf{v} \cdot \mathbf{v}}
\end{aligned}$$

Taking the square root of both sides will give us the distance between the point and the line.

The equivalent code is

```
float IvLine3::DistanceSquared(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;
```

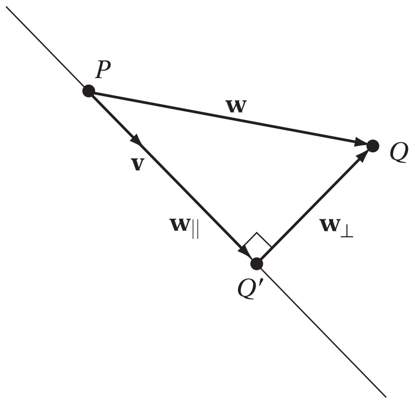


Figure 11.2 Computing distance from point to line, using right triangle.

```
        float vsq = mDirection.Dot(mDirection);
        float wsq = w.Dot(w);
        float proj = w.Dot(mDirection);

        return wsq - proj*proj/vsq;
    }
```

Note that in this case we're computing the squared distance. In most cases we'll be using this to avoid computing a square root. Another optimization is possible if we can guarantee that **v** is normalized; in that case we can avoid calculating and dividing by $\mathbf{v} \cdot \mathbf{v}$, since its value is 1.

11.2.3 Closest Point on Line Segment to Point

Source Code
Library
IvMath
Filename
IvLineSegment3

Recall that a line segment can be defined as the convex combination of two points P_0 and P_1, or

$$S(t) = (1 - t)P_0 + tP_1$$

where $0 \leq t \leq 1$. We can rewrite this as

$$S(t) = P_0 + t(P_1 - P_0)$$

or

$$S(t) = P + t\mathbf{v}$$

where t is similarly constrained. In this case **v** should not be normalized, as its length is the length of our line segment, and the endpoints are P and $P + \mathbf{v}$.

In the problem of finding the closest point on a line, we computed the projection of the point onto the line. Doing the same for a line segment gives us three cases (Figure 11.3). In the first case, the result of projecting Q_0 lies outside the segment but closest to P_0. In the second case, the result of projecting Q_1 lies outside the segment but closest to P_1. In the third case, the projected Q_2 lies on the segment, and we can use the same projection calculations that we used with a line.

To determine which case we're in, we begin by noting that

$$t = \frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}$$

is acting as our parameter t for the projected point, where again $\mathbf{w} = Q - P$. If $t < 0$, then the projected point lies beyond P_0, and the closest point is P_0. Similarly, if $t > 1$, then the closest point is P_1.

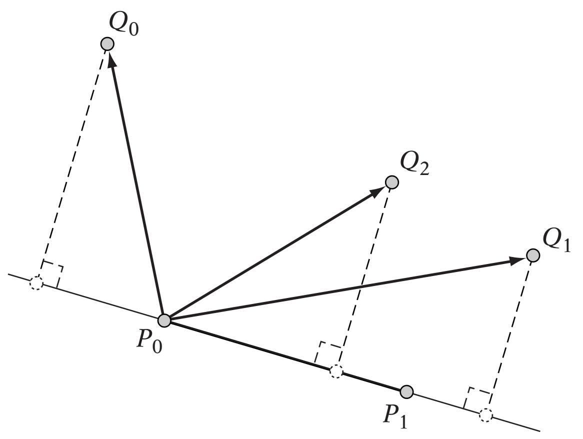


Figure 11.3 Three cases when projecting point onto line segment.

Testing t directly requires a floating-point division. By modifying our test we can defer the division to be performed only when we truly need it, that is, when the point lies on the segment. Since $\mathbf{v} \cdot \mathbf{v} > 0$, then $\mathbf{w} \cdot \mathbf{v} < 0$ in order for $t < 0$. And in order for $t > 1$, then $\mathbf{w} \cdot \mathbf{v} > \mathbf{v} \cdot \mathbf{v}$.

The equivalent code is

```
IvVector3 IvLineSegment3::ClosestPoint(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;

    float proj = w.Dot(mDirection);
    if ( proj <= 0 )
        return mOrigin;
    else
    {
        float vsq = mDirection.Dot(mDirection);
        if ( proj >= vsq )
            return mOrigin + mDirection;
        else
            return mOrigin + (proj/vsq)*mDirection;
    }
}
```

11.2.4 Line Segment-Point Distance

Source Code
Library
IvMath
Filename
IvLineSegment3

As with lines, we can compute the distance to the line segment by computing the distance to the closest point on the line segment. If we recall, there are three cases: the closest point is P_0, P_1, or a point somewhere else on the segment, which we'll calculate.

If the closest point is P_0, then we can compute the distance as $\|Q - P_0\|$. Since $\mathbf{w} = Q - P_0$, then the squared distance is equal to $\mathbf{w} \cdot \mathbf{w}$.

If the closest point is P_1, then the squared distance is $(Q - P_1) \cdot (Q - P_1)$. However, we're representing our endpoint as $P_1 = P_0 + \mathbf{v}$, so this becomes $(Q - P_0 - \mathbf{v}) \cdot (Q - P_0 - \mathbf{v})$. We can rewrite this as

$$\begin{aligned} \text{distsq}(Q, P_1) &= ((Q - P_0) - \mathbf{v}) \cdot ((Q - P_0) - \mathbf{v}) \\ &= (\mathbf{w} - \mathbf{v}) \cdot (\mathbf{w} - \mathbf{v}) \\ &= \mathbf{w} \cdot \mathbf{w} - 2\mathbf{w} \cdot \mathbf{v} + \mathbf{v} \cdot \mathbf{v} \end{aligned}$$

We've already calculated most of these dot products when determining whether we're closest to P_1, so all we need to compute is $\mathbf{w} \cdot \mathbf{w}$ and add.

If the closest point lies elsewhere on the segment, then we use the line distance calculation just given. The final code is

```
float IvLineSegment3::DistanceSquared(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;

    float proj = w.Dot(mDirection);
    if ( proj <= 0 )
    {
        return w.Dot(w);
    }
    else
    {
        float vsq = mDirection.Dot(mDirection);
        if ( proj >= vsq )
        {
            return w.Dot(w) - 2.0f*proj + vsq;
        }
        else
        {
            return w.Dot(w) - proj*proj/vsq;
        }
    }
}
```

11.2.5 Closest Points Between Two Lines

Source Code
Library
IvMath
Filename
IvLine3

Sunday [105] provides the following construction for finding the closest points between two lines. Note that in this case there are two closest points, one on each line, since there are two degrees of freedom. The situation is shown in Figure 11.4. Line L_1 is described by the point P_0 and the vector $\mathbf{u}$. Correspondingly, line L_2 is described by the point Q_0 and the vector $\mathbf{v}$, or

$$L_1(s) = P_0 + s\mathbf{u}$$
$$L_2(t) = Q_0 + t\mathbf{v}$$

Vectors $\mathbf{u}$ and $\mathbf{v}$ are not necessarily normalized.

We'll define the two closest points that we're looking for as lying at parameters s_c and t_c on the lines, and call them $L_1(s_c)$ and $L_2(t_c)$, respectively. We'll refer to the vector from $L_2(t_c)$ to $L_1(s_c)$ as $\mathbf{w}_c$.

Expanding $\mathbf{w}_c$, we have

$$\begin{aligned}\mathbf{w}_c &= L_1(s_c) - L_2(t_c)\\ &= P_0 + s_c\mathbf{u} - Q_0 - t_c\mathbf{v}\\ &= (P_0 - Q_0) + s_c\mathbf{u} - t_c\mathbf{v}\end{aligned}$$

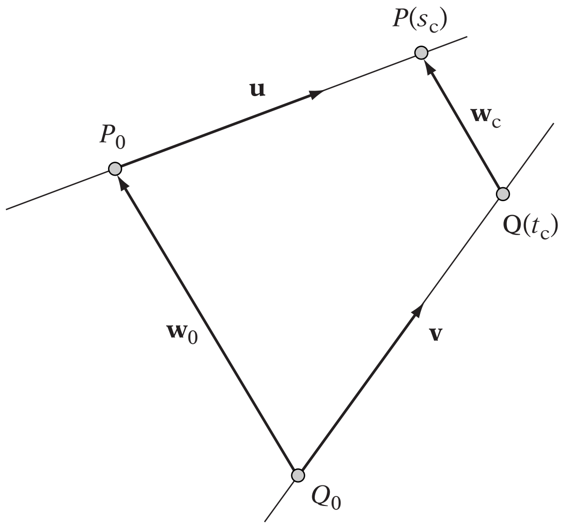


Figure 11.4 Finding closest points between two lines.

We'll use $\mathbf{w}_0$ to represent the difference vector $P_0 - Q_0$, so

$$\mathbf{w}_c = \mathbf{w}_0 + s_c\mathbf{u} - t_c\mathbf{v} \tag{11.1}$$

In order for $\mathbf{w}_c$ to represent the vector of closest distance, it needs to be perpendicular to both L_1 and L_2. This means that

$$\mathbf{w}_c \cdot \mathbf{u} = 0$$
$$\mathbf{w}_c \cdot \mathbf{v} = 0$$

Substituting in equation 11.1 and expanding, we get

$$0 = \mathbf{w}_0 \cdot \mathbf{u} + s_c\mathbf{u} \cdot \mathbf{u} - t_c\mathbf{u} \cdot \mathbf{v} \tag{11.2}$$
$$0 = \mathbf{w}_0 \cdot \mathbf{v} + s_c\mathbf{u} \cdot \mathbf{v} - t_c\mathbf{v} \cdot \mathbf{v} \tag{11.3}$$

We have two equations and two unknowns s_c and t_c, so we can solve for this system of equations. Doing so, we get the result that

$$s_c = \frac{be - cd}{ac - b^2} \tag{11.4}$$
$$t_c = \frac{ae - bd}{ac - b^2} \tag{11.5}$$

where

$$a = \mathbf{u} \cdot \mathbf{u}$$
$$b = \mathbf{u} \cdot \mathbf{v}$$
$$c = \mathbf{v} \cdot \mathbf{v}$$
$$d = \mathbf{u} \cdot \mathbf{w}_0$$
$$e = \mathbf{v} \cdot \mathbf{w}_0$$

There is one case where we need to be careful. If the two lines are parallel, then $\mathbf{u}$ and $\mathbf{v}$ are parallel, so $|\mathbf{u} \cdot \mathbf{v}| = \|\mathbf{u}\|\|\mathbf{v}\|$. Then the denominator $ac - b^2$ equals

$$\begin{aligned} ac - b^2 &= (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v}) - (\mathbf{u} \cdot \mathbf{v})^2 \\ &= \|\mathbf{u}\|^2\|\mathbf{v}\|^2 - (\|\mathbf{u}\|\|\mathbf{v}\|)^2 \\ &= 0 \end{aligned}$$

This leads to a division by 0. The problem is that there are an infinite number of pairs of closest points, spaced along each line. In this case we'll just find the closest point Q' on L_2 to the origin P_0 of line L_1, and return P_0 and Q'.

```
void ClosestPoints( IvVector3& point1,
                    IvVector3& point2,
                    const IvLine3& line1,
                    const IvLine3& line2 )
{
    IvVector3 w0 = line1.mOrigin - line2.mOrigin;
    float a = line1.mDirection.Dot( line1.mDirection );
    float b = line1.mDirection.Dot( line2.mDirection );
    float c = line2.mDirection.Dot( line2.mDirection );
    float d = line1.mDirection.Dot( w0 );
    float e = line2.mDirection.Dot( w0 );

    float denom = a*c - b*b;
    if ( ::IsZero(denom) )
    {
        point1 = mOrigin;
        point2 = other.mOrigin + (e/c)*other.mDirection;
    }
    else
    {
        point1 = mOrigin + ((b*e - c*d)/denom)*mDirection;
        point2 = other.mOrigin + ((a*e - b*d)/denom)*other.mDirection;
    }
}
```

11.2.6 Line-Line Distance

Source Code
Library
IvMath
Filename
IvLine3

From the calculation of closest points between two lines, we know that $\mathbf{w}_c$ is the vector of closest distance. Therefore, its length equals the distance between the two lines. Rather than compute the closest points directly, we can substitute the values of s_c and t_c into equation 11.1 and compute the length of $\mathbf{w}_c$. As before, to avoid the square root, we can use $\|\mathbf{w}_c\|^2 = \mathbf{w}_c \cdot \mathbf{w}_c$ instead.

The code is as follows:

```
float DistanceSquared( const IvLine3& line1, const IvLine3& line2  )
{
    // compute parameters
    IvVector3 w0 = line1.mOrigin - line2.mOrigin;
    float a = line1.mDirection.Dot( line1.mDirection );
    float b = line1.mDirection.Dot( line2.mDirection );
    float c = line2.mDirection.Dot( line2.mDirection );
    float d = line1.mDirection.Dot( w0 );
    float e = line2.mDirection.Dot( w0 );
```

```
        float denom = a*c - b*b;
        // if lines parallel
        if ( ::IsZero(denom) )
        {
            IvVector3 wc = w0 - (e/c)*line2.mDirection;
            return wc.Dot(wc);
        }
        // otherwise
        else
        {
              IvVector3 wc = w0 + ((b*e - c*d)/denom)*line1.mDirection
                                - ((a*e - b*d)/denom)*line2.mDirection;
              return wc.Dot(wc);
        }
    }
```

11.2.7 Closest Points Between Two Line Segments

Source Code
Library
IvMath
Filename
IvLineSegment3

Finding the closest points between two line segments follows from finding the closest points between two lines. We compute s_c and t_c, as we've done, but then need to clamp the results to the ranges of s and t defined by the endpoints of the two line segments. As before, we'll define our line segments as starting at the source point of the line, and ending at that source point plus the line vector. So for line L_1, the two points are P_0 and $P_0 + \mathbf{u}$ and for line L_2, the two points are Q_0 and $Q_0 + \mathbf{v}$. This gives us parameters 0 and 1 for the locations of the two endpoints. If our results s_c and t_c lie between the values 0 and 1, then our closest points lie on the two segments, and we're done.

Otherwise, we need to clamp our test to each of the endpoints and try again. To see how to do that, let's take a look at the $s = 0$ endpoint. Remember that what we want to do is find the smallest possible distance between the two points while not sliding off the end of the segment; namely, we want to minimize the length of $\mathbf{w}_c$ while maintaining $s = 0$. Since length is always increasing, we'll use $\|\mathbf{w}_c\|^2$, which will be much easier to minimize. Remember that

$$\mathbf{w}_c = \mathbf{w}_0 + s_c\mathbf{u} - t_c\mathbf{v}$$

Since we're clamping s_c to 0, this becomes

$$\mathbf{w}_c = \mathbf{w}_0 - t_c\mathbf{v}$$

And so for this endpoint we try to find the minimum value for

$$\mathbf{w}_c \cdot \mathbf{w}_c = (\mathbf{w}_0 - t_c\mathbf{v}) \cdot (\mathbf{w}_0 - t_c\mathbf{v}) \tag{11.6}$$

To do this, we return to calculus. To find a minimum value (in this case there is only one) for a function, we find a place where the derivative is 0. Taking the derivative of equation 11.6 in terms of t_c, we get the result

$$0 = -2\mathbf{v} \cdot (\mathbf{w}_0 - t_c\mathbf{v})$$

Solving for t_c:

$$t_c = \frac{\mathbf{v} \cdot \mathbf{w}_0}{\mathbf{v} \cdot \mathbf{v}} \tag{11.7}$$

So for the fixed point on line L_1 at $s = 0$, this gives us the parameter of the closest point on line L_2. As we can see, this is equivalent to computing the closest point between a line and a point, where the line is L_2 and the point is P_0.

For the $s = 1$ endpoint, we follow a similar process. Our minimization function is

$$\mathbf{w}_c \cdot \mathbf{w}_c = (\mathbf{w}_0 + \mathbf{u} - t_c\mathbf{v}) \cdot (\mathbf{w}_0 - t_c\mathbf{v}) \tag{11.8}$$

The corresponding zero derivative function is

$$0 = -2\mathbf{v} \cdot (\mathbf{w}_0 + \mathbf{u} - t_c\mathbf{v})$$

And solving for t_c gives us

$$t_c = \frac{\mathbf{v} \cdot \mathbf{w}_0 + \mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}$$

Again, this is equivalent to computing the closest point between a line and a point, where the line is L_2 and the point is $P_0 + \mathbf{v}$. The solutions for s_c when clamping to $t = 0$ or $t = 1$ are similar.

One nice thing about these functions is that they use the a through e values that we've already calculated for the basic line-line distance calculation. So equation 11.7 becomes

$$t_c = \frac{e}{c}$$

So which endpoints do we check? Well, if the parameter s_c is less than 0, then the closest segment point to line L_2 will be the $s = 0$ endpoint. And if s_c is greater than 1, then the closest segment point will be at $s = 1$. Choosing one or the other, we re-solve for t_c, and check that it lies between 0 and 1. If not, we perform the same process to clamp t_c to either the $t = 0$ or $t = 1$ endpoint, and recalculate s_c accordingly (with some minor adjustments to ensure that we keep s_c within 0 and 1).

Once again, there is a trick we can do to avoid multiple floating-point divisions. Instead of computing, say, s_c directly and testing against 0 and 1, we can compute the numerator s_N and denominator s_D. The initial s_D is always greater than zero, so we know that if s_N is less than zero, s_c is less than zero and we clamp to $s = 0$ accordingly. Similarly, if s_N is greater than s_D, we know that $s_c > 1$, and we clamp to $s = 1$. The same can be done for the t values. Using this, we can recalculate the numerator and denominator when necessary, and do the floating-point divides only after all the clamping has been done.

For example, the following code snippet calculates the s values:

```
// clamp s_c to 0
if (sN < 0.0f)
{
    sN = 0.0f;
    tN = e;
    tD = c;
}
// clamp s_c to 1
else if (sN > sD)
{
    sN = sD;
    tN = e + b;
    tD = c;
}
```

The full code is too long to contain here, but can be found on the demo CD.

11.2.8 Line Segment–Line Segment Distance

Source Code
Library
IvMath
Filename
IvLineSegment3

Finding the segment to segment squared distance is similar to line to line distance: we follow the procedure for closest points between line segments, calculate $\mathbf{w}_c$ directly from the final s_c and t_c, and then compute its length. The full code can be found on the CD in the `IvLineSegment3` friend function `DistanceSquared()`.

11.2.9 General Linear Components

Source Code
Library
IvMath
Filename
IvLine3
IvRay3
IvLineSegment3

Testing ray versus ray or line versus line segment is actually a simplification of the segment-segment closest point and distance determination. Instead of clamping against both components, we need only clamp against those endpoints that are necessary. So for example, if we treat $P_0 + s\mathbf{u}$ as the parameterization of a line segment, and $Q_0 + t\mathbf{v}$ as a line, then we need only to ensure that s_c is between 0 and 1, clamp to the appropriate endpoint, and adjust t_c accordingly. Similarly, if we're working with rays, we need only to clamp s_c or t_c to 0.

Implementations of these algorithms can be found in the appropriate classes.

11.3 Object Intersection

Now that we've covered some methods for measuring distance between primitives, we can talk about object intersection. The most direct, and naive, approach to determine whether two objects are intersecting is to work directly from raw object data. We could start with a triangle in object A and a triangle in object B and see if they are intersecting. Then we move to the next triangle in object A and test again. While ultimately this may work (the exception is if one object is inside the other), it will take a while to do and most of the time performing all those tests isn't even necessary. Take the two objects in Figure 11.5. They are clearly not intersecting—we can tell that in an instant. But our minds are not considering each object as a collection of lines and doing individual tests. Rather we are comparing them as a whole, as two rough blobs, and determining that the blobs aren't intersecting. By using a similar process in our intersection routines, we can save ourselves a lot of time.

For instance, suppose we surround each object with a sphere (Figure 11.6). We can begin by testing for intersection between the spheres.

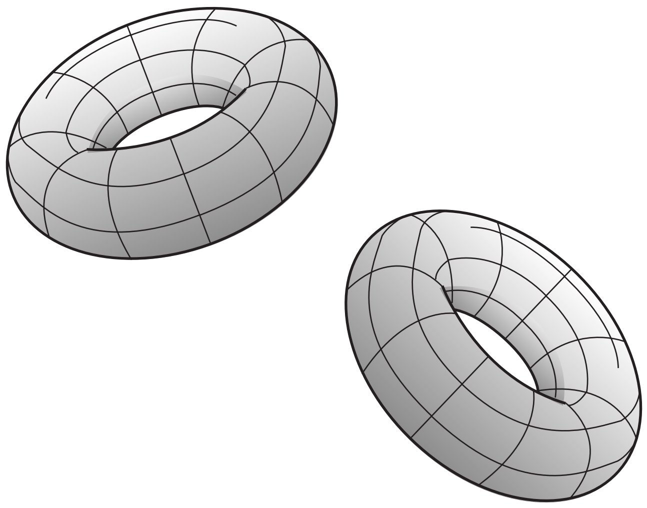


Figure 11.5 Non-intersecting objects.

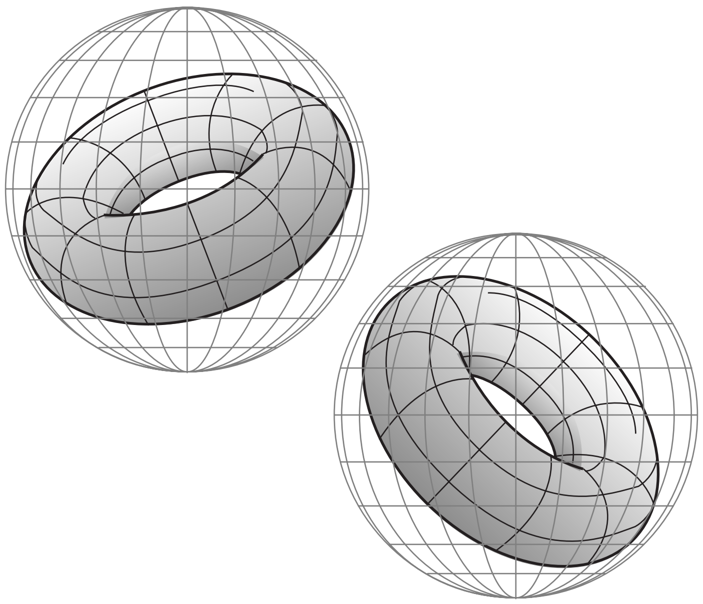


FIGURE 11.6 Non-intersecting objects with bounding sphere.

If the two spheres aren't intersecting, we know the objects aren't either. If the spheres are intersecting, we can try comparing another simplified version of our object — say, two boxes. The boxes fit the shape of our objects better but are still a simpler test than our full triangle-triangle comparison. If the boxes intersect, only then do we perform our complex collision detection routine.

This technique of using simplified objects to test intersections before performing more expensive operations is commonly used in game engines, and is necessary to get collision detection and other intersection-based systems running in real time. The simplified objects are known as *bounding objects*, and are named specifically after the basic primitive we used to approximate the object: bounding spheres and bounding boxes. In games, we can often get away with ignoring the underlying geometry completely and only using bounding objects to determine intersections. For example, when handling collisions in this way, either the action happens so fast that we don't notice any overlapping objects or objects reacting to collision when they appear separated, or the error is so slight that it doesn't matter. In any case, choosing the side of making the simulation run faster for a better play experience is usually a good decision.

To keep things concise, we will be focused primarily on detecting intersections between a few simple shapes. Other books are more detailed, covering many different polytopes (the 3D equivalent of polygons) and interactions between all sorts of bounding objects. In our case we'll focus on a few simple shapes, beginning with the simplest objects, and moving on to the most complex, or most expensive, to compute.

Within each section we'll only consider three cases of intersection. We'll first look at intersections between objects of the same bounding type, which is useful in collision detection. Second, we'll cover intersections between a ray and the particular bounding object, which we'll need for picking and visibility testing for AI. Finally, we'll discuss how to determine intersection between a plane and the bounding object, which can be used for both culling against frustum planes and collisions with essential planar objects like walls. In all cases we aren't concerned with the exact point of intersection, just whether we intersect.

11.3.1 Spheres

Definition

Source Code
Library
IvCollision
Filename
IvBounding
Sphere

The simplest possible bounding object is a sphere. It also has the most compact representation: a center point C and a radius r (Figure 11.7). When bounding a rigid object, a sphere is also independent of the object's orientation. This allows us to update a sphere quickly—when an object moves, we need only to update the sphere's position. If the object is scaled, we can scale the radius accordingly. The combination of low memory usage, fast update

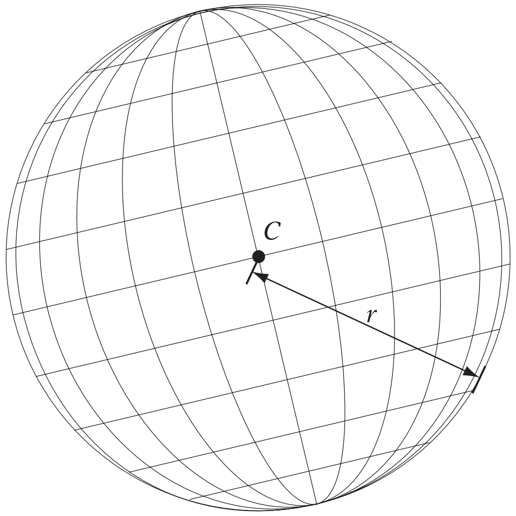


Figure 11.7 Bounding sphere.

time, and fast intersection tests makes bounding spheres a first choice in any real-time system.

The surface of the sphere is defined as all points P such that the length of the vector from C to P is equal to the radius

$$\sqrt{(P_x - C_x)^2 + (P_y - C_y)^2 + (P_z - C_z)^2} = r$$

or

$$\sqrt{(P - C) \cdot (P - C)} = r$$

Ideally, we'll want to choose the smallest possible sphere that encompasses the entire object. Too small a sphere, and we may skip two objects that are actually intersecting. Too large and we'll be unnecessarily performing our more expensive tests for objects that are clearly separate. Unfortunately, the most obvious methods for choosing a bounding sphere will not always generate as tight a fit as we might like.

One such method is to take the local origin of the object as our center C, and compute r by taking the maximum distance from that to all the vertices in the object. There are many problems with this. The most common is that the local origin could be considerably offset from the most desirable center point for the object (Figure 11.8a). This could happen if you have a character whose origin is at its feet, so it can be placed on the ground properly. An alternate but equivalent situation is where the origin is at a reasonable center point for

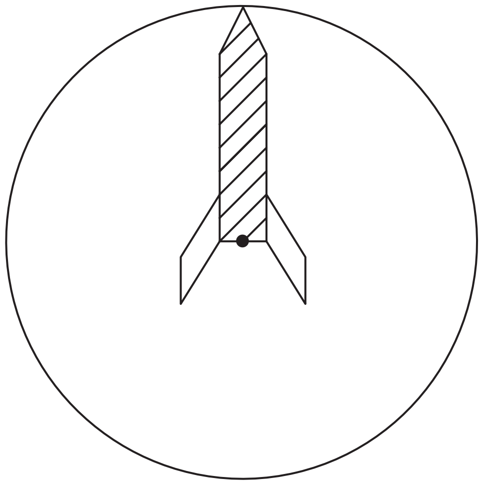


FIGURE 11.8a Bounding sphere, offset origin.

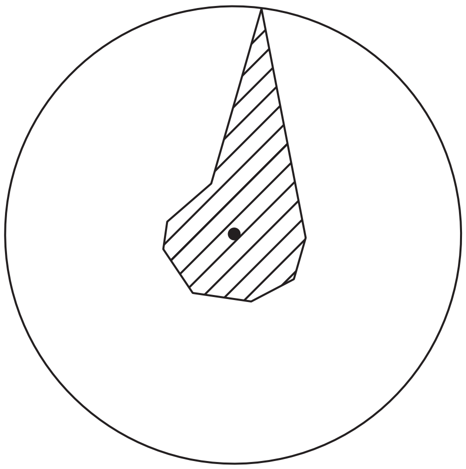


Figure 11.8b Bounding sphere, outlying point.

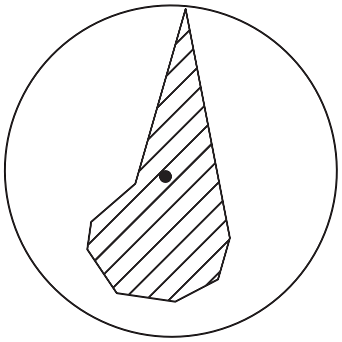


Figure 11.8c Bounding sphere, using centroid, object vertices.

the majority of the object's vertices, but there are one or two outlying vertices that cause problems (Figure 11.8b).

Eberly [27] provides a number of methods for finding a better fit. One is to average all the vertex locations to get the centroid and use that as our center. This works well for the case of the noncentered origin (Figure 11.8c), but still is a problem for the object with the outlying points. The reason is that the majority of the points lie within a small area and thus weight the centroid in that direction, pulling it away from the extrema.

We could also take an axis-aligned bounding box in the object's local space, and use its endpoints to compute our sphere position and radius

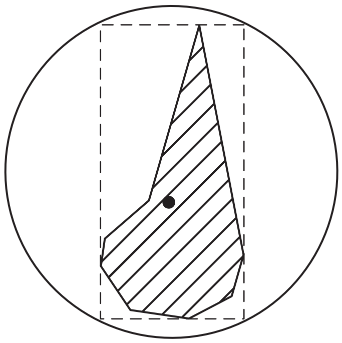


Figure 11.8d Bounding sphere, using box center, box vertices.

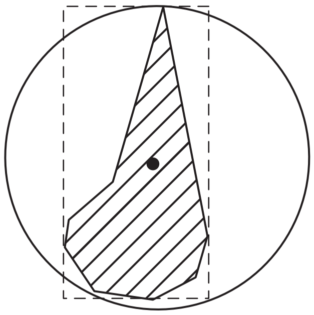


Figure 11.8e Bounding sphere, using box center, object vertices.

(Figure 11.8d). This tends to center the sphere better but leads to a looser fit. A compromise method uses the center of the bounding box as our sphere position, and computes the radius as the maximum distance from the center to our points. This gives a slightly better result (Figure 11.8e). The code for this last method is

```
void
IvBoundingSphere::Set( const IvPoint3* points, unsigned int numPoints )
{
  ASSERT( points );

  // compute minimal and maximal bounds
  IvVector3 min(points[0]), max(points[0]);
```

```
    for ( unsigned int i = 1; i < numPoints; ++i )
    {
      if (points[i].x < min.x)
        min.x = points[i].x;
      else if (points[i].x > max.x )
        max.x = points[i].x;
      if (points[i].y < min.y)
        min.y = points[i].y;
      else if (points[i].y > max.y )
        max.y = points[i].y;
      if (points[i].z < min.z)
        min.z = points[i].z;
      else if (points[i].z > max.z )
        max.z = points[i].z;
    }

    // compute center and radius
    mCenter = 0.5f*(min + max);
    float maxDistance = ::DistanceSquared( mCenter, points[0] );
    for ( unsigned int i = 1; i < numPoints; ++i )
    {
      float dist = ::DistanceSquared( mCenter, points[i] );
      if (dist > maxDistance)
        maxDistance = dist;
    }
    mRadius = ::IvSqrt( maxDistance );
  }
```

It should be noted that none of these methods is guaranteed to find the smallest bounding sphere. The standard algorithm for this is by Welzl [117], who showed that linear programming can be used to find the optimally smallest sphere surrounding a set of points. Two implementations are readily available online: one by Bernd Gaertner is provided under the GNU General Public License; another by Dave Eberly is at *www.magic-software.com*.

While we don't want to be cavalier about using ridiculously large bounding spheres, in some cases having the tightest possible fit isn't that much of an issue. Our objects will not be generally spherical, and so we'll be using something more complex for our final intersection test. As long as our spheres are reasonably close to a good fit, they will act to cull a great number of obvious cases, which is all we can ask for.

Sphere-Sphere Intersection

Determining whether two spheres are intersecting is as simple as their representation. We need only to determine whether the distance between their centers is less than the sum of their two radii (Figure 11.9), or

$$\sqrt{(C_1 - C_2) \cdot (C_1 - C_2)} <= r_1 + r_2 \tag{11.9}$$

The square root operation is expensive, and in any case it is unnecessary. Since we're not looking for the absolute difference, just a relation, we can use

$$(C_1 - C_2) \cdot (C_1 - C_2) <= (r_1 + r_2)^2 \tag{11.10}$$

As promised, this gives us an extremely cheap test for culling large numbers of intersections. This is why bounding spheres are used everywhere in computer graphics and simulation; we perform an initial fast check with a bounding sphere first before even considering the more complex cases.

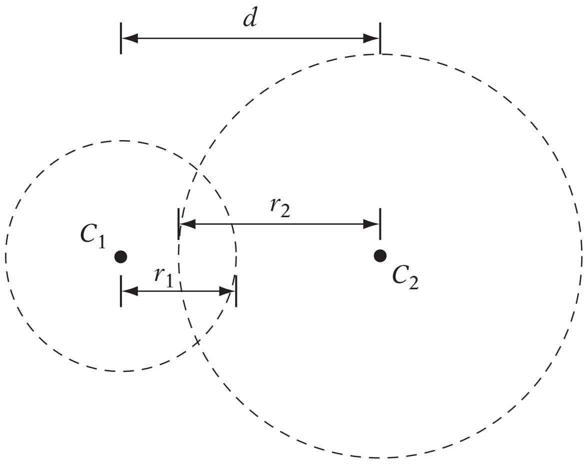


FIGURE 11.9 Sphere-sphere intersection.

The code is as follows:

```
bool
IvBoundingSphere::Intersect( const IvBoundingSphere& other )
{
    IvVector3 centerDiff = mCenter - other.mCenter;
    float radiusSum = mRadius + other.mRadius;
    return ( centerDiff.Dot(centerDiff) <= radiusSum*radiusSum );
}
```

Sphere-Ray Intersection

Intersection between a sphere and a ray is nearly as simple. Instead of testing two centers and comparing the distance with the sum of two radii, we test the distance between a single sphere center and a ray. If the distance is less than or equal to the sphere's radius, then the ray intersects the sphere (Figure 11.10).

We can use the line-point distance measurement described as the basis for this test. The code is as follows (it assumes an initial nonzero, nonnormalized **v**):

```
bool
IvBoundingSphere::Intersect( const IvRay3& ray )
{
    // compute intermediate values
    IvVector3 w = mCenter - ray.mOrigin;
    float wsq = w.Dot(w);
    float proj = w.Dot(ray.mDirection);
    float rsq = mRadius*mRadius;

    // if sphere behind ray, no intersection
    if ( proj < 0.0f && wsq > mRadius*mRadius )
       return false;

    float vsq = ray.mDirection.Dot(ray.mDirection);

    // test length of difference vs. radius
    return ( vsq*wsq - proj*proj <= vsq*mRadius*mRadius );
}
```

An additional check has been added since we're using a ray. If the sphere lies behind the origin of the ray, then there is no intersection. This is true if the angle between the difference vector **w** and the line direction is greater than 90 degrees (`proj < 0.0f`) and the line origin lies outside of the sphere (`wsq > mRadius*mRadius`).

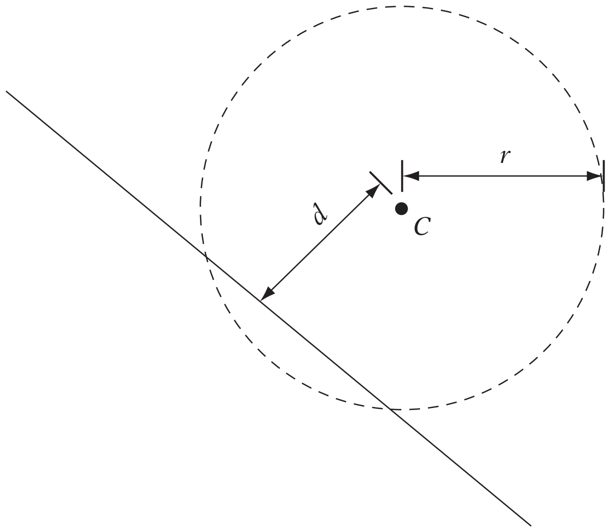


Figure 11.10 Line-sphere intersection.

We also remove the need for a floating-point divide by multiplying through by `vsq`. This adds 2 multiplications, but this should still be faster on most floating-point processors. As before, if we can guarantee that the ray direction vector is normalized, then we can remove the need for `vsq` altogether.

Sphere-Plane Intersection

Testing whether a sphere lies entirely on one side of a plane can be done quite efficiently. Recall that we can determine the distance between a point and such a plane by taking the absolute value of the result of the plane equation. If the result is positive and the distance is greater than the radius, then the sphere lies on the inside of the plane. If the result is negative, and the distance is greater than sphere's radius, then the sphere lies outside of the plane. Otherwise, the sphere intersects the plane.

The code for this test is

```
float
IvBoundingSphere::Classify( const IvPlane& plane )
{
    float distance = plane.test(mCenter):
    if ( distance > radius)
    {
        return distance-radius;
    }
    else if ( distance < -radius )
    {
        return distance+radius;
    }
    else
    {
        return 0.0f;
    }
}
```

Here we're returning a signed distance, like the standard plane test. If the sphere intersects, we return zero. Otherwise, we return the signed distance minus the signed distance of the radius.

11.3.2 Axis-Aligned Bounding Boxes

Source Code
Library
IvCollision
Filename
IvAABB

Definition

Spheres work well as either cheap culling objects or as bounding objects for a small class of models (i.e., if you're tossing grenades or writing a billards game). For more angular objects, we need a better fitting bounding surface. One possibility is the bounding box. Just like the bounding sphere, the ideal bounding box is the smallest possible box that encloses a model.

The first type we'll consider is the AABB, or axis-aligned bounding box, so-called because the box edges are aligned to the *world* axes. This makes representation of the box simple: we use two points, one each for the minimum and maximum xyz positions (Figure 11.11). When the object is translated, to update the box we translate the min and max points. Similarly, if the model is scaled, we scale the two points relative to the box center. However, because the box is aligned to the world axes, any rotation of the object means that we have to recalculate the min-max points from the model vertices' new positions in world space.

The other disadvantage AABBs have is that in many cases, like spheres, they still aren't a very close fit to the model they are trying to approximate (Figure 11.12). And for rounded objects like submarines or organic objects like humans, the fact that they have corners is a disadvantage as well. However, they are relatively cheap to compute and cheap to test as well, so they continue to prove useful.

One advantage that world axis-aligned boxes have over a box oriented to the model's local space is that we need only recompute them once per

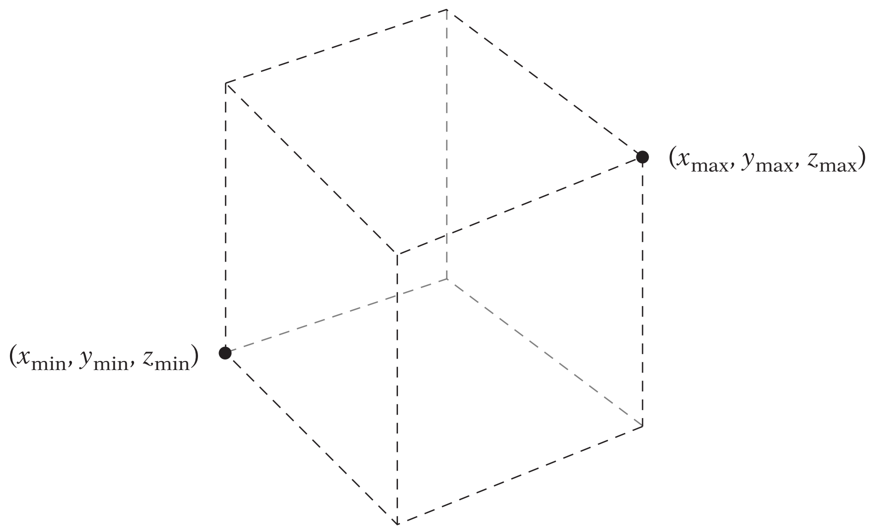

$(x_{\max}, y_{\max}, z_{\max})$
$(x_{\min}, y_{\min}, z_{\min})$

FIGURE 11.11 Axis-aligned bounding box.

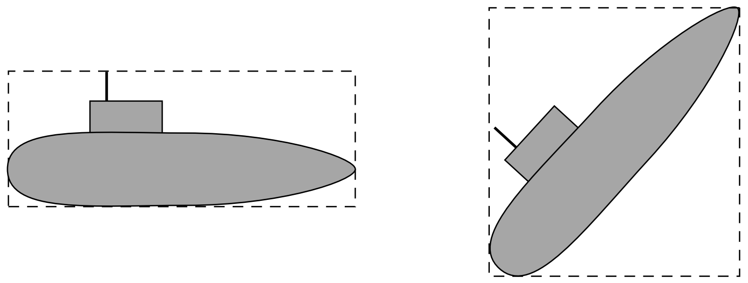


FIGURE 11.12 Fitting-axis-aligned bounding box.

frame, and then we can compare them directly without further transformation, since they are all in the same coordinate frame. So while AABBs have a high per-frame overhead (since we have to recalculate them each time an object reorients), they are *extremely* cheap to test against one another. As we'll see, there is a lot more overhead for determining intersection between oriented boxes. Oriented boxes are generally cheap per-frame (they move with the transforms of the object) but are more expensive to test against one another.

To compute an AABB, we first transform the model into world space. Then we set the minimum and maximum points to be equal to the first point (in world space, remember) in the model. Starting with the second point, we compare the xyz values of each point with those in the minimum and maximum.

If any coordinate is less than that in the minimum, set the minimum coordinate to that value. And the same for the maximum, except use greater than. When done, this will give you the axis-aligned extrema for your box.

```
void
IvAABB::Set( const IvPoint3* points, unsigned int numPoints )
{
    ASSERT( points );

    // compute minimal and maximal bounds
    mMinima.Set(points[0]);
    mMaxima.Set(points[0]);
    for ( unsigned int i = 1; i < numPoints; ++i )
    {
        if (points[i].x < mMinima.x)
            mMinima.x = points[i].x;
        else if (points[i].x > mMaxima.x )
            mMaxima.x = points[i].x;
        if (points[i].y < mMinima.y)
            mMinima.y = points[i].y;
        else if (points[i].y > mMaxima.y )
            mMaxima.y = points[i].y;
        if (points[i].z < mMinima.z)
            mMinima.z = points[i].z;
        else if (points[i].z > mMaxima.z )
            mMaxima.z = points[i].z;
    }
}
```

AABB-AABB Intersection

In order to understand how we find intersections between two axis-aligned boxes, we introduce the notion of a *separating plane*. The general idea is this: we check the boxes in each of the coordinate directions in world space. If we can find a plane that separates the two boxes in any of the coordinate directions, then the two boxes are not intersecting. If we fail all three separating plane tests, then they are intersecting and we handle it appropriately.

Let's look at the process of finding a separating plane between two boxes in the x-direction. Since the boxes are axis-aligned, this becomes a one-dimensional problem on a number line. The min and max values of the two boxes become the extrema of two intervals on the line. If the two intervals are separate, then there is a separating plane and the two boxes are separate along the x-direction. This is the case only if the maximum value of one interval is less than the minimum value of the other interval (Figure 11.13). Expressing this for all three axes:

```
bool
IvAABB::Intersect( const IvAABB& other )
{
  // if separated in x direction
  if (mMinima.x > other.mMaxima.x || other.mMinima.x > mMaxima.x )
    return false;

  // if separated in y direction
  if (mMinima.y > other.mMaxima.y || other.mMinima.y > mMaxima.y )
    return false;

  // if separated in z direction
  if (mMinima.z > other.mMaxima.z || other.mMinima.z > mMaxima.z )
    return false;

  // no separation, must be intersecting
  return true;
}
```

Examining this code makes another advantage of AABBs clear. If we're using 3D objects in an essentially 2D game, we can ignore the z-axis and so save a step in our computations. This is not always possible with boxes aligned to the local axes of an object.

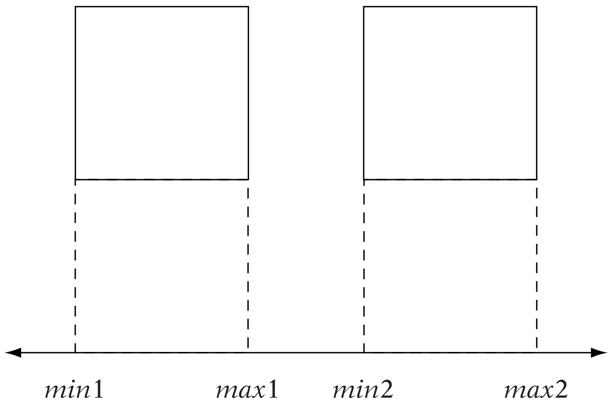


FIGURE 11.13 Axis-aligned box-box separation test.

AABB-Ray Intersection

Determining intersection between a ray and an axis-aligned box is similar to determining intersection between two boxes. We check one axis direction at a time as before, except that in this case there is a little more interaction between steps.

Figure 11.14 shows a 2D cross section of the situation. The ray R shown intersects the minimum and maximum x planes of the box at $R(s_x)$ and $R(t_x)$, respectively, and the minimum and maximum y planes at $R(s_y)$ and $R(t_y)$. Instead of testing for extrema overlaps in the box axes directions, we'll test whether there is overlap between the line segment from $R(s_x)$ to $R(t_x)$, and the line segment from $R(s_y)$ to $R(t_y)$. This is the same as testing whether the intervals of the line parameters $[s_x, t_x]$ and $[s_y, t_y]$ overlap.

If the ray misses the box, as in the figure, then the $[s_x, t_x]$ interval doesn't overlap the $[s_y, t_y]$ interval, just like the preceding box-box intersection. So if there's no overlap (if $t_x < s_y$, or vice versa), then there's no intersection, and we stop. If they do overlap, then we test that overlap interval against the z intersections. If there's overlap there as well, then we know that the ray intersects the box.

For each axis, we begin by computing the parameters where the ray (represented by the point P and vector $\mathbf{v}$) crosses the min and max planes. So for example, in the x direction we'll calculate intersections with the

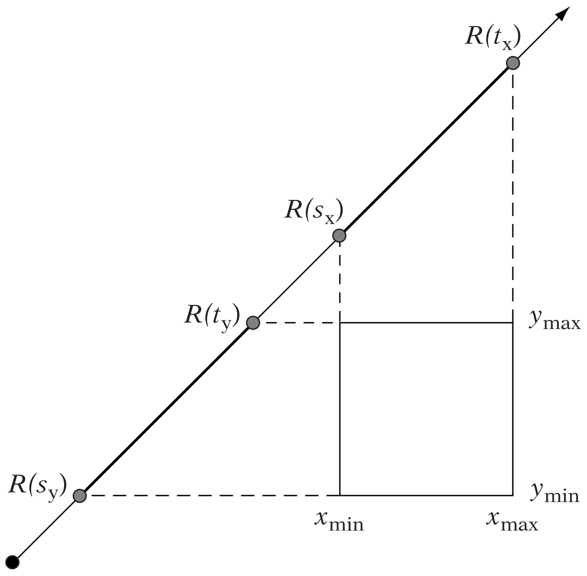


Figure 11.14 Axis-aligned box-ray separation test.

$x = x_{min}$ and $x = x_{max}$ planes. To do this, we need to solve the following equations:

$$P_x + s_x v_x = x_{min}$$
$$P_x + t_x v_x = x_{max}$$

Solving for s_x and t_x, we get

$$s_x = \frac{x_{min} - P_x}{v_x}$$
$$t_x = \frac{x_{max} - P_x}{v_x}$$

There's one special case we need to handle: clearly if v_x is zero, then there are no solutions for s_x and t_x; the ray is parallel to the minimum and maximum planes. In this case we need to test whether P_x lies between x_{min} and x_{max}. If not, the ray misses the box and there is no intersection.

We'll track our parameter overlap interval by using two values s_{max} and t_{min}, initialized to the maximum interval $[-\infty, \infty]$. These represent the maximum s and minimum t values seen so far. After we calculate intersection parameters for each axis, we'll sort them so that $s < t$, and then update s_{max} and t_{min} if $s > s_{max}$ or $t < t_{min}$. We know that the ray misses the box if we ever find that $s_{max} > t_{min}$. For example, looking at Figure 11.14, after doing the x-axis calculations we see that $s_{max} = s_x$ and $t_{min} = t_x$. After the y-axis parameters are computed, t_{min} is updated to t_y, and s_{max} remains s_x. But $s_x > t_y$, so there is no intersection.

Since we're using a ray, there is one further check: if any t value is ever less than zero, we know that both parameters are less than zero, and that the box is behind the ray and there is no intersection. The code, abbreviated for space, is as follows:

```
bool
IvAABB::Intersect( const IvRay3& ray )
{
    float maxS = -FLT_MAX;
    float minT = FLT_MAX;

    // do x coordinate test (yz planes)

    // ray is parallel to plane
    if ( ::IsZero( ray.mDirection.x ) )
    {
       // ray passes by box
```

```
            if ( ray.mOrigin.x < mMin.x  || ray.mOrigin.x > mMax.x )
                return false;
        }
        else
        {
            // compute intersection parameters and sort
            float s = (mMin.x - ray.mOrigin.x)/ray.mDirection.x;
            float t = (mMax.x - ray.mOrigin.x)/ray.mDirection.x;
            if ( s > t )
            {
               float temp = s;
               s = t;
               t = temp;
            }

            // adjust min and max values
            if ( s > maxS )
                maxS = s;
            if ( t < minT )
                minT = t;
            // check for intersection failure
            if ( minT < 0.0f || maxS > minT )
                return false;
        }

        // do y and z coordinate tests (xz & xy planes)
        ...

        // done, have intersection
        return true;
    }
```

AABB-Plane Intersection

The most naive test to determine whether a box intersects a plane is to see whether a single box edge crosses the plane. That is, if two neighboring vertices lie on either side of the plane, there is an intersection. There are 12 edges, so this requires 24 plane tests. There are two improvements we can make to this. The first is to note that we need test only *opposing* corners of the box, that is, two vertices that lie at either end of a diagonal that passes through the box center. This cuts the number of "edges" to be checked down to 4. The second improvement is provided by Möller and Haines [79], who note that we really need to test only one: the diagonal most closely aligned with the plane normal. Figure 11.15 shows a cross section of the situation.

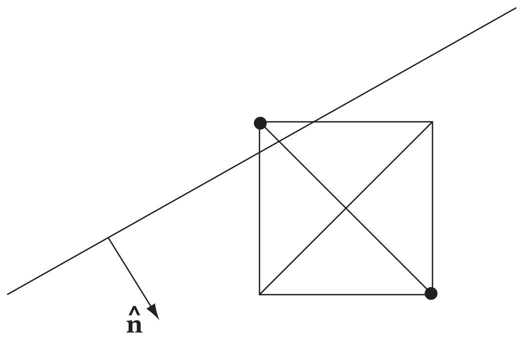


FIGURE 11.15 Axis-aligned box-plane separation test.

Code to manage this is as follows. As before, we return zero if there is an intersection, the signed distance otherwise.

```
float
IvAABB::Classify( const IvPlane& plane )
{
    IvVector3 diagMin, diagMax;
    // set min/max values for x direction
    if ( plane.mNormal.x >= 0)
    {
        diagMin.x = mMin.x;
        diagMax.x = mMax.x;
    }
    else
    {
        diagMin.x = mMin.x;
        diagMax.x = mMax.x;
    }

    // ditto for y and z directions
    ...
    // minimum on positive side of plane, box on positive side
    float test = plane.mNormal.Dot( diagMin ) + plane.mD;
    if ( test > 0.0f )
        return test;

    test = plane.mNormal.Dot ( diagMax ) + plane.mD;
    // min on non-positive side, max on non-negative side, intersection
    if ( test >= 0.0f )
        return 0.0f;
    // max on negative side, box on negative side
    else
        return test;
}
```

11.3.3 Swept Spheres

Definition

Source Code
Library
IvCollision
Filename
IvCapsule

The bounding sphere and the axis-aligned bounding box have one problem: there is no real sense of orientation. The sphere is symmetric across all axes and the AABB is always aligned to the world axes. For objects that have definite long and short axes (a human, for example), this doesn't provide for an ideal approximation. The next two bounding objects we'll consider are not tied to the world axes at all, which makes them much more suitable for general models.

The simplest of such bounding regions are the swept spheres. If we consider the sphere as a region enclosed by a radius around a point, or a zero-dimensional center, the swept spheres use higher dimensional centers. One example is the *capsule*, which is a line segment surrounded by a radius (Figure 11.16a). Another possibility is the lozenge, which has a quadrilateral center (Figure 11.16b). For our purposes, we'll concentrate on capsules (Eberly [27] provides more information on lozenges and other swept spheres).

Computing the capsule in local space for a set of points is fairly straightforward, but not as simple as spheres or bounding boxes. We are first going to assume that our model is generally axis-aligned in local space. This is not unreasonable considering that the artists usually build models in this way. For models that are not axis-aligned, see Eberly [27] or Van Verth [110].

Our first step is to find the long axis for the model. We do this by computing the bounding box, and finding the longest side. The line that we will use for our base line segment runs through the middle of the box. We'll use the center of one end of the box as our line point A, and the box axis $\mathbf{w}$ as our line vector. We could use the local origin and a coordinate axis for our line, but while

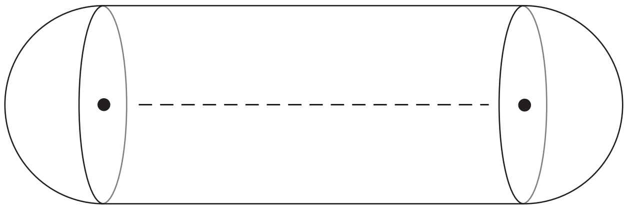


Figure 11.16a Capsule.

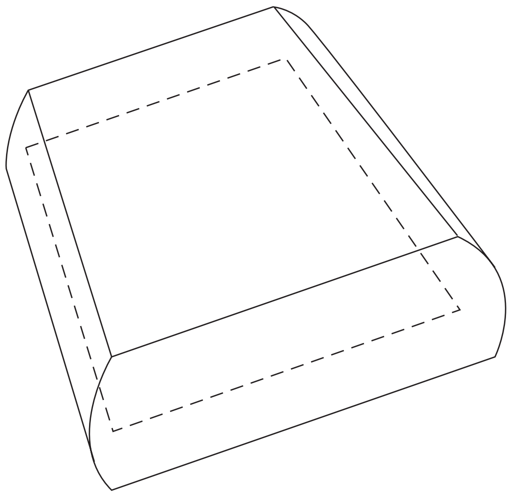


Figure 11.16b Lozenge.

we're willing to assume axis alignment, we're not so optimistic as to assume that the model is centered on a coordinate axis.

Now we need to compute the radius r of the capsule. For each point in the model, we compute the distance from the point to the line. The maximum distance becomes our radius. The line combined with the radius gives us a tube with radius r and ends extending to infinity. All the points in the model just fit inside the tube.

The final part to building the capsule is capping the tube with two hemispheres that just contain any points near the end of the model. Eberly [27] describes a method for doing this. The center of each hemisphere is one of the two endpoints of the line segment, so finding the hemisphere allows us to define the line segment. Let's consider the endpoint with the smaller t value—call it $L(\xi_0)$—shown in Figure 11.17. We want to slide the endcap in from the right until we find the smallest ξ_0 such that all points in the model either lie on the hemisphere (such as point P_0) or to the right of it (point P_1). Another way to think of this is that for each point we'll compute the hemisphere centered

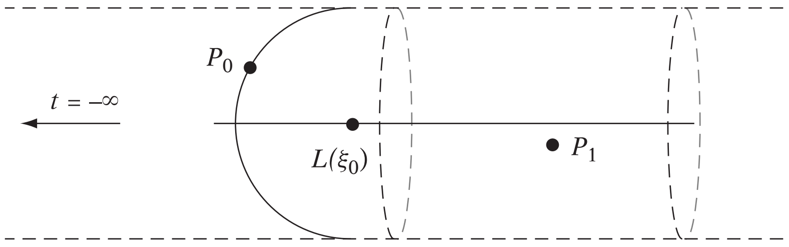


Figure 11.17 Capsule endcap fitting.

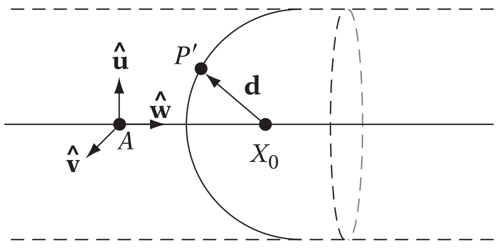


FIGURE 11.18 Determining hemisphere center X_0 for given point P'.

on the line that exactly contains it, and choose the one with the smallest ξ_0 value. If we do the same at the other end, with hemispheres oriented the other way and choosing the one with largest parameter value ξ_1, then all points will be tightly enclosed by the capsule.

To set this up, we first need to transform our points from the local space of the model to the local space of the line. We'll build a coordinate frame consisting of the line point A, normalized line vector $\hat{\mathbf{w}}$, and two vectors perpendicular to $\hat{\mathbf{w}}$: $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$. Subtracting the line point from the model point, and multiplying by a 3×3 matrix formed from $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$, transforms the model space point P to a local line space P' with line space coordinates (u, v, w). Since $\hat{\mathbf{w}}$ is normalized, a point $L(\xi_0)$ on the line equals $(0, 0, \xi_0)$ in line space.

If P' lies on a hemisphere with radius r and center X_0 on the line, the length of a vector $\mathbf{d}$ from X_0 to P_i' should be equal to the radius r (Figure 11.18). Given this and the other parameters, we should be able to solve for X_0, and hence ξ_0.

The vector $\mathbf{d} = P' - X_0$. In line space $\mathbf{d} = (u, v, w) - (0, 0, \xi) = (u, v, w - \xi)$. Ensuring that $\|\mathbf{d}\| = r$ means that

$$u^2 + v^2 + (w - \xi_0)^2 = r^2$$

Solving for ξ_0, we get

$$\xi_0 = w - \left(\pm\sqrt{r^2 - (u^2 + v^2)}\right)$$

Since this is a hemisphere, we want X_0 to be to the right of P, so $w \geq \xi_0$ and this becomes

$$\xi_0 = w + \sqrt{r^2 - (u^2 + v^2)}$$

Computing this for every point P in our model and finding the minimum ξ_0 gives us our first endpoint. Similarly, the second endpoint is found by finding the maximum value of

$$\xi_1 = w - \sqrt{r^2 - (u^2 + v^2)}$$

Capsule-Capsule Intersection

Handling capsule-capsule intersection is very similar to sphere-sphere intersection. Instead of calculating the distance between two points, and determining whether that is less than the sum of the two radii, we calculate the distance between two line segments and check against the radii. As before, if the distance is less than the sum of the two radii, we have intersecting capsules.

```
bool
IvCapsule::Intersect( const IvCapsule& other )
{
    float radiusSum = mRadius + other.mRadius;
    return ( mSegment.DistanceSquared( other.mSegment )
                                        <= radiusSum*radiusSum );
}
```

Capsule-Ray Intersection

Capsule-ray intersection follows from capsule-capsule collision. Instead of finding the distance between two line segments, we need to find the distance between a ray and a line segment, and compare to the radius of the capsule:

```
bool
IvCapsule::Intersect( const IvRay3& ray )
{
    // test distance between line and segment vs. radius
    return ( ray.DistanceSquared( mSegment ) <= mRadius*mRadius );
}
```

Capsule-Plane Intersection

There are two tests necessary to determine whether a capsule intersects a plane. First of all, if the two endpoints of the line segment defining the capsule lie on either side of the plane, then clearly the capsule intersects the plane. However, even if the line segment lies on one side of the plane, the distance between one of the endpoints and the plane may be less than the radius. In this case the capsule and plane would also intersect. Both cases are easy to test; we already have the pieces in place.

The code is

```
float
IvCapsule::Classify( const IvPlane& plane )
```

```
{
    float s0 = plane.Test( mSegment.GetEndpoint0() );
    float s1 = plane.Test( mSegment.GetEndpoint1() );

    // points on opposite sides or intersecting plane
    if (s0*s1 <= 0.0f)
        return 0.0f;

    // intersect if either endpoint is within radius distance of plane
    if( ::IvAbs(s0) <= mRadius || ::IvAbs(s1) <= mRadius )
        return 0.0f;

    // return signed distance
    return ( ::IvAbs(s0) < ::IvAbs(s1) ? s0 : s1 );
}
```

11.3.4 Object-Oriented Boxes

Source Code
Library
IvSimulation
Filename
IvOBB

World axis-aligned boxes are easy to create and fast to use for detecting intersections, but are not a very tight fit around models that are not themselves generally aligned to the world axes (Figure 11.19). A more accurate approach is to create an initial bounding box that is a tight fit around the model in local space, and then rotate and translate the box as well as the model. These are known as object-oriented bounding boxes, or OBBs. This has another advantage in that we don't have to recalculate the box every time the model moves, just transform the initial one. Also, for rigid models with a large number of vertices, recomputing the AABB every frame may be too expensive. The disadvantage is that testing intersections between two object-oriented boxes is more complicated. In the axis-aligned case, we could simplify our cases down

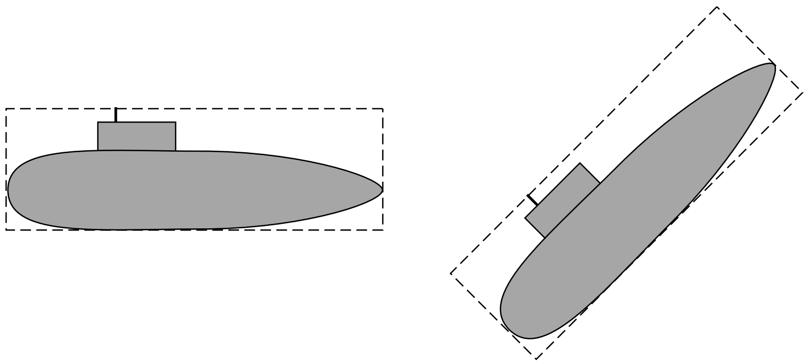


Figure 11.19 Oriented bounding boxes.

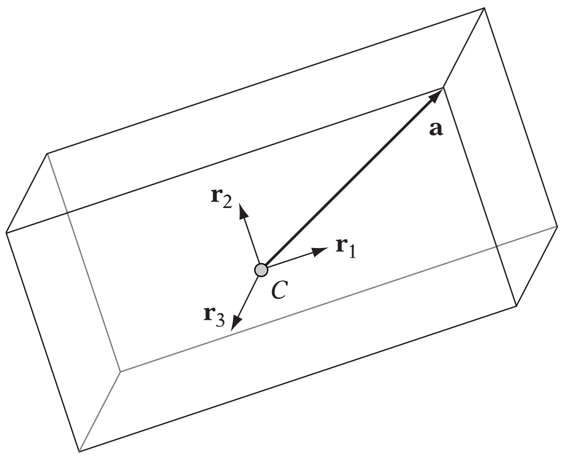


FIGURE 11.20 Properties of OBBs.

to three tests because of the alignment. In the OBB case, the two can be at any relative orientation to each other, which complicates the issue considerably.

The representation for an OBB A consists of the center point C_a, an orientation matrix $\mathbf{R}_a$, and an extent vector $\mathbf{a}$ (Figure 11.20). The extent vector represents the difference from the center point to the point of maximum x, y, and z on the box. Note that the center of the box is not necessarily the same as the local origin of the model, nor does the orientation of the box have to match the orientation of the model. If either is the case, some adjusting of the model's local-to-world transformation will have to be done to generate the box axes and center location in world space. If the box to model space orientation transformation is $\mathbf{R}_{box \to model}$ and the model's orientation is $\mathbf{R}_{model \to world}$, then the box's local to world rotation is

$$\mathbf{R}_{box \to world} = \mathbf{R}_{model \to world} \cdot \mathbf{R}_{box \to model}$$

To simplify our life, however, we can use boxes aligned to the model's local coordinates, with a vector $\mathbf{d}$ in model space indicating the box center relative to the model center (as mentioned in Chapter 3, it's not usually practical to build models with their bounding box center as their local origin). In either case, any time we need the box center $\mathbf{c}$ in world space we can use

$$\mathbf{c} = \mathbf{R}_{model \to world}\mathbf{d} + \mathbf{t}$$

OBB-OBB Intersection

There have been many methods for testing intersections between two arbitrarily oriented boxes, including linear programming techniques and closest-feature tracking. The most efficient technique known to date, however, uses the concept of separating axes and is due to Gottschalk, Lin, and Manocha [48]. The following discussion is heavily drawn from this paper, with some additional concepts due to Eberly [27] and van den Bergen [109].

Recall that to test whether two axis-aligned boxes were intersecting, we did three tests, one for each axis x, y, and z. For each test, we checked the extents of each box along each of the axis directions. This is equivalent to projecting the box along the basis vectors **i**, **j**, and **k**. If the intervals of a given projection don't overlap, then there is a separating plane normal to the test vector and therefore no intersection. The corresponding axis is known as a separating axis.

This works well for axis-aligned boxes, but we need a slightly different test for oriented boxes. To simplify our equations and improve performance, we'll use transformations relative to box A. We end up with a single translation vector $\mathbf{c}$ from A to B, where $\mathbf{c} = \mathbf{R}_a^T \cdot (C_b - C_a)$, and a relative rotation matrix $\mathbf{R} = \mathbf{R}_b^T \mathbf{R}_a$. A's extent vector remains the same, since it's relative to its local space. B's extent vector becomes $\mathbf{R}^T \mathbf{b}$.

Now suppose we have a potential separating axis direction $\mathbf{v}$. We want to perform the same test we did with the AABBs: project each box onto the vector and check to see whether the projections are separate or not. Another way of representing this is to project the box centers onto the vector as endpoints, and then project the extent vectors closest to the center onto the vector as well (Figure 11.21). If the distance between the projected box centers is less than the sum of the lengths of the projected extents, then there is no intersection. Expressed mathematically, there is no intersection if

$$|\mathbf{c} \cdot \mathbf{v}| > |\mathbf{a} \cdot \mathbf{v} + (\mathbf{R}^T \mathbf{b}) \cdot \mathbf{v}|$$

This works if the extent vectors are aligned appropriately to give us the maximum projected length, but we can't make that assumption. Instead we'll use a pseudo-dot product that forces maximum length, so the equivalent to $\mathbf{a} \cdot \mathbf{v}$ is

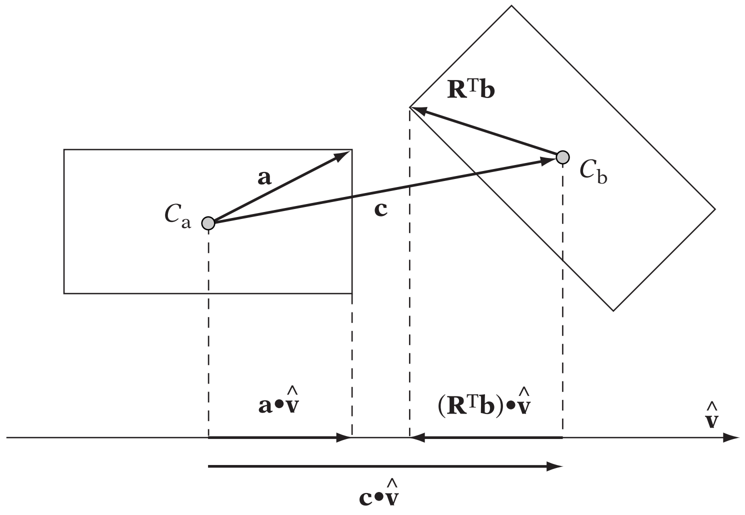
$$|a_x v_x| + |a_y v_y| + |a_z v_z|$$

This is legal because the extents can be taken from any of the 8 octants, so we can get any sign we want for any term.

An equivalent equation can be found for $(\mathbf{R}^T \mathbf{b}) \cdot \mathbf{v}$. The final separating axis equation is

$$|\mathbf{c} \cdot \mathbf{v}| > \sum_i |a_i v_i| + \sum_i |(\mathbf{R}^T \mathbf{b})_i v_i| \tag{11.11}$$

While this gives us our test, there is an infinite number of choices for $\mathbf{v}$, which is not practical. Gottschalk [48] demonstrates that any separating



$\mathbf{R}^T\mathbf{b}$
C_b
$\mathbf{a}$
C_a
$\mathbf{c}$
$\mathbf{a}\bullet\hat{\mathbf{v}}$
$(\mathbf{R}^T\mathbf{b})\bullet\hat{\mathbf{v}}$
$\hat{\mathbf{v}}$
$\mathbf{c}\bullet\hat{\mathbf{v}}$

FIGURE 11.21 Example of OBB separation test.

plane will either be parallel to one of the box faces or parallel to an edge from each box. This means that a maximum of 15 separating axis tests are necessary: 3 against the axes of box A, 3 against the axes of box B, and 9 cross products using one axis from A and one from B.

The nice thing about this result is that it allows us to simplify our equations considerably. For example, let's use the cross product of the local x axis from A and local y axis from B. In A's local space, the x-axis of A is $\mathbf{i} = (1, 0, 0)$. If we represent the matrix $\mathbf{R}$ as the three column vectors $(\mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2)$, then the y-axis of B in A's space is (r_{01}, r_{11}, r_{21}). Performing the cross product $\mathbf{i} \times \mathbf{r}_1$, we get

$$\mathbf{v} = (0, -r_{21}, r_{11}) \tag{11.12}$$

Converting this to terms relative to B's basis via the transpose of $\mathbf{R}$:

$$\mathbf{R}^T(\mathbf{i} \times \mathbf{r}_1) = \begin{bmatrix} \mathbf{r}_0 \cdot (\mathbf{i} \times \mathbf{r}_1) \\ \mathbf{r}_1 \cdot (\mathbf{i} \times \mathbf{r}_1) \\ \mathbf{r}_2 \cdot (\mathbf{i} \times \mathbf{r}_1) \end{bmatrix} = \begin{bmatrix} \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_0) \\ \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_1) \\ \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_2) \end{bmatrix} = \begin{bmatrix} \mathbf{i} \cdot (-\mathbf{r}_2) \\ \mathbf{i} \cdot \mathbf{0} \\ \mathbf{i} \cdot \mathbf{r}_0 \end{bmatrix}$$

So $\mathbf{v}$ in B space is

$$\mathbf{R}^T\mathbf{v} = (-r_{01}, 0, r_{00}) \tag{11.13}$$

Substituting equations 11.12 and 11.13 into equation 11.11 and multiplying out the terms, the final axis test is

$$|c_2 r_{11} - c_1 r_{21}| > a_1|r_{32}| + a_2|r_{11}| + b_0|r_{02}| + b_2|r_{00}|$$

The test for other axes can be derived similarly. All use the absolute value of elements from the matrix **R** so it is far more efficient to precompute them and then perform the axis tests. If this is done, the algorithm takes about 200 operations. It can be found in `IvOBB::Intersect()`.

One caveat: any implementation of this algorithm needs to take steps to avoid numerical problems with floating-point precision. In particular, if two edges, one from each box, are nearly parallel, the resulting cross product will be near-zero. This will lead to invalid results for the separation test. The solution is to detect the condition, and only test against the six main axes of the boxes.

OBB-Ray Intersection

Detecting intersection between a linear component and an oriented box is much simpler than detecting intersection between two boxes. One method is to transform the ray into the box's local space and perform a standard AABB intersection test. To transform the linear component, the origin point is transformed by the inverse of the box's world transform matrix, and the direction vector by the inverse rotation of the box's transformation matrix. The newly transformed line, ray, or line segment can be passed into the appropriate AABB routine.

An alternative is to use a modified version of the AABB algorithm, as described by Möller and Haines [79]. In this case, instead of using planes normal to the three world axes, we'll use planes normal to the three box axes. Recall that these axes are specified as the three column vectors in our rotation matrix.

Each axis has two parallel planes associated with it. If we treat the box's center as the origin of our frame, the extent vector **a** contains the magnitude of our d values for these planes. For example, two of the parallel box planes are $r_{00}x + r_{10}y + r_{20}z + a_x = 0$ and $r_{00}x + r_{10}y + r_{20}z - a_x = 0$.

If we translate our ray so that its origin is relative to the box origin, we can determine s and t parameters for the intersections with these planes, just as we did with the axis-aligned box. In this case, the formulas for s and t for each axis (including the translation) are

$$s = \frac{\mathbf{r}_i \cdot (C - P) - \mathbf{a}_i}{\mathbf{R}_i \cdot \mathbf{v}} \qquad t = \frac{\mathbf{r}_i \cdot (C - P) + \mathbf{a}_i}{\mathbf{R}_i \cdot \mathbf{v}}$$

We also need to modify our test to determine whether the ray is parallel to the current pair of planes we're testing—this is easily done by taking the dot product of the direction vector **v** and the plane normal, and seeing if it is close to zero. If so, the ray is parallel to the plane, and we need to project the vector $C - P$ onto the current axis, and see if the result lies outside the extents.

The modified code is

```
bool
IvOBB::Intersect( const IvRay3& ray )
{
    float maxS = -FLT_MAX;
    float minT = FLT_MAX;

    // compute difference vector
    IvVector3 diff = mCenter - ray.mOrigin;

    // for each axis do
    for (int i = 0; i < 3; ++i)
    {
        // get axis i
        IvVector3 axis = mRotation.GetColumn( i );
        // project relative vector onto axis
        float e = axis.Dot( diff );
        float f = ray.mDirection.Dot( axis );

        // ray is parallel to plane
        if ( ::IsZero( f ) )
        {
            // ray passes by box
            if ( -e - mA[i] > 0 || -e + mA[i] > 0 )
               return false;
            continue;
        }

        float s = (e - mA[i])/f;
        float t = (e + mA[i])/f;

        // fix order
        ...
        // adjust min and max values
        ...
        // check for intersection failure
        ...
    }

    // done, have intersection
    return true;
}
```

Performance can be improved here by storing the rotation matrix as an array of three vectors instead of an `IvMatrix33`.

OBB-Plane Intersection

As we did with with OBB-ray intersection, we can classify the intersection between an OBB and a plane by transforming the plane to the OBB's frame and using the AABB-plane classification algorithm. Since the transformation is just a pure rotation and a translation, we can find the transformed normal by

$$\hat{\mathbf{n}}' = \mathbf{R}^T \hat{\mathbf{n}}$$

We apply the transpose since we're going from world space into box space. The minimal and maximal points for the AABB in this case are the extent vector and its negative, **a** and −**a**, respectively.

An alternative, presented by Möller and Haines [79], is to use the principle of separating planes again. This time, our test vector will be the plane normal, and we'll project the box diagonal on to it. To ensure we get maximum extent, we'll add the absolute values of the elements together, similar to what we did before:

$$r = |(a_0\mathbf{r}_0) \cdot \mathbf{n}| + |(a_1\mathbf{r}_1) \cdot \mathbf{n}| + |(a_2\mathbf{r}_2) \cdot \mathbf{n}|$$

Here each $\mathbf{r}_i$ represents a column of the rotation matrix. The box intersects the plane if the distance between the box center and the plane is less than r. The resulting code is

```
float IvOBB::Classify( const IvPlane& plane )
{
    IvVector3 xNormal = ::Transpose(mRotation)*plane.mNormal;
    float r = mExtents.x*::IvAbs(xNormal.x) + mExtents.y*::IvAbs(xNormal.y)
        + mEextents.z*::IvAbs(xNormal.z);

    float d = plane. Test(mCenter);
    if (::IvAbs(d) < r)
       return 0.0f;
    else if (d < 0.0f)
       return d + r;
    else
       return d - r;
}
```

11.3.5 Triangles

Source Code
Library
IvMath
Filename
IvTriangle

All of the bounding objects we've discussed up until now have been approximations to our model (assuming our model is more complex than, say, a box or a sphere). To test actual intersections between models, we need to get right down to the basic building block of our geometry: the triangle. As before, we will be representing our triangle as the convex combination of three points.

Triangle-Triangle Intersection

A naive approach to determining triangle-triangle intersection uses the triangle-ray intersection test that follows. If one of the line segments composing an edge of one triangle intersects the other triangle, then the two triangles are intersecting. While this works, there are faster methods. One such is presented by Martin Held in his ERIT system [62]. The general algorithm has four major steps.

Figure 11.22 shows the situation. Taking the first triangle T, composed of points P_0, P_1, and P_2, we compute its plane equation. Recall that the plane equation for a normal $\mathbf{n} = (a, b, c)$ and a point on the plane $P_0 = (x_0, y_0, z_0)$ is

$$0 = ax + by + cz - (ax_0 + by_0 + cz_0)$$

or

$$0 = ax + by + cz + d$$

In this case the plane normal is computed from $(P_1 - P_0) \times (P_2 - P_0)$ and normalized, and the plane point is P_0.

Now we take our second triangle, composed of points Q_0, Q_1, and Q_2. We plug each point into T's plane equation and test whether all three lie on the same side of the plane. This is true if all three results have the same sign. If they do, there is no intersection and we quit. Otherwise we store the results d_0, d_1, and d_2 generated from the plane equation for each point and continue.

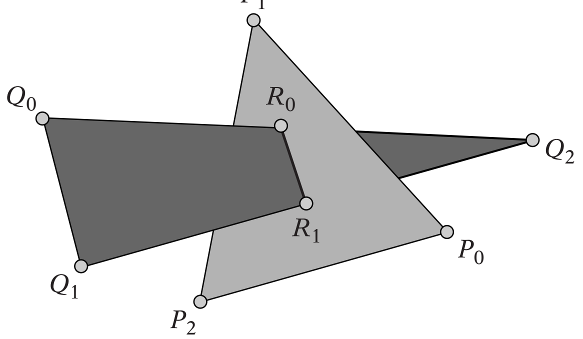


FIGURE 11.22 Triangle intersection.

Using the d_is, we determine which edges of the second triangle cross the plane. If a given d_i,d_j pair have opposite signs, then the corresponding points are on opposite sides of the plane. In Figure 11.22, those pairs are Q_0, Q_2 and Q_1, Q_2. We can compute the intersections of the corresponding triangle edge with the plane, using the formula

$$R = Q_i + \frac{d_i}{d_i - d_j}(Q_j - Q_i)$$

Doing this for each pair will produce two endpoints (R_0, R_1) of a line segment L lying on T's plane.

The final step is determining whether the line segment is outside the boundary of T. We'll simplify our 3D problem to a 2D one by projecting the triangle T and line segment L to one of the xy, xz, or yz planes to create T' and L'. To improve our accuracy, we'll choose the one which provides the maximum area for the projection of T. If we look at the normal $\mathbf{n}$ for T, one of the coordinate values (x, y, z) will have the maximum absolute value, that is, the normal is pointing generally along that axis. If we drop that coordinate and keep the other two, this will give us the maximum projected area.

To test whether the projected line segment is inside T', we compute the line equation $Ax + By + C$ for each pair of projected edge points. We can use this like the plane equation to test to which side of a line a point lies. If both endpoints R_0' and R_1' lie on the inner side of each line, then the line segment lies inside the triangle, and we have an intersection.

There is one other case: the line segment may be crossing an edge of the triangle. We can test for this by computing the intersection of the line segment with the line formed from each edge. If the intersection lies on the edge, then the line segment crosses the triangle, and we have an intersection.

Triangle-Ray Intersection

There are two possible approaches to determining triangle-ray intersection. The first is to use the plane equation for the triangle (computed from the three vertices) and determine the intersection point of the ray with the plane (if any). We can then use a point-in-triangle test to determine whether the intersection lies within the triangle.

While a relatively simple approach, it has some disadvantages. First of all, we need to either store the plane equation or, if we're short on space, compute it every time we wish to do the intersection test. Second, it's a two-pass algorithm: compute the plane intersection, and then test whether it's in the triangle. Fortunately, we have an alternative. The following approach, presented by Möller and Haines [79], uses affine combinations to compute the ray-triangle intersection.

We define our triangle as having vertices V_0, V_1, and V_2. We can define two edge vectors $\mathbf{u}$ and $\mathbf{v}$ (Figure 11.23), where

$$\begin{aligned}\mathbf{u} &= V_1 - V_0\\ \mathbf{v} &= V_2 - V_0\end{aligned}$$

Recall that the point V_0 with the vectors $\mathbf{u}$ and $\mathbf{v}$ can be used to create an affine combination that spans the plane of the triangle, with barycentric coordinates (u, v). So the formula for a point $T(u, v)$ on the plane is

$$\begin{aligned}T(u, v) &= V_0 + u\mathbf{u} + v\mathbf{v}\\ &= V_0 + u(V_1 - V_0) + v(V_2 - V_0)\end{aligned}$$

Rearranging terms, we get

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

We want the contribution of each point to be nonnegative, so for a point inside the triangle

$$\begin{aligned}u &\geq 0\\ v &\geq 0\\ u + v &\leq 1\end{aligned}$$

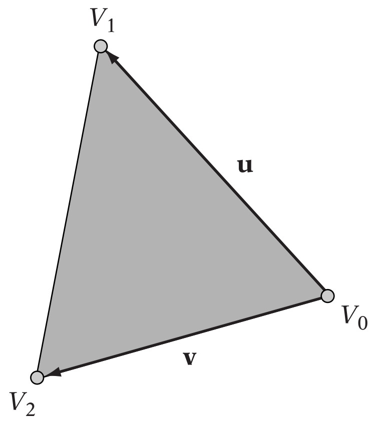


FIGURE 11.23 Affine space of triangle.

If u or $v < 0$, then the point is on the outside of one of the two axis edges. If $u+v > 1$, the point is outside the third edge. So if we can compute the barycentric coordinates for the intersection point $T(u, v)$, we can easily determine whether the point is outside the triangle.

To compute the u, v coordinates of the intersection point, the result of the line equation $L = P + t\mathbf{d}$ will equal a solution to the affine combination $T(u, v)$ (Figure 11.24). So

$$P + t\mathbf{d} = (1 - u - v)V_0 + uV_1 + vV_2$$

We can express this as a matrix product

$$\begin{bmatrix} -\mathbf{d} \; V_1 - V_0 \; V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = P - V_0$$

Using Cramer's rule, or row-reduction, we can solve this matrix equation for (t, u, v). The final result is

$$\begin{aligned} t &= \frac{\mathbf{q} \cdot \mathbf{e}_2}{\mathbf{p} \cdot \mathbf{e}_1} \\ u &= \frac{\mathbf{p} \cdot \mathbf{s}}{\mathbf{p} \cdot \mathbf{e}_1} \\ v &= \frac{\mathbf{q} \cdot \mathbf{d}}{\mathbf{p} \cdot \mathbf{e}_1} \end{aligned}$$

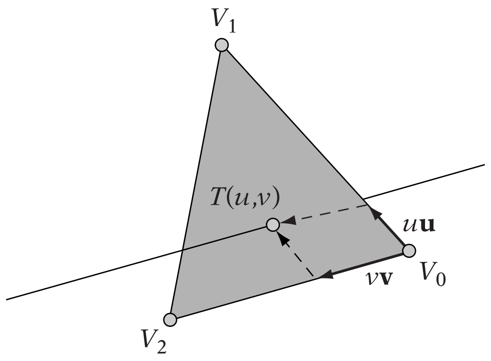


FIGURE 11.24 Barycentric coordinates of line intersection.

where

$$\begin{aligned}
\mathbf{e}_1 &= V_1 - V_0 \\
\mathbf{e}_2 &= V_2 - V_0 \\
\mathbf{s} &= P - V_0 \\
\mathbf{p} &= \mathbf{d} \times \mathbf{e}_2 \\
\mathbf{q} &= \mathbf{s} \times \mathbf{e}_1
\end{aligned}$$

The final algorithm includes checks for division by zero and intersections that lie outside the triangle:

```
bool
TriangleIntersect( const IvVector3& v0, const IvVector3& v1,
                     const IvVector3& v2, const IvRay& ray )
{
  // test ray direction against triangle
  IvVector3 e1 = v1 - v0;
  IvVector3 e2 = v2 - v0;
  IvVector3 p =  ray.mDirection.Cross(e2);
  float a = e1.Dot(p)

  // if result zero, no intersection or infinite intersections
  // (ray parallel to triangle plane)
  if ( ::IsZero(a) )
    return false;

  // compute denominator
  float f = 1.0f/a;

  // compute barycentric coordinates
  IvVector3 s = ray.mOrigin - v0;
  u = f*s.Dot(p)
  if (u < 0.0f || u > 1.0f) return false;

  IvVector3 q = s.Cross(e1);
  v = f*ray.mDirection.Dot(q);
  if (v < 0.0f || u+v > 1.0f) return false;

  // compute line parameter
  t = f*e2.Dot(q);
```

```
    return (t >= 0);
}
```

Parameters u, v, and t can be returned if the barycentric coordinates on the triangle or the parameter for the exact point of intersection are needed.

Triangle-Plane Intersection

We covered triangle-plane intersection when we discussed triangle-triangle intersection. We take our triangle, composed of points P_0, P_1, and P_2, and plug each point into the plane equation. If all three lie on the same side of the plane, then there is no intersection. Otherwise, there is, and if we desire we can find the particular line segment of intersection, as described earlier. If there is no intersection, the signed distance is the plane equation result of minimum magnitude.

11.4 A Simple Collision System

Now that we have some methods for testing intersection between various primitive types, we can make use of them in a practical system. The example we'll consider is collision detection. Rather than building a fully general collision system, we'll do only as much as we need to for a basic game — in our case, we'll use a submarine game as our example. This is to keep things as simple as possible and to illustrate various points to consider when building your own system. It's also good to keep in mind that a particular subsystem of a game, whether it is collision or rendering, needs only to be as accurate as the game calls for. Building a truly flexible collision system that handles all possible situations may be overkill and eat up processing time that could be used to do work elsewhere.

11.4.1 Choosing a Base Primitive

The first step in building the system is to choose the base bounding shape for our models. We'll see in the following sections how we can use a hierarchy of bounding primitives to get a better fit to the model's surface, but for now we'll consider only one per model. Which primitive we choose depends highly on the expected topology we're trying to approximate with it. For example, if we're writing a pool game, using bounding spheres for our balls makes perfect sense. However, for a human character bounding spheres are not a good choice because one axis of the model is far longer than the other two—not a good fit. In particular, getting characters through an interior space might be a tricky proposition unless all your doorways and hallways are at least six feet wide.

Considering that our model is made of triangles, using them should give us the most accurate results. However, while they are cheap as a one-on-one test, it would be costly to test every possible triangle-triangle combination between two objects. This becomes more feasible when we have some sort of culling hierarchy to whittle down the possible triangle pairs to a few contenders—we'll discuss that in more detail shortly. However, if we can get a good fit with a simpler bounding volume, we can get a reasonably accurate measure of collision by doing a volume-volume test without having to do the full triangle-triangle test.

Since AABBs change size depending on the model's orientation, they are not usually a good choice for a base bounding primitive. They are more often used as a culling test, such as in the sweep-and-prune system described in Section 11.4.4.

Among the primitives we've discussed, this leaves us with capsules and OBBs. Which we choose depends on our performance requirements and how angular our models are. If we have mostly boxy models—like tanks—capsules or even lozenges won't provide very compelling collisions. An OBB is a better shape to choose for this situation. For our case, however, submarines and torpedoes are both generally sausage-shaped. If we had to go with a single bounding object which approximates a submarine, capsules are an excellent choice.

11.4.2 Bounding Hierarchies

Source Code
Demo
Hierarchy

Unless our objects are almost exactly the shape of the bounding primitive (such as our pool ball example), then there are still going to be places where our test indicates intersection where there is visibly no collision. For example, the conning tower of our submarine makes the bounding capsule encompass a large area of empty space at the top of the hull. Suppose a torpedo is heading towards our submarine and through that area. Instead of harmlessly passing over the hull as we would expect from the visual evidence, it will explode because we have detected a collision with the inaccurately large bounding region.

The solution is to use a set of bounding primitives to get a better approximation to the surface of the model. In our submarine example, we could use one capsule for the main hull and one for the conning tower. If we are willing to allow a slightly forgiving system, we could ignore the conning tower for the purposes of collision and get a very nice fit with the hull capsule. Or we could go the more detailed route and add one for the conning tower, as well

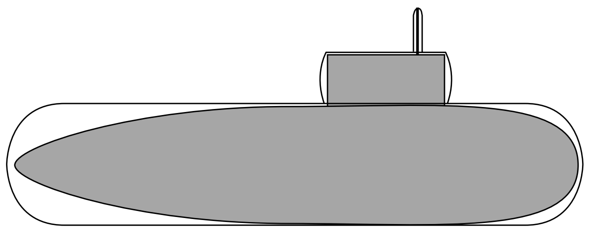


Figure 11.25 Using multiple bounding objects.

as a third for the periscope (Figure 11.25). To check for intersection, we test each bounding primitive for the first model against all the primitives in the second, much as we would have done for the triangles.

To speed this up we can keep our original bounding capsule and use it as a rough test before checking further. Better still, we can generate bounding spheres for each model and test against those instead. It's a very cheap test and can do a great job of culling large numbers of cases. We could also generate bounding spheres for each of our smaller capsules, and use these spheres in preliminary culling steps before checking individual capsule pairs.

This gives us a bounding hierarchy for our model (Figure 11.26). We compare the top level bounding spheres first. Only if they are intersecting do we then move on to the lower level of sphere check and capsule check. This can cull out a large number of cases and make it much more likely that we'll be testing only the two lower-level capsules that are actually intersecting.

Bounding hierarchies work very well with scene graphs, and it's fairly simple to add this functionality to our existing classes. We begin by adding an `IvBoundingSphere` member to each `IvSpatial` object. In addition, our `IvGeometry` leaf nodes will have two `IvCapsule` members: one for local space and one that we'll transform into world space. This gives us our culling sphere hierarchy, with capsules as the lowest-level test. Now we need to pregenerate the bounding parameters before initiating the collision test process. This is done as a part of the recursive propagation of transform information. We propagate the transform information down from the root. When we reach a leaf node, we generate a new world space sphere and capsule from the updated transform data. Then as we undo the recursion, we propagate the changes in the bounding spheres back up. At each `IvNode` level, we merge its children's bounding spheres to obtain the sphere for the node. Note that if an update is called on a node other than the root, undoing the recursion is not sufficient. We must contain propagating the new bound upward, all the way to the root.

The procedure to merge two spheres is as follows. If one sphere completely surrounds the other, then the larger sphere is clearly the minimum enclosing sphere. However, in most cases the two spheres are interpenetrating or separate. The situation can be seen in Figure 11.27. We have two spheres, one

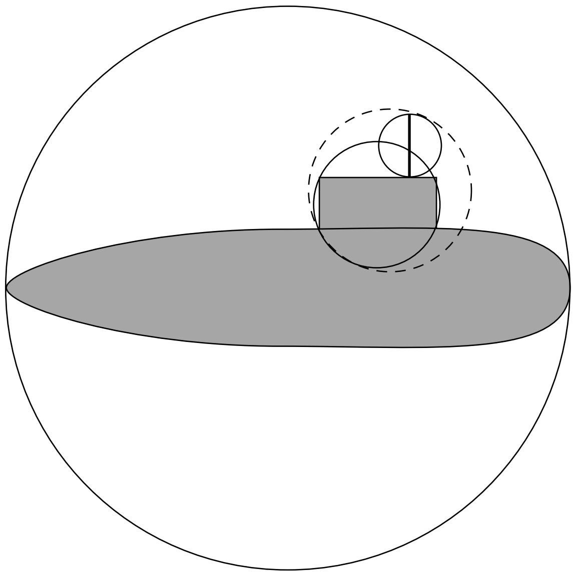


Figure 11.26 Using bounding hierarchy.

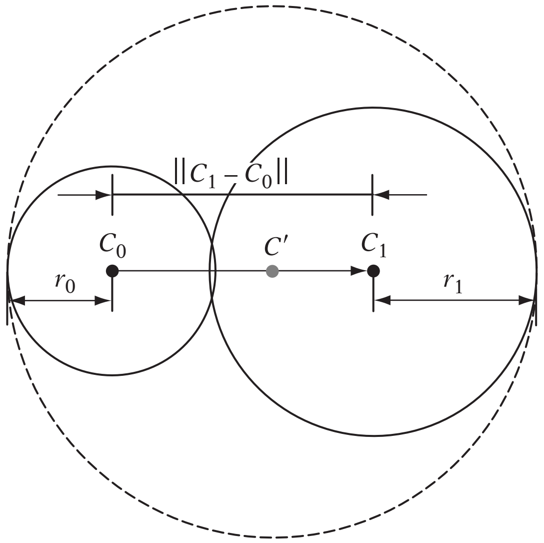


Figure 11.27 Merging two spheres.

with center C_0 and radius r_0, and the other with center C_1 and radius r_1. The diameter of the new sphere will be $r_0 + \|C_1 - C_0\| + r_1$, so the radius r will be $1/2(r_0+r_1)+1/2\|C_1-C_0\|$. The new center will lie along the line $C_0+t(C_1-C_0)$. We determine t by moving r_0 in distance back along the line to the edge of the sphere, and then r units forward to the new center. The resulting code is

Source Code
Library
IvCollision
Filename
IvBoundingSphere

```
IvBoundingSphere Merge( const IvBoundingSphere& s0,
                        const IvBoundingSphere& s1 )
{
    IvVector3 diff = s1.mCenter - s0.mCenter;
    float distsq = diff.Dot(diff);
    float radiusdiff = s1.mRadius - s0.mRadius;
    // one sphere inside other
    if ( distsq <= radiusdiff*radiusdiff )
            if ( s0.mRadius > s1.mRadius )
                return s0;
            else
                return s1;

    // build new sphere
    float dist = ::IvSqrt( distsq );
    float newRadius = 0.5f*( s0.mRadius + s1.mRadius + dist );
    IvVector3 newCenter = s0.mCenter;
    if (!::IsZero( dist ))
        newCenter += ((newRadius-s0.mRadius)/dist)*diff;
    return IvBoundingSphere( newCenter, newRadius );
}
```

Source Code
Library
IvScene
Filename
IvGeometry
IvNode

Finding collisions between the two hierarchies is another recursive process. We'll define a virtual method in `IvSpatial` called `Colliding`, which checks for collision between the current object and another `IvSpatial` object. Represented in pseudocode, this is

```
Boolean IvGeometry::Colliding(IvSpatial* other)
{
   if other is not IvGeometry node
       return other->Colliding( this )
   else
       if both spheres and capsules intersecting return TRUE
}
```

For IvNodes, we use the following:

```
Boolean IvNode::Colliding(IvSpatial* other)
{
   if bounding spheres are not colliding, return FALSE
   else
       if this node has children
           for each child do
               if child->Colliding(other)
                   return TRUE
           return FALSE
       else if other node has children
           for each child in other node do
               if other_child->Colliding(this)
                   return TRUE
           return FALSE
       else
           return FALSE  // shouldn't happen
}
```

This will find the first collision between the hierarchies. You may wish to find them all (there may be more than one if our models are not convex). If so, instead of returning TRUE immediately when a collision is found, store the collision information and proceed to the next child.

We can take this technique of using bounding hierarchies further. For example, if we want to do triangle-triangle intersection testing, we can build a hierarchy to perform coarser but cheaper intersection tests. If two objects are intersecting, we can traverse the two hierarchies until we get to the two intersecting triangles (there may be more than two if the objects are concave). Obviously, we'll want to create much larger hierarchies in this case. Generating them so that they are as efficient as possible — they both cull well and have a reasonably small tree size — is not a simple task. Gottschalk *et al*. [48] provide some information for building OBB-trees, while Ericson [34] covers the general cases.

Spheres, capsules, AABBs, and OBBs have all been used as primitives for culling bounding hierarchies. Most tests have been done for hierarchies with triangles as leaf nodes. Gottschalk [48] demonstrates that OBBs work better than both AABBs and spheres if our models have static geometry. However, if we're constantly deforming our vertices — for example, with skinned character models — recomputing the OBBs in the hierarchy is an expensive step. Using spheres or AABBs can be a better choice in this circumstance.

11.4.3 Dynamic Objects

So far we have been using intersection tests assuming that our objects don't move between frames. This is clearly not so. In games, objects are constantly moving, and we need to be careful when we use static tests to catch collisions between moving objects.

For example, in one frame we have two objects moving towards each other, clearly heading for a collision somewhere in the center of the screen (Figure 11.28a). Ideally, in the next frame we want to catch a snapshot of them just as they collide, or are slightly intersecting. However, if we take too large a simulation step, they may have passed partially through each other (Figure 11.28b). Using a frame-by-frame static test we will miss the initial collision. Worse yet, if we take a larger step, the two objects will have passed right through each other, and we'll miss the collision entirely.

One way to catch this is to sweep our bounding primitives along a path and then test intersection between the swept primitives that we've generated. A simple example of this is testing intersection between two moving spheres. If we sweep a sphere along a line segment, we get — no surprise — a capsule. Based on the two objects' velocities, we can generate capsules for each object and test for intersection. If one is found, then we know the two objects may collide somewhere between frames and we can investigate further.

We generally have to worry about this problem only when the relative velocities of objects are large enough or the frame times are long enough that

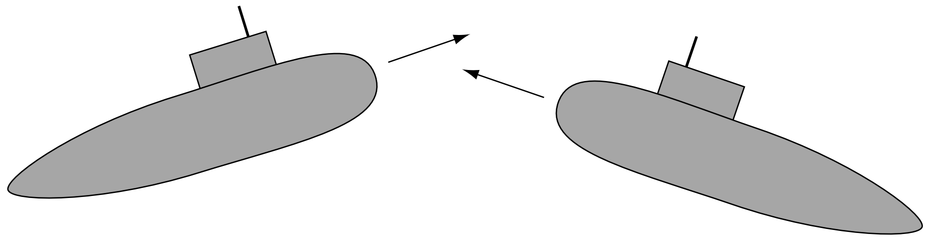


Figure 11.28a Potential collision.

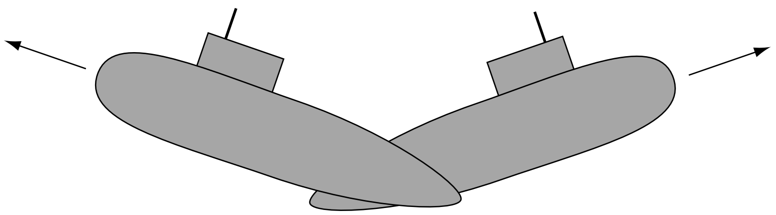


Figure 11.28b Partially missed collision.

one object can move, relative to another, farther than half its thickness in the direction of travel. For example, a tank with a speed of 30 km/hr moves about 0.12 m/frame, assuming 60 frames/sec. If the tank is 10 m long, its movement is miniscule compared to its total length and we can probably get away with static testing. Suppose, however, that we fire a 1 m long missile at that tank, traveling at 120 km/hr. We also have a bug in our rendering code which causes us to drop to 10 frames/sec, giving us a travel distance of 3-1/3 m. The missile's path crosses through the tank at an angle and is already through it by the next frame. This may seem like an extreme example, but in collision systems it's often best to plan for the extreme case.

Walls, since they are infinitely thin, also insist for a dynamic test of some kind. In a first-person shooter you don't want your players using a cheat to teleport through a wall by moving too fast. One way to handle this is to do a simple test of the player's path versus the nearest wall plane. Another is to create a plane for each wall with the normals pointing into the room; if a plane test shows that the object is on the negative side of the plane, then it's no longer in the room.

Submarines are large and move relatively slowly for their size, so for this collision system we don't need to worry about this issue. However, it is good to be aware of it. For more information on managing dynamic tests, see Eberly [27].

11.4.4 Performance Improvements

Source Code
Demo
SweepPrune

Now that we've handled questions of which bounding shapes to use on our objects and how to achieve a tighter fit even with simple primitives, we'll consider ways of improving our performance. The main way we'll approach this is to cut down on intersection tests. We've already handled this to some extent at the model level, by using a bounding hierarchy to cut down on intersection tests between primitives. Now we want to look at the world level, by cutting down on tests between models. For example, if two objects are relatively small and at opposite ends of the map from each other, it's a pretty good bet that they're not colliding.

The most basic way to check collisions among all objects is the following loop:

```
for each object i
    for each object j, where j <> i
        test for collision between i and j
```

There are a number of problems with this. First of all, we're doing $n(n-1)$ tests, which is an $O(n^2)$ algorithm. Half of those tests are duplicates: if we test for collision between objects 1 and 5, we'll also test for collision between 5 and 1. Also, there may be a number of objects that we wish to collide with that simply aren't moving. We don't want to test collision between two such static objects. A better loop which handles these cases is

```
for each object i
    for each object j, where j > i
        if (i is moving or j is moving)
            test for collision between i and j
```

There are other possibilities. We can have two lists: one of moving objects called `Colliders` and one of moving or static objects called `Collidables`. In the first loop we iterate through the `Colliders` and in the second the `Collidables`. Each `Collider` should be tagged after its turn through the loop, to ensure collision pairs aren't checked twice. Still, even with this change, we're still doing $O(nm)$ tests, where n is the number of `Colliders` and m is the number of `Collidables`. We need to find a way to further cut down the number of checks.

Most approaches involve some sort of spatial subdivision to do this. The simplest is to slice the world, along the x-axis say, by a series of evenly spaced planes (Figure 11.29). This creates a set of slabs, bounded by the planes along the x-direction, and by whatever bounds we've set for our world in the y- and z-directions. For each slab, we store the set of objects that intersect it. To test for collisions for a particular object, we determine which slabs it intersects and then test against only the objects in those slabs. This approach can be extended to other spatial subdivisions, such as a grid or voxel-based system.

One of the disadvantages of the regular spatial subdivisions is that they don't handle clumping very well. Let's consider slabs again. If our world is fairly sparse, there may be large numbers of slabs with no objects in them, and a very few with most of the objects in them. We still may end up doing a large number of checks within each slab—which is the problem we were trying to avoid.

There is another possibility used by a number of collision-detection systems, known as the *sweep-and-prune* method. It is similar to the separating axis test that we used for OBBs (it's also related to some scanline rasterization algorithms). Instead of using a regular grid for our world, we'll use the extents of our objects as our grid. For each object, we project its extents onto the x-axis. To keep things efficient, we can use our root-level bounding sphere to compute our extents, which for a sphere with center C and radius r, gives us an interval of $[c_x - r, c_x + r]$.

Given the extent endpoint pairs for each object, we'll mark them with a pointer to the object, and indicate for each value whether it is the low (*start*) or high (*finish*) endpoint. Finally, we sort all endpoints from low to high.

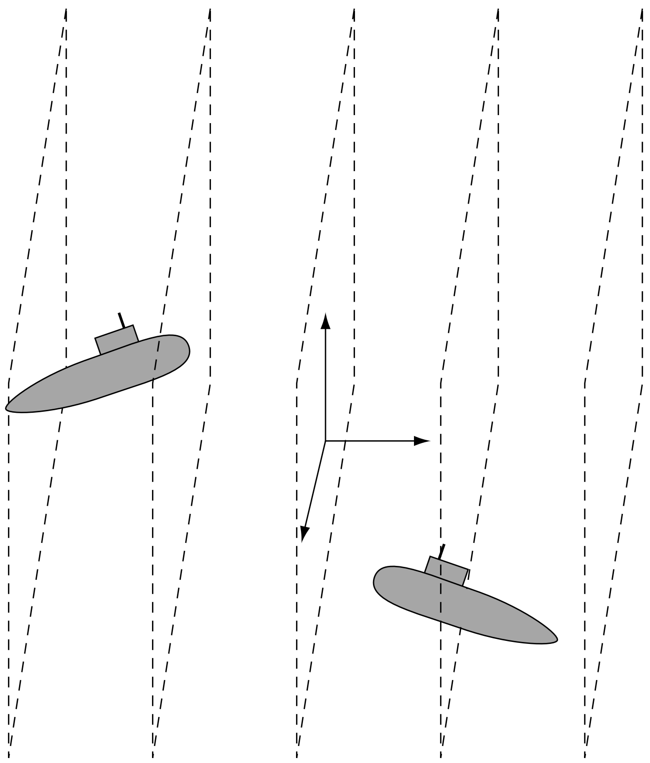


FIGURE 11.29 Cutting collision space into slabs.

Once the sorted list of endpoints is created, the collision detection process runs as follows:

```
for each endpoint do
   if a start point
   if object is moving
      check collisions against all objects in list
   else
      check collisions against moving objects in list
   add corresponding object to list
else if a finish point
   remove corresponding object from list
```

Figure 11.30 shows how this works. We sweep from left to right along the *x*-axis and use the sorted endpoints to test intersections of intervals before the more complex intersection tests.

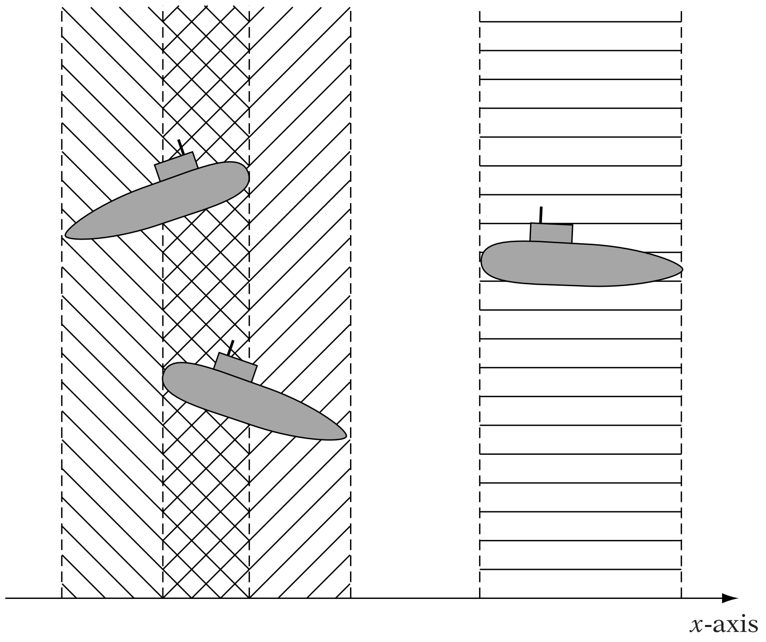


Figure 11.30 Dividing collision space by sweep and prune.

Normally this would be an $\Omega(n \log n)$ algorithm due to the sorting operation. However, if the timestep is small enough, the relative position of the objects won't have changed that much from frame to frame — this is referred to as temporal coherence. Any changes that do happen will be rare but localized. Therefore, if we use a sorting algorithm that works best on mostly sorted lists, such as bubble or insertion sort, we can get linear time for our sort and hence an $O(n)$ algorithm.

This algorithm still has problems, of course. If our objects are highly localized (or clumped) in the x direction, but separated in the y direction, then we may still be doing a high number of unnecessary intersection tests. But it is still much better than the naive $O(n^2)$ algorithm we were using before.

11.4.5 Related Systems

The other two systems we mentioned earlier were ray casting, for picking and AI tests, and frustum culling. Both systems can benefit from the techniques described in our collision system, in particular the use of bounding hierarchies and spatial partitioning.

Consider the case of ray casting. Instead of testing the ray directly against the object, we can take the ray and pass it through the hierarchy until (if we desire) we get the exact triangle of intersection. Further culling of testing

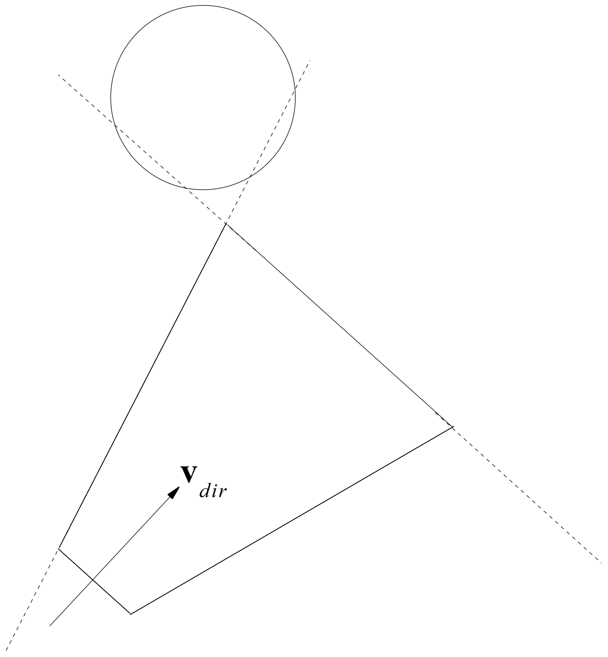


FIGURE 11.31 False positive for frustum intersection.

can be done by using a spatial partitioning system such as voxels to consider only those objects that lie in the areas of the spatial partitioning that intersect the ray.

When handling frustum culling, the most basic approach involves testing an object against the six frustum planes. If, after this test, we determine that the object lies outside one of the planes, then we consider it outside the frustum and do not render it. As with ray casting, we can improve performance by using a bounding hierarchy at progressive levels to remove obvious cases. We can also use a spatial partition again, and consider only objects that lie in the areas of the partition within the view frustum.

However, there is one aspect of frustum culling of which we need to be careful. This also applies to any intersection test that requires determining whether we are inside a convex object. Consider the situation shown in Figure 11.31. The bounding sphere is near the corner of the view frustum and clearly intersecting two planes. By using the scheme described, this sphere would be considered as intersecting the frustum, but it is clearly not. An alternative is shown in Figure 11.32a. Instead of using the frustum, we trace around the frustum with the bounding sphere to get a rounded, larger frustum[1]. This represents the maximum extent that a bounding sphere can

1. This process is also known as convolution.

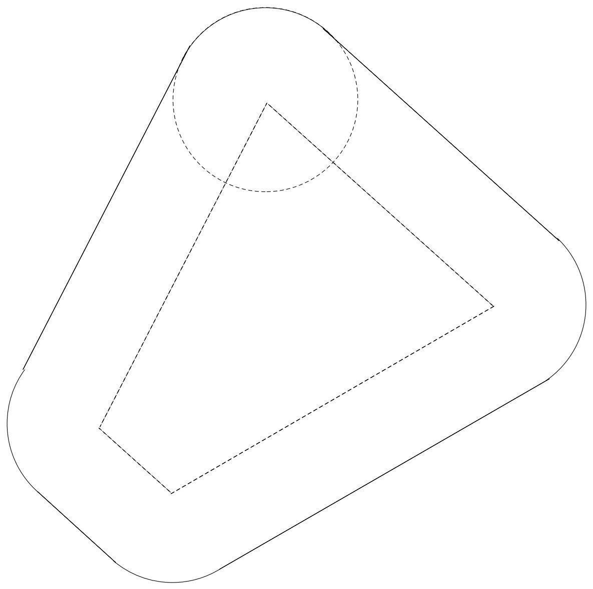


Figure 11.32a Expanding view frustum for simpler inclusion test.

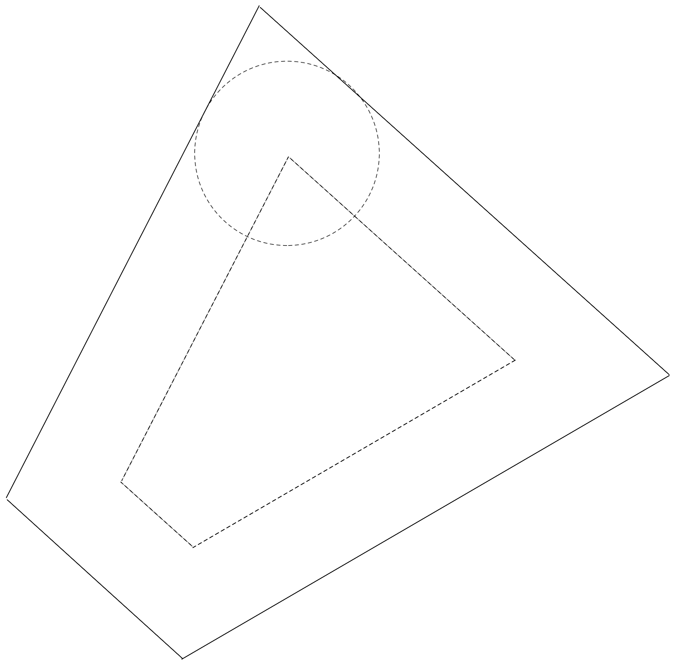


Figure 11.32b Expanding view frustum for simpler inclusion test.

have and still be inside the frustum. Instead of testing the sphere, we can test its center against this shape. In practice we can just push out the frustum planes by the sphere radius (Figure 11.32b), which is close enough. Similar techniques can be used for other bounding objects; see Möller and Haines [79] or Watt and Policarpo [113] for more details.

11.4.6 Section Summary

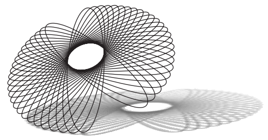
The proceeding should give some sense of the decisions that have to be made when handling collision detection or other systems that involve object intersection: pick base primitives, choose when you'll use them, consider whether to manage dynamic intersections, and cull unnecessary tests. However, this shouldn't be taken as the only approach. There are many other possible algorithms that handle much more complex cases than these. For example, there are systems, such as the University of North Carolina's I-COLLIDE, that track closest pairs of objects. This allows for considerable culling of intersection tests. There are also more sophisticated methods for managing spatial partitions, such as portals, octrees, BSP trees, and *kd*-trees. Whether the algorithmic complexity is necessary will depend on the application.

11.5 Chapter Summary

Testing intersection between geometric primitives is a standard part of any interactive application. This chapter has presented a few examples to provide a taste of how such algorithms are created. Most derive from a careful use of the basic properties of vectors and points as presented in Chapter 1. Using our intersection methods wisely allows us to build an efficient system for detecting collision between objects, casting rays for AI visibility checks and picking, and frustum culling.

For those who are interested in reading further, a more thorough presentation of geometric distance and intersection methods can be found in Schneider and Eberly [96]. These techniques fall under a general class of algorithms known as computational geometry; good references are Preparata and Shamos [91], and O'Rourke [84]. Two different approaches to building collision detection systems can be found in van den Bergen [109] and Ericson [34]. Finally, use of intersection techniques in rendering, plus information on more complex spatial partitioning techniques, can be found in both Möller and Haines [79] and Watt and Policarpo [113].

Chapter 12 Rigid Body Dynamics



12.1 Introduction

In many games, we move our objects around using a very simple movement model. In such a game, if we hold down the up arrow key, for example, we apply a constant forward translation, once a frame, to the object until the key is released, at which point the object immediately stops moving. Similarly, we can apply a constant rotation to the object if the left arrow key is held, and again, it stops upon release. This is fine for something with fast action like a platform game or a first-person shooter, where we want quick response to our input. As soon as we hit a key, our character starts moving and stops immediately upon release. This motion model is known as *kinetics* and can be thought of as an application of the theories of Aristotle.

But suppose we want to do a more realistically styled game, for example, a submarine game. Submarines don't start and stop on a dime. When the propeller starts turning, it takes some time for the submarine to start forward. And they don't really have instantaneous brakes — when the engine is shut off they will drift for quite a while before stopping. Turning is much the same — they will respond slowly to application of the rudder and then straighten out over time.

Even in a fast action game, we may want to model how objects in the world react to our main character. When we push an object, we don't expect it to stop instantly when we stop pushing, nor do we expect it to keep moving forever. If we knock a chair over, we don't expect it to fall straight back and then stick to the floor; we expect it to turn depending on where we hit it, and then bounce and possibly roll once. We want the game world to react to our character as the real world reacts to us, in a physically correct manner.

For both of these cases, we will want a better model of movement, known as a physically based simulation. One chapter is hardly enough space to encompass this broad topic, which covers the preceding effects, as well as objects deforming due to contact, fluid simulation, and soft body simulations such as cloth and rope. Instead, we'll concentrate on a simplified problem which is useful in many circumstances: objects that don't deform (known as *rigid bodies*) and move based on Newton's laws of motion (known as *dynamics*). We'll discuss techniques for translating rigid bodies through space in a physically based manner (linear dynamics) and then how to encompass rotational effects (rotational dynamics). Finally, we'll discuss some methods for handling collisions within our simulation, again covering linear and rotational effects in turn.

The convention in physics is to represent some vector quantities by capital letters. To maintain compatibility with physics texts we will use the same notation and assume that the reader can distinguish between such quantities and the occasional matrix by context.

12.2 Linear Dynamics

12.2.1 Moving with Constant Acceleration

Let's consider our object's movement through our game world as a function $X(t)$, which represents the position of the object for every time t. If we plot just the x values against t for our simple motion model, we would end up with a graph similar to that in Figure 12.1. Notice that we travel in a straight line for a while and then turn sharply in another direction, or we hold position. This is like our piecewise linear interpolation, except that in this case, the future x values are unknown; they are determined by the input of the player. For a given frame i, this can be represented by a line equation

$$X_i(h_i) = X_i + h_i\mathbf{v}_i$$

where X_i represents the position at the start of frame i, $\mathbf{v}_i$ is a vector generated from the player input which points along each line segment, and h_i is our frame time. We'll simplify things further by considering just the function on the first line segment, from time $t \geq 0$:

$$X(t) = X_0 + t\mathbf{v}_0$$

where $X_0 = X(0)$.

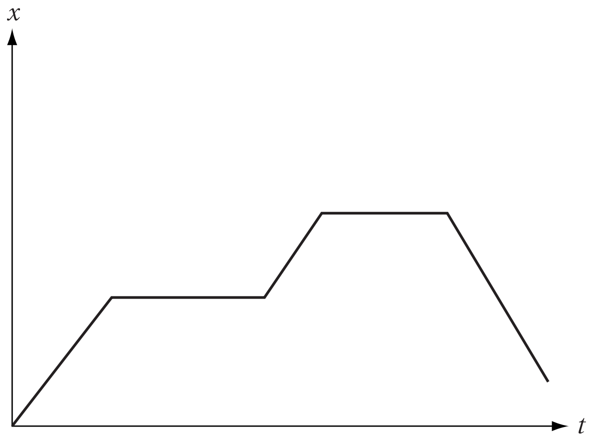


FIGURE 12.1 Graph of current motion model, showing x-coordinate of particle as a function of time.

If we take the derivative of this function with respect to t, we end up with

$$\frac{d\mathbf{X}}{dt} = \mathbf{X}'(t) = \mathbf{v}_0 \tag{12.1}$$

This derivative of the position function is known as *velocity*, which is usually measured in meters per second, or m/s. For our original motion model, we have a constant velocity across each segment. If we continue taking derivatives, we find that the second derivative of our position function is zero, which is what we'd expect when our velocity is constant.

Now let's assume that our second derivative, instead of being zero, is a constant nonzero function. To achieve this, we'll change our velocity function to

$$\mathbf{v}(t) = \mathbf{v}_0 + t\mathbf{a} \tag{12.2}$$

Now $\mathbf{v}(t)$ is also an affine function, this time with a constant derivative vector $\mathbf{a}$, called acceleration, or

$$\frac{d\mathbf{v}}{dt} = \mathbf{v}'(t) = \mathbf{a} \tag{12.3}$$

The units for acceleration are usually measured in meters per second squared, or m/s^2.

Our original function $X(t)$ used a constant $\mathbf{v}_0$, so now we'll need to rewrite it in terms of $\mathbf{v}(t)$. Since $\mathbf{v}$ is changing at a constant rate across our time interval, we can instead use the average velocity across the interval, which is just one-half the starting velocity plus the ending velocity, or

$$\bar{\mathbf{v}} = \frac{1}{2}(\mathbf{v}_0 + \mathbf{v}(t))$$

Substituting this into our original $X(t)$ gives us

$$X(t) = X_0 + t\left[\frac{1}{2}(\mathbf{v}_0 + \mathbf{v}(t))\right]$$

Substituting in for $\mathbf{v}(t)$ gives the final result of

$$X(t) = X_0 + t\mathbf{v}_0 + \frac{1}{2}t^2\mathbf{a} \tag{12.4}$$

Our equation for position becomes a quadratic equation, and our velocity is represented as a linear equation:

$$\begin{aligned} P_i(t) &= P_i + t\mathbf{v}_i + \frac{1}{2}t^2\mathbf{a}_i \\ \mathbf{v}_i(t) &= \mathbf{v}_i + t\mathbf{a}_i \end{aligned}$$

So given a starting position and velocity, and an acceleration which is constant over the entire interval $[0, t]$, we can compute any position within the interval. As an example, let's suppose we have a projectile, with an initial velocity $\mathbf{v}_0$ and initial position P_0. We represent acceleration due to gravity by the constant g, which is 9.8 m/s^2. This acceleration is applied only downward, or in the $-z$ direction, so $\mathbf{a}$ is the vector $(0, 0, -g)$. If we plot the z component as a function of t, then we get a parabolic arc, as seen in Figure 12.2. This function will work for any projectile (assuming we ignore air friction), from

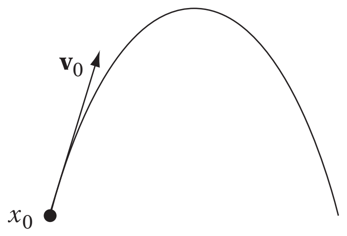


FIGURE 12.2 Parabolic path of object with initial velocity and affected only by gravity.

a thrown rock (low initial velocity) to a cannonball (medium initial velocity) to a bullet (high initial velocity)[1].

Within our game, we can use these equations on a frame-by-frame basis to compute the position and velocity at each frame, where the time between frames is h_i. So for a given frame $i + 1$:

$$
\begin{aligned}
X_{i+1} &= X_i + h_i \mathbf{v}_i + \frac{1}{2} h_i^2 \mathbf{a}_i \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + h_i \mathbf{a}_i
\end{aligned}
$$

12.2.2 Forces

One question that has been left open is how to compute our acceleration value. We do so based on a vector quantity known as a *force*. Forces cause change in an object's motion, pushing or pulling it around, either to speed it up or slow it down. So for example, to throw a ball your hand and arm exert a certain force on it, to begin its motion through the air. That force, when applied, produces an acceleration directly proportional to the object's mass, measured in kilograms. The proportional relationship is shown in Newton's second law of motion:

$$\mathbf{F} = m\mathbf{a}$$

The units for force end up being $kg\text{-}m/s^2$, or newtons, in homage to its creator.

In the previous section, we represented gravity as an acceleration, but in truth it is a force whose value is always proportional to the mass of the object. For an object with mass m on Earth, its magnitude is mg and its direction points to the center of the Earth, although we usually assume the world is locally flat. Other possible forces include the friction caused by air or water molecules pushing against an object to slow it down, or the thrust generated by a rocket engine or propeller, or simply the *normal force* of the ground pushing up to counteract gravity (there has to be such a force; otherwise we'd sink into the earth). In general, if something is pushing or pulling on an object, there is a force there.

Usually we have more than one force applied to an object at a time. Taking our ball example, we have the initial force when the ball is thrown, force due to gravity, and forces due to air resistance and wind. After the ball leaves your hand, that pushing force will be removed, leaving only gravity and air effects. Forces are vectors, so in both cases we can add all forces on an object together

1. In most cases, this last is approximated by a line equation for efficiency reasons.

to create a single force which encapsulates their total effect on the object. We then scale the total force by $1/m$ to get the acceleration for equation 12.4.

For simplicity's sake, we will assume for now that our forces are applied in such a way that we have no rotational effects. In Section 12.4 we'll discuss how to handle such cases.

12.2.3 Linear Momentum

As we've seen, the relationship between acceleration and velocity is

$$\mathbf{a} = \frac{d\mathbf{v}}{dt}$$

There is a corresponding related entity **P** for a force **F**, which is

$$\mathbf{F} = m\mathbf{a} = m\frac{d\mathbf{v}}{dt} = \frac{d\mathbf{P}}{dt}$$

The quantity $\mathbf{P} = m\mathbf{v}$ is known as the *linear momentum* of the object, and it represents the tendency for an object to remain in its current linear motion. The heavier the object or faster it is moving, the greater the force needed to change its velocity. So while a pebble at rest is easier to kick aside than a boulder, this is not necessarily true if the pebble is shot out of a gun.

An important property of Newtonian physics is the conservation of momentum. Suppose we take a collection of objects and treat them as a single system of objects. Now consider only the forces within the system, that is, only those forces acting between objects. Newton's third law of motion states that for every action, there is an equal and opposite reaction. So for example, if you push on the ground due to gravity, the ground pushes back just as much, and the forces cancel. Due to this, within the system, pairwise forces between objects will cancel and the total force is zero. If the external force is 0 as well, then

$$\mathbf{F} = \frac{d\mathbf{P}}{dt} = 0$$

so **P** is constant. No matter how objects may move within the system, the total momentum must be conserved. This property will be useful to us when we consider collisions.

12.2.4 Moving with Variable Acceleration

There is a problem with the approach that we've been taking so far: we are assuming that total force, and hence acceleration, is constant across the entire interval. For more complex simulations this is not the case. For example, it is common to compute a drag force proportional to but opposite in direction to velocity:

$$\mathbf{F}_{drag} = -m\rho\mathbf{v} \tag{12.5}$$

This can provide a simple approximation to air friction; the faster we go, the greater the friction force. The quantity ρ in this case controls the magnitude of drag. An alternative example is if we wish to model a spring in our system. The force applied depends on the current length of the spring, so the force is dependent on position:

$$\mathbf{F}_{spring} = -kX$$

The spring constant k fulfills a similar role to ρ: it controls the proportion of force dependent on the position. In both of these cases, since acceleration is directly dependent on the force, it will vary over the time interval as velocity or position vary. It is no longer constant. So for these cases, equations 12.2 and 12.4 are incorrect.

In order to handle this, we'll have to use an alternative approach. We begin by deriving a function for velocity in terms of any acceleration. Rewriting equation 12.3 gives us

$$d\mathbf{v} = \mathbf{a}\,dt$$

To find $\mathbf{v}$ we take the indefinite integral or antiderivative of both sides

$$\int d\mathbf{v} = \int \mathbf{a}\,dt$$

For example, if we assume as before that $\mathbf{a}$ is constant, we can move it outside the integral sign

$$\int d\mathbf{v} = \mathbf{a}\int dt$$

And integrating gives us

$$\mathbf{v} = t\mathbf{a} + \mathbf{c}$$

We can solve for $\mathbf{c}$ by using our velocity $\mathbf{v}_0$ at time $t = 0$:

$$\begin{aligned}\mathbf{c} &= \mathbf{v}_0 - 0 \cdot \mathbf{a}\\ &= \mathbf{v}_0\end{aligned}$$

So our final equation is as before:

$$\mathbf{v}(t) = \mathbf{v}_0 + t\mathbf{a}$$

We can perform a similar integration for position. Rewriting equation 12.1 gives

$$dX = \mathbf{v}(t)dt$$

We can substitute equation 12.2 into this to get

$$dX = \mathbf{v}_0 + t\mathbf{a}\ dt$$

Integrating this, as we did with velocity, produces equation 12.4 again.

For general equations we perform the same process, re-integrating $d\mathbf{v}$ to solve for $\mathbf{v}(t)$ in terms of $\mathbf{a}(t)$. So, using our drag example, we can divide equation 12.5 by the mass m to give acceleration:

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} = -\rho\mathbf{v}(t)$$

Rearranging this and integrating gives

$$\int d\mathbf{v} = \int -\rho\mathbf{v}(t)\ dt$$

We can consult a standard table of integrals to find that the answer in this case is

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\rho t}$$

where, as before, $\mathbf{v}_0 = \mathbf{v}(0)$.

While this particular equation was relatively straightforward, in general calculating an exact solution is not as simple as the case of constant acceleration. First of all, differential equations in which the quantity we're solving for is part of the equation are not always easily—if at all—solvable by analytic means. In many cases we will not necessarily be able to find an exact equation for $\mathbf{v}(t)$, and thus not for $X(t)$. And even if we can find a solution, every time we change our simulation equations, we'll have to integrate them again, and modify our simulation code accordingly. Since we'll most likely have many different possible situations with many different applications of force, this could grow to be quite a nuisance. Because of both these reasons, we'll have to use a numerical method that can approximate the result of the integration.

12.3 Initial Value Problems

12.3.1 Definition

The solutions for $\mathbf{v}$ and X that we're trying to integrate fall under a class of differential equation problems called *initial value problems*. In an initial value problem, we know the following about a function $\mathbf{y}(t)$:

1. An initial value of the function $\mathbf{y}_0 = \mathbf{y}(t_0)$
2. A derivative function $\mathbf{f}(t, \mathbf{y}) = \mathbf{y}'(t)$
3. A time interval h

The problem we're trying to solve is, given these parameters, what is the value at $\mathbf{y}(t_0 + h)$? For our purposes, this actually becomes a series of initial value problems: at each frame our previous solution becomes our new initial value $\mathbf{y}_i$, and our interval h_i will be based on the current frame time. Once computed, our new solution will become the next initial value $\mathbf{y}_{i+1}$. More specifically, the initial value $\mathbf{y}_i$ is our current position X_i and current velocity $\mathbf{v}_i$, stored in a single 6-vector as

$$\mathbf{y}_i = \begin{bmatrix} X_i \\ \mathbf{v}_i \end{bmatrix}$$

So how do we evaluate the derivative function $\mathbf{f}(t, \mathbf{y})$? This will be another vector quantity:

$$\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \mathbf{X}'_i \\ \mathbf{v}'_i \end{bmatrix}$$

The value of our derivative for X_i is our current velocity $\mathbf{v}_i$. Our derivative for $\mathbf{v}_i$ is the acceleration, which is based on the current total force. To compute this total force, it is convenient to create a function called `CurrentForce()`, which takes X and $\mathbf{v}$ as arguments and combines any forces derived from position and velocity with any constant forces, such as those created from player input. We'll represent this as $\mathbf{F}_{tot}(t, X, \mathbf{v})$ in our equations. So given our current state, the result of our function $\mathbf{f}(t, \mathbf{y})$ will be

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \mathbf{v}_i \\ \mathbf{F}_{tot}(t_i, X_i, \mathbf{v}_i)/m \end{bmatrix}$$

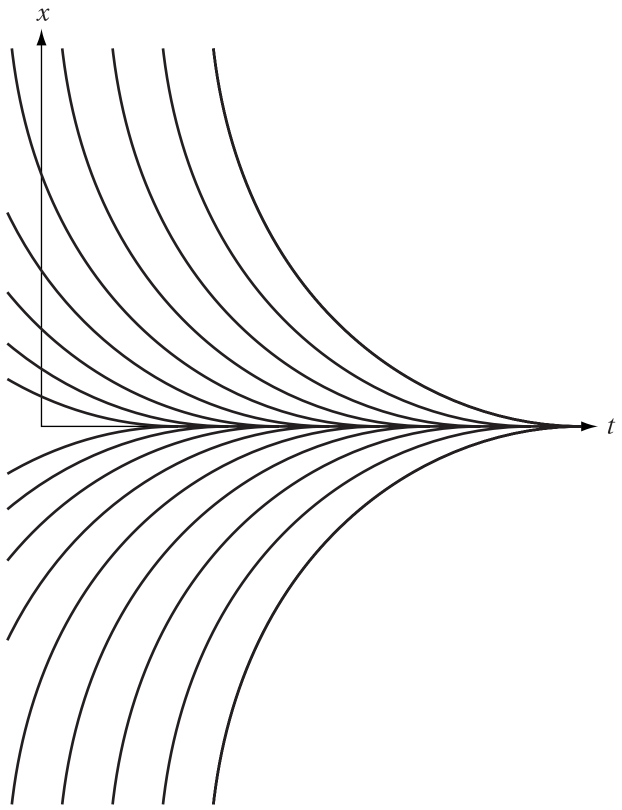


FIGURE 12.3 Various solution lines for initial value problem $x' = -\rho x$, integrated against time. Which line is correct depends on initial value chosen.

The function $\mathbf{f}(t, \mathbf{y})$ is important in understanding how we can solve this problem. If we graph such a function for a fixed t and $\mathbf{y}$ we can see that it is a vector field. Figure 12.3 shows a two-dimensional plot of one such vector field, accentuating certain lines of flow. If we start at a particular point and follow the vector flow, this will trace out one possible solution to the differential equation, starting at that initial value. This gives us a sense of what our general approach will be. We'll start at $\mathbf{y}_i$ and then, using our derivative function, take steps in time to generate new samples that approximate the function, until we generate an approximation for $\mathbf{y}_{i+1}$. In a way, we are doing the opposite of what we were doing when we were interpolating. Instead of generating an approximation to an unknown function based on known sample points, we're generating approximate sample points based on the derivative of an unknown function.

12.3.2 Euler's Method

Source Code
Demo
Force

Assuming our current time is t and we want to move ahead h in time, we could use Taylor's series to compute $\mathbf{y}(t+h)$:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\mathbf{y}'(t) + \frac{h^2}{2}\mathbf{y}''(t) + \cdots + \frac{h^n}{n!}\mathbf{y}^{(n)}(t) + \cdots$$

We can rewrite this to compute the value for timestep $i+1$, where the time from t_i to t_{i+1} is h_i:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i\mathbf{y}'_i + \frac{h_i^2}{2}\mathbf{y}''_i + \cdots + \frac{h_i^n}{n!}\mathbf{y}_i^{(n)} + \cdots$$

This assumes, of course, that we know all the values for the entire infinite series at timestep i, which we don't—we have only $\mathbf{y}_i$ and $\mathbf{y}'_i$. However, if h_i is small enough and all values of $\mathbf{y}''_i$ are bounded, we can use an approximation instead:

$$\begin{aligned}\mathbf{y}_{i+1} &\approx \mathbf{y}_i + h_i\mathbf{y}'_i \\ &\approx \mathbf{y}_i + h_i\mathbf{f}(t_i, \mathbf{y}_i)\end{aligned}$$

Another way to think of this is that we have a function $\mathbf{f}(t_i, \mathbf{y}_i)$ that, given a time t_i and initial value $\mathbf{y}_i$, can compute tangents to the unknown function's curve. We can start at our known initial value, and step h_i distance along the tangent vector to get to the next approximation point in the vector field (Figure 12.4).

Separating out position and velocity gives us

$$\begin{aligned}X_{i+1} &\approx X_i + h_iX'_i \\ &\approx X_i + h_i\mathbf{v}_i \\ \mathbf{v}_{i+1} &\approx \mathbf{v}_i + h_i\mathbf{v}'_i \\ &\approx \mathbf{v}_i + h_i\mathbf{F}_{tot}(t_i, X_i, \mathbf{v}_i)/m\end{aligned}$$

This is known as Euler's method.

To use this in our game, we start with our initial position and velocity. At each new frame, we grab the difference in time between the previous frame and current frame and use that as h_i. To compute $\mathbf{f}(t_i, \mathbf{y}_i)$ for the velocity, we use our `ComputeForces()` method to add up all of the forces on our object and divide the result by the mass to get our acceleration. Plugging in our current values, we use the preceding formulas to generate our new position and velocity. In code, this looks like

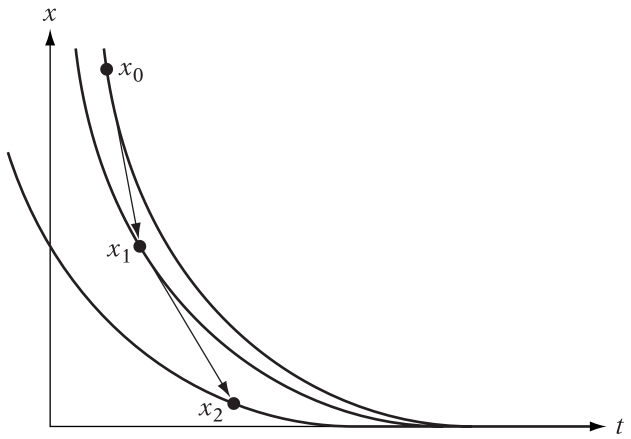


FIGURE 12.4 Using Euler's method to move from one position to another, using derivative function. Note that we end up stepping from one solution curve to another.

```
void
SimObject::Integrate( float h )
{
    IvVector3 accel;

    // compute acceleration
    accel = ComputeForces( mTime, mPosition, mVelocity ) / mMass;
    // clear small values
    accel.Clean();

    // compute new position, velocity
    mPosition += h*mVelocity;
    mVelocity += h*accel;
    // clear small values
    mVelocity.Clean();
}
```

It's important to compute the new velocity after the new position in this case, so that we don't overwrite the velocity prematurely.

Note that we clear near-zero values in the new velocity. This prevents little shifts in position due to tiny changes in velocity, such as those generated after an object has slowed down due to drag. While technically accurate, they can be visually distracting, so after a certain point we clamp our velocity to zero. The same is done with acceleration.

For many cases, this works quite well. If our time steps are small enough, then the resulting approximation points will lie close to the actual function and we will get good results. However, the ultimate success of this method is based on the assumption that the slope at the current point is a good estimate of the slope over the entire time interval h. If not, then the approximation can drift off the function, and the farther it drifts, the worse the tangent approximation can get. An example of this can be seen in Figure 12.5. The first step in our approximation takes us to a point in the vector field where the derivatives are flowing in the other direction, and we thus oscillate around the actual solution. Once the error grows, in many cases further steps don't get us back, and we continue to drift off of the actual solution.

For Euler's method, we say that the error is directly dependent on the time step, or $O(h)$. So one solution to this problem is to decrease the time step — for example take a step of $h/2$, followed by another step of $h/2$. While this may solve some cases, we may need to take a smaller time step, say $h/4$.

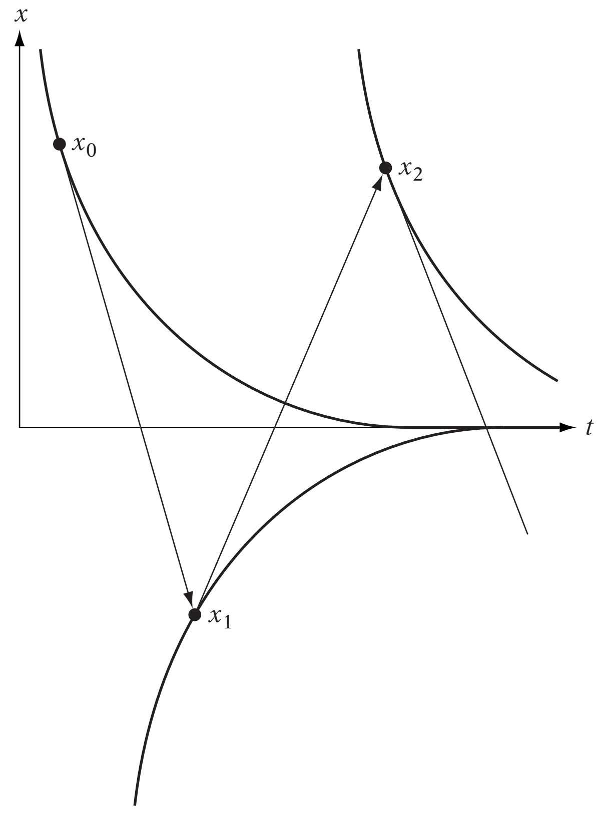


FIGURE 12.5 Taking too large a simulation step and oscillating around the solution.

And this may still lead to significant error. In the meantime, we are grinding our simulation to a halt while we recalculate quantities 4 or 8 or however many times for a single frame.

Situations that can lead to problems with Euler's method are often characterized by large forces. If we examine the remaining terms of the Taylor expansion,

$$\frac{h_i^2}{2}\mathbf{y}_i'' + \cdots + \frac{h_i^n}{n!}\mathbf{y}_i^{(n)} + \cdots$$

we can see why this could cause a problem. When we set up our approximation, we assumed that h_i was small and $\mathbf{y}_i''$ bounded. If we're considering position, a large force leads to a large acceleration, which leads to a larger difference between our approximation and the actual value. Larger values of h_i will magnify this error. Also, if the force changes quickly, this means that the magnitude of the velocity's second derivative is high, and so we can run into similar problems with velocity.

There are other issues with our particular example. It falls into a class of differential equations known as *stiff* systems. Situations that can lead to stiffness problems are often characterized by large spring and damping forces, such as in a stiff spring (hence the name). Examples of such systems have terms with rapidly decaying values, such as $e^{-\rho t}$ —exactly the case when we apply drag. These terms tend to zero as t approaches infinity but, as we've seen, won't always converge with a numerical method unless we control the step size appropriately. The larger ρ is, the smaller h must be. This can also affect systems where we wouldn't expect the term to contribute that much. For example, suppose the solution to our system is $y(t) = 1 + e^{-200t}$. As t increases from zero, $y(t)$ quickly approaches 1. However, approximating this with a numerical method without taking care to control the error can lead the e^{-200t} term to dominate the calculations, which leads to invalid results.

So while we can try to reduce stepsize, the number of calculations required may make it unworkable. Fortunately, there are other methods that we can try.

12.3.3 Midpoint Method

Source Code
Demo
Force

So far we've been using the derivative at the beginning of the interval as our estimate of the average tangent. A better possibility may be to take the derivative in the middle of the interval. To do this, we first use Euler's method to take a step halfway into the interval; that is, we integrate using a step size of $h/2$. Given our estimated position and velocity at the halfway point, we calculate $\mathbf{f}(t, \mathbf{y})$ at this location. We then go back to our original starting location, and

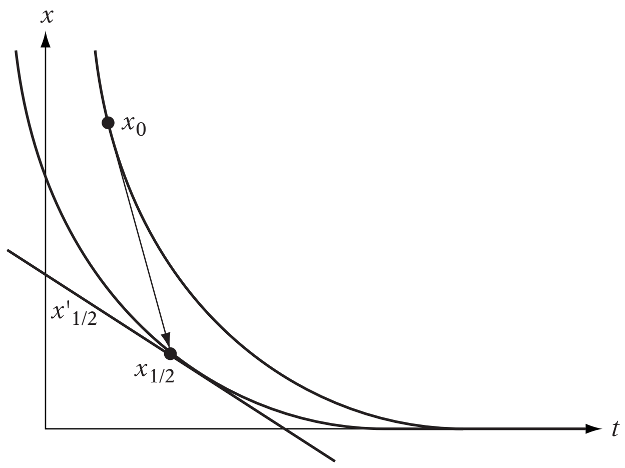


Figure 12.6a First step of midpoint method. Step one-half time increment using Euler's method and compute derivative there.

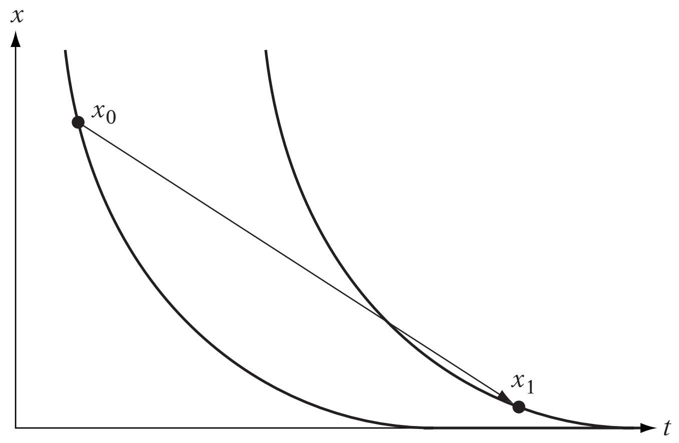


Figure 12.6b Using the midpoint derivative to step forward to our next position.

use the derivatives we calculated at the midpoint to move across the entire interval. This method is known as the *midpoint method*.

Figures 12.6a and 12.6b show how this works with our original function. In Figure 12.6a, the arrow shows our initial half-step, and the line our estimated tangent. Figure 12.6b uses the tangent we've calculated with our full time step, and our final location. As we can see, with this method we are following much closer to the actual solution and so our error is much less than before. The order of the error for the midpoint method is dependent on the square of the time step, or $O(h^2)$, which for values of h less than 1 is better than Euler's method. Instead of approximating the function with a line, we are approximating it with a quadratic.

Code to compute the midpoint method is as follows:

```
void
SimObject::Integrate( float h )
{
    IvVector3 totalForce = ComputeForces( mPosition, mVelocity );
    IvVector3 accel;

    // compute acceleration
    accel = 1.0f/mMass * totalForce;
    // clear small values
    accel.Clean();

    // compute midpoint position, velocity
    float h2 = 0.5f*h;
    IvVector3 midPosition = mPosition + h2*mVelocity;
    IvVector3 midVelocity = mVelocity + h2*accel;
    // clear small values
    midVelocity.Clean();

    // compute force there
    totalForce = ComputeForces( midPosition, midVelocity );
    accel = 1.0f/mMass * totalForce;
    // clear small values
    accel.Clean();

    // compute final position, velocity
    mPosition += h*midVelocity;
    mVelocity += h*accel;
    // clear small values
    mVelocity.Clean();
}
```

While the midpoint method does have better error tolerance than Euler's method, it still has problems when h gets large enough. To handle this, we'll have to consider some methods with better error tolerances still.

12.3.4 Higher-order Methods

Both the midpoint method and Euler's method fall under a larger class of algorithms known as *Runga-Kutta methods*. Whereas both of our previous techniques used a single estimate to compute a tangent for the entire interval, others within the Runga-Kutta family compute multiple tangents at fixed time steps across the interval and take their weighted average.

One possibility is to take the derivative at the end of the interval, and average with the derivative at the beginning. Like the midpoint method, we can't actually compute the derivative at the end of the interval, so we'll approximate it by performing normal Euler integration and computing the derivative at that point. This is known as the *modified Euler's method*. Interestingly, the error for this approach is still $O(h^2)$, due to the fact that we're taking an inaccurate measure of the final derivative. Another approach is *Heun's method*, which takes 1/4 of the starting derivative, and 3/4 of an approximated derivative 2/3 along the step size. Again, its error is $O(h^2)$, or no better than the midpoint method.

The standard $O(h^4)$ method is known as *Runga-Kutta order four*, or simply RK4. RK4 can be thought of as a combination of the midpoint method and modified Euler, where we weight the midpoint tangent estimates higher than the endpoint estimates. Representing this with our function notation:

$$
\begin{aligned}
\mathbf{u}_1 &= h_i \mathbf{f}(t_i, \mathbf{y}_i) \\
\mathbf{u}_2 &= h_i \mathbf{f}(t_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{u}_1) \\
\mathbf{u}_3 &= h_i \mathbf{f}(t_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{u}_2) \\
\mathbf{u}_4 &= h_i \mathbf{f}(t_i + h_i, \mathbf{y}_i + \mathbf{u}_3) \\
\mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{6}[\mathbf{u}_1 + 2\mathbf{u}_2 + 2\mathbf{u}_3 + \mathbf{u}_4]
\end{aligned}
$$

Clearly, improved accuracy doesn't come without cost. To perform standard Euler requires calculating a result for $f(t, \mathbf{y})$ only once. Midpoint, modified Euler, and Heun's need two calculations, and RK4 takes four. While to achieve the level of error tolerance of RK4 would require many more evaluations of Euler's method, using RK4 still adds both complexity and increased simulation time that may not be necessary. It does depend on your application, but for simple rigid-body simulations with fast frame rates and low accelerations, Euler's method or one of the other two methods will probably be suitable.

12.3.5 Verlet Integration

Source Code Demo Force

There is another class of integration methods, known as *Verlet methods*, that is commonly used in molecular dynamics. Verlet methods have come to the attention of the games community because they can be useful in simulating collections of small, unoriented masses known as particles—in particular, when constrained distances between particles are required [65]. Such systems of constrained particles can simulate soft objects such as cloth, rope, and dead bodies (this last is also known as rag doll physics).

The most basic Verlet method can be derived by adding the Taylor expansion for the current timestep with the expansion for the previous timestep:

$$\begin{aligned}\mathbf{y}(t+h)+\mathbf{y}(t-h) &= \mathbf{y}(t)+h\mathbf{y}'(t)+\frac{h^2}{2}\mathbf{y}''(t)+\cdots\\ &\quad+\mathbf{y}(t)-h\mathbf{y}'(t)+\frac{h^2}{2}\mathbf{y}''(t)-\cdots\end{aligned}$$

Solving for $\mathbf{y}(t+h)$ gives us

$$\mathbf{y}(t+h)=2\mathbf{y}(t)-\mathbf{y}(t-h)+h^2\mathbf{y}''(t)+O(h^4)$$

Rewriting in our stepwise format:

$$\mathbf{y}_{i+1}=2\mathbf{y}_i-\mathbf{y}_{i-1}+h_i^2\mathbf{y}_i''$$

This gives us an $O(h^2)$ solution for integrating position from acceleration, without involving velocity at all. This can be a problem if we want to use velocity in our calculations, but we can estimate it as

$$\mathbf{v}_i=\frac{(X_{i+1}-X_i)}{2h_i}$$

One question may be, How do we find the first $\mathbf{y}_{i-1}$? The standard method is to start the process off with one pass of standard Euler or other Runga-Kutta method and store the initial position and integrated position. From there we'll have two positions to apply to our Verlet integration.

Standard Verlet has a few advantages: it is time invariant, which means that we can run it forwards and then backwards and end up in the same place. Also, the lack of velocity means that we have one less quantity to calculate. Because of this, it is often used for particle systems, which generally are not dependent on velocity. However, if we want to apply friction based on velocity or when we want to handle spinning rigid objects, the lack of velocity and angular velocity makes it more difficult. There are ways around this, as described in [65], but in most cases it will be easier to use a method that allows us to track both velocity terms. One other disadvantage is that our velocity estimation is (a) not very accurate and (b) one time step behind our position.

If you wish to use Verlet methods and require velocity, you have two choices. *Leapfrog Verlet* tracks velocity, but at half a time step off from the position calculation:

$$\begin{aligned}\mathbf{v}(t+h/2) &= \mathbf{v}(t-h/2) + h\mathbf{a}(t)\\ X(t+h) &= X(t) + h\mathbf{v}(t+h/2)\end{aligned}$$

Like with standard Verlet, we can start this off with a Runga-Kutta method by computing velocity at a half-step and proceed from there. If velocity on a whole step is required, it can be computed from the velocities, but as with standard Verlet, one time step behind position:

$$\mathbf{v}_i = \frac{(\mathbf{v}_{i+1/2} - \mathbf{v}_{i-1/2})}{2}$$

As with standard Verlet, leapfrog Verlet is an $O(h^2)$ method.

The third, and most accurate, Verlet method is *velocity Verlet*:

$$\begin{aligned}X(t+h) &= X(t) + h\mathbf{v}(t) + \frac{h^2}{2}\mathbf{a}(t)\\ \mathbf{v}(t+h) &= \mathbf{v}(t) + h/2[\mathbf{a}(t) + \mathbf{a}(t+h)]\end{aligned}$$

Unlike with the previous Verlet methods, we now have to compute the acceleration twice: once at the start of the interval and once at the end. This can be done in a stepwise manner by:

$$\begin{aligned}\mathbf{v}_{i+1/2} &= \mathbf{v}_i + h_i/2\mathbf{a}_i\\ X_{i+1} &= X_i + h_i\mathbf{v}_{i+1/2}\\ \mathbf{v}_{i+1} &= \mathbf{v}_{i+1/2} + h_i/2\mathbf{a}_{i+1}\end{aligned}$$

In between the position calculation and the velocity calculation, we recompute our forces and then the acceleration $\mathbf{a}_{i+1}$. Note that in this case the forces can be dependant only on position, since we have added only half of the acceleration contribution to velocity. In the case of molecular dynamics or particles, this isn't a problem since most of the forces between them will be positional, but again, for rigid body problems this is not the case.

12.3.6 Implicit Methods

All the methods we've described so far integrate based on the current position and velocity. They are called *explicit methods* and make use of known quantities at each time step, for example Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{y}'_i$$

But even higher-order explicit methods don't handle extreme cases of stiff equations very well. Consider the following situation, as presented by Witkin and Baraff [119]. Suppose we have a planet revolving in a perfectly circular orbit. Our initial position is at the top of the orbit, and our initial velocity pointing out to the right. Our only force is gravity, pointing towards the center of the orbit. After one time step, our new position is off of the original orbit and into a new one (Figure 12.7). In addition, we've added a tiny bit of the gravitational acceleration to the velocity, making it slightly larger in magnitude. After two steps, our position is further off our desired position, and our velocity is still larger. As we continue, our approximation grows worse and worse, spiraling away from our actual function.

What has happened is that our simulation error has accumulated and we have added more and more energy to our system. One solution is to take tinier steps, but as we saw with other stiff systems, we're just trading simulation time for more accuracy.

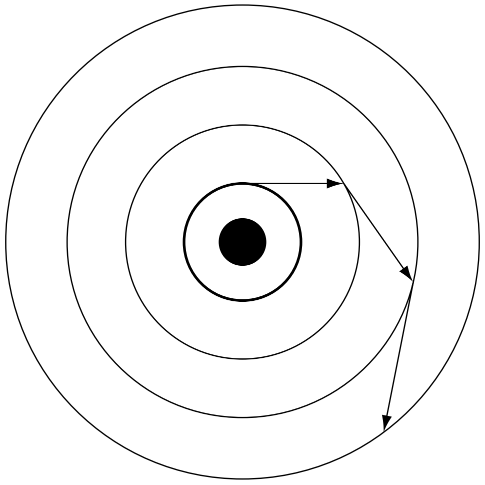


Figure 12.7 Using Euler's method to simulate an orbit. The result spirals off of the actual solution.

Implicit methods make use of quantities from the next time step:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \mathbf{y}'_{i+1}$$

This particular implicit method is known as *backwards Euler*. The idea is that we are going to grab the derivative at our destination rather than at our current position. That is, we are going to find a $\mathbf{y}_{i+1}$ with the derivative that, if we were to run the simulation backwards, would end up at $\mathbf{y}_i$.

Implicit methods don't add energy to the system, but instead lose it. This doesn't guarantee us more accuracy, but it does avoid simulations that spin out of control—instead, they'll dampen down to an equilibrium state. Since, in most cases, we're going to add a damping factor anyway, this is a small price to pay for a more stable simulation.

This sounds good in theory, but in practice, how do we calculate $\mathbf{y}'_{i+1}$? One way is to solve for it directly. For example, let's consider our air friction example again. Recall that our force is directly dependent on velocity, but in the opposing direction. Considering only velocity:

$$\mathbf{v}_{i+1} = \mathbf{v}_i - h\rho\mathbf{v}_{i+1}$$

Solving for $\mathbf{v}_{i+1}$ gives us

$$\mathbf{v}_{i+1} = \frac{\mathbf{v}_i}{1 + h\rho}$$

Figure 12.8 graphs this against the actual solution $\mathbf{v}_0 e^{-\rho t}$. Note that we don't converge as fast when using the implicit method. However, we do converge, and so this is better than the explicit method, which as we've seen oscillates wildly for large h values.

We can't always use this approach. Either we will have a function too complex to solve in this manner, or we'll be experimenting with a number of functions and won't want to take the time to solve each one individually. Another way is to use a *predictor-corrector* method. We move ahead one step using an explicit method to get an approximation. Then we use that approximation to calculate our $\mathbf{y}'_{i+1}$. This will be more accurate than the explicit method alone, but it does involve twice the number of calculations, and we're depending on the accuracy of the first approximation to make our final calculation.

Another more accurate approach is to rewrite the equation so that it can be solved as a linear system. If we represent $\mathbf{y}_{i+1}$ as $\mathbf{y}_i + \Delta\mathbf{y}_i$, and ignore the factor t, we can rewrite backwards Euler as

$$\mathbf{y}_i + \Delta\mathbf{y}_i = \mathbf{y}_i + h_i \mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$$

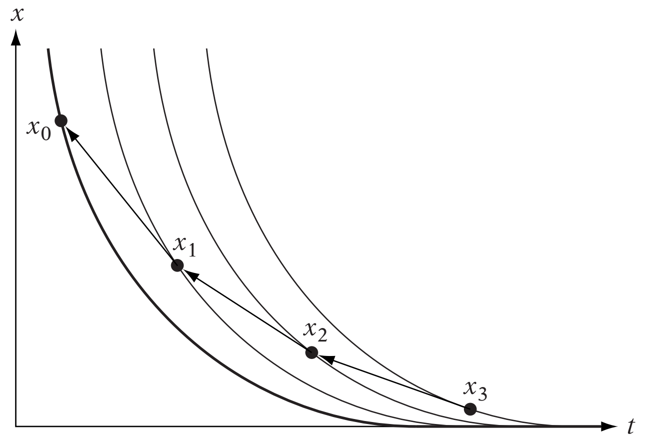


Figure 12.8 Comparing the exact solution with implicit Euler. The arrows for implicit Euler point backwards to indicate that we are getting the derivative from the next time step.

or

$$\Delta\mathbf{y}_i = h_i\mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$$

We can approximate $\mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$ as $\mathbf{f}(\mathbf{y}_i) + \mathbf{f}'(\mathbf{y}_i)\Delta\mathbf{y}_i$. Note that $\mathbf{f}'(\mathbf{y}_i)$ is a matrix since $\mathbf{f}(\mathbf{y}_i)$ is a vector. Substituting this approximation, we get

$$\Delta\mathbf{y}_i \approx h_i(\mathbf{f}(\mathbf{y}_i) + \mathbf{f}'(\mathbf{y}_i)\Delta\mathbf{y}_i)$$

Solving for $\Delta\mathbf{y}_i$ gives

$$\Delta\mathbf{y}_i \approx \left(\frac{1}{h_i}\mathbf{I} - \mathbf{f}'(\mathbf{y}_i)\right)^{-1}\mathbf{f}(\mathbf{y}_i)$$

In most cases, this linear system will be sparse, so it can be solved in near-linear time. More information can be found in [119].

As mentioned, implicit methods are really only necessary when our equations are so stiff that explicit methods are not practical. Examples of situations where implicit methods are useful are when simulating cloth, rope, or rag doll physics. In general it is better to begin with an explicit method because it is more efficient, and only if you see wild oscillations or other signs of stiff systems do you look into implicit methods.

12.4 Rotational Dynamics

12.4.1 Definitions

The equations and methods that we've discussed so far allow us to create physical simulations which modify an object's position. However, one aspect of dynamics we've passed over is simulating changes in an object's orientation due to the application of forces, or *rotational dynamics*. When discussing rotational dynamics, we use quantities that are very similar to those used in linear dynamics. Comparing the two:

Linear	**Rotational**
position X	orientation $\mathbf{\Omega}$ or $\mathbf{q}$
velocity $\mathbf{v}$	angular velocity ω
force $\mathbf{F}$	torque τ
linear momentum $\mathbf{P}$	angular momentum $\mathbf{L}$
mass m	inertial tensor $\mathbf{J}$

We'll discuss each of these quantities in turn.

12.4.2 Orientation and Angular Velocity

Orientation we have seen before; we'll represent it by a matrix $\mathbf{\Omega}$ or a quaternion $\mathbf{q}$. The angular velocity ω represents the change in orientation, or

$$\omega = \frac{d\mathbf{\Omega}}{dt}$$

It is a vector quantity, where the vector direction is the axis we rotate around to effect the change in orientation, and the length of the vector represents the rate of rotation around that axis, in radians per second.

The orientation and angular velocity are applied to an object around a point known as the *center of mass*. The center of mass can be defined as the point associated with an object where, if you apply a force at that point, it will move without rotating. One can think of it as the point where the object would perfectly balance. Figure 12.9 shows the center of mass for some common objects. The center of mass for a seesaw is directly in the center, as we'd expect. The center of mass for a hammer, however, is closer to one end than the other, since the head of the hammer is more massive than the handle.

For our objects, we'll assume that we have some sense of where the center of mass is — either it's set by the artist or by some other means. One possibility discussed shortly is to compute the center of mass directly from

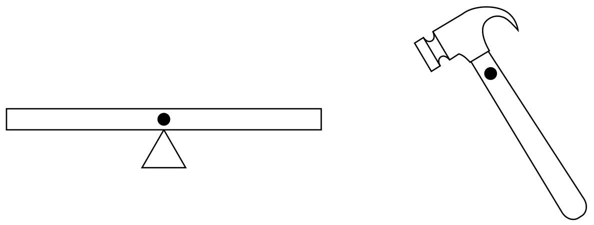


Figure 12.9 Comparing centers of mass. The seesaw balances close to the center, while the hammer has center of mass closer to one end.

our model data. Other choices are to use the local model origin, or the bounding box center (or centroid) as an approximation. Once the center of mass is determined, it is usually convenient to translate our object so that we can treat the local model origin as the center of mass, and therefore use the same orientation and position representation for both simulation and rendering.

It is possible to convert from angular velocity to linear velocity. Given an angular velocity ω, and a point at displacement $\mathbf{r}$ from the center of mass, we can compute the linear velocity at the point by using the equation

$$\mathbf{v} = \omega \times \mathbf{r} \tag{12.6}$$

This makes sense if we look at a rotating sphere. If we look at various points on the sphere (Figure 12.10a), their linear velocity is orthogonal to both the axis of rotation and their displacement vector, and this corresponds to the direction of the cross product. The length of $\mathbf{v}$ will be

$$\|\mathbf{v}\| = \|\omega\|\|\mathbf{r}\|\sin\theta$$

where θ is the angle between ω and $\mathbf{r}$. This also makes sense. As the rate of rotation $\|\omega\|$ increases, we'd expect the linear velocity of each point on the object to increase. As we move out from the equator, a rotating point has to move a longer linear distance in order to maintain the same angular velocity relative to the center (Figure 12.10b), so as $\|\mathbf{r}\|$ increases, $\|\mathbf{v}\|$ will increase. Finally, the linear velocity of a point as we move from the equator to the poles will decrease to zero (Figure 12.10c) and the quantity $\sin\theta$ provides this.

12.4.3 Torque

Up until now we've been simplifying our equations by applying forces only at the center of mass, and therefore generating only linear motion. On the other hand, if we apply an off-center force to an object, we expect it to spin.

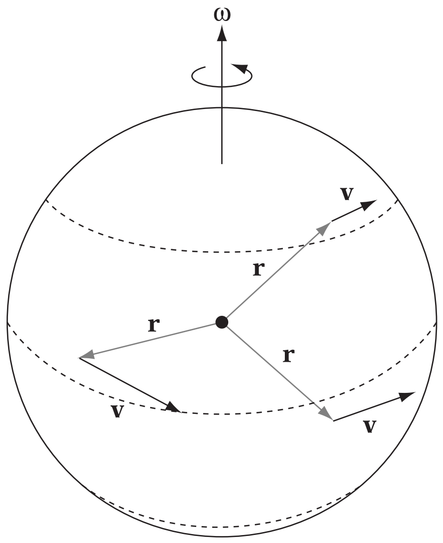


FIGURE 12.10a Linear velocity of points on surface of rotating sphere. Velocity is orthogonal to both angular velocity vector and displacement vector from center of rotation.

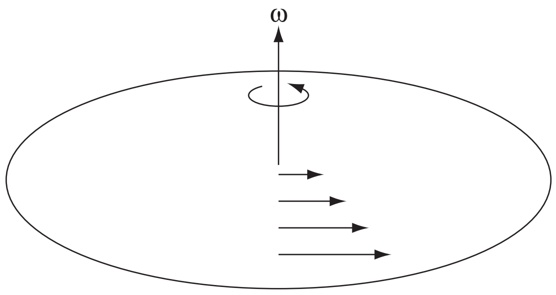


FIGURE 12.10b Comparison of speed of points on surface of rotating disk. Points farther from center of rotation have larger linear velocity.

The rotational force created, known as *torque*, is directly dependent on the location where the force is applied. The farther away from the center of mass we apply a given force, the larger the torque. To compute torque, we take the cross product of the vector from the center of mass to the force application point, with the corresponding force (Figure 12.11) or

$$\tau = \mathbf{r} \times \mathbf{F} \tag{12.7}$$

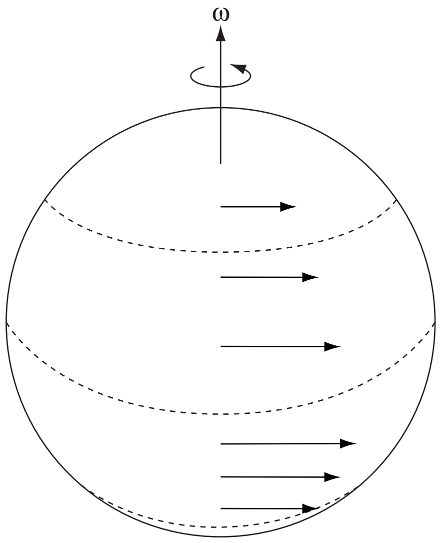


Figure 12.10C Comparison of speed of points on surface of rotating sphere. Points closer to equator of sphere have larger linear velocity.

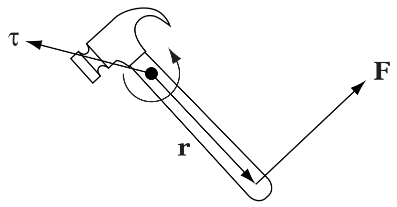


Figure 12.11 Computing torque. Torque is the cross product of displacement vector and force vector.

The direction of τ combined with the right-hand rule tells us the direction of rotation the torque will attempt to induce. If you align your right thumb along the direction of torque, your curled fingers will indicate the direction of rotation—if the vector is pointing towards you this is counterclockwise around the axis of torque. The magnitude of τ provides the magnitude of the corresponding torque.

To compute the total torque, we need to compute the corresponding torque for each application of force, and then add them up. Adding the offsets and taking the cross product of the resulting vector with the total force will not compute the correct result, as shown by Figure 12.12. The sum of the offsets is $\mathbf{0}$, producing a torque of $\mathbf{0}$, which is clearly not the case—the true total torque as shown will start the circle rotating counterclockwise.

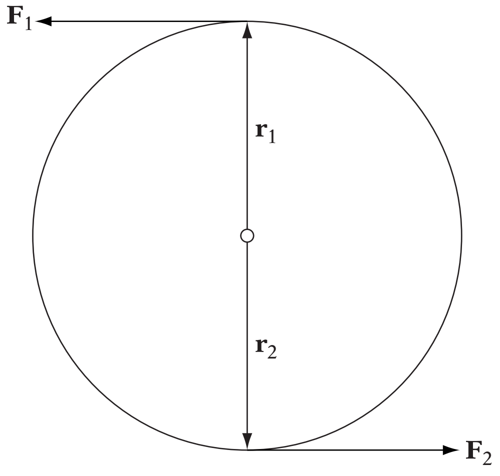


Figure 12.12 Adding two torques. If forces and displacements are added separately and then the cross product is taken total torque will be 0. Each torque must be computed and then added together.

12.4.4 Angular Momentum and Inertial Tensor

Recall that a force **F** is the derivative of the linear momentum **P**. There is a related quantity **L** for torque, such that

$$\tau = \frac{d\boldsymbol{L}}{dt}$$

Like linear momentum, the angular momentum **L** describes how much an object tends to stay in motion, but in rotational motion rather than linear motion. The higher the angular momentum, the larger the torque needed to change the object's angular velocity. Recall that linear momentum is equal to the mass of the object times its velocity. Angular momentum is similar, except that we use angular velocity, and the rotational equivalent of mass, the inertial tensor matrix:

$$\mathbf{L} = \mathbf{J}\omega \tag{12.8}$$

Why use a matrix **J** instead of a scalar, as we did with mass? The problem is that while shape has no effect (other than, say, for friction) on the general equations for linear dynamics, it does have an effect on how objects rotate. Take the classic example of a figure skater in a spin. As she starts the spin, her arms are out from her sides, and she has a low angular velocity. As she brings her arms in, her angular velocity increases until she opens her arms again to gracefully pull out of the spin. Torque is near-zero in this case (ignoring some minimal friction from the ice and air), so we can consider angular momentum to be constant. Since angular velocity is clearly changing and mass is constant, the shape of the skater is the only factor that has a direct effect to cause this change.

So to represent this effect of shape on rotation, we use a 3×3 symmetric matrix, where

$$\mathbf{J} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

We need these many factors because, as we've said, rotation depends heavily on shape and each factor describes how the rotation changes around a particular axis. The diagonal elements are called the *moments of inertia*. If we're in the correct coordinate frame, then the nondiagonal elements, or *products of inertia*, are zero. For such a frame, the axes are called the *principle axes*. For example, if the object is symmetric, the principle axes lie along the axes of symmetry and through the center of mass. We'll see next how to handle the case if our object is *not* in the principle axes frame.

The following are some examples of simple inertial tensors for objects with constant density and mass m:

Sphere (radius of r):

$$\begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$$

Solid cylinder (main axis aligned along x, radius r, length d):

$$\begin{bmatrix} \frac{1}{2}mr^2 & 0 & 0 \\ 0 & \frac{1}{4}mr^2 + \frac{1}{12}md^2 & 0 \\ 0 & 0 & \frac{1}{4}mr^2 + \frac{1}{12}md^2 \end{bmatrix}$$

Box ($x_{dim} \times y_{dim} \times z_{dim}$):

$$\begin{bmatrix} \frac{1}{12}m(y_{dim}^2 + z_{dim}^2) & 0 & 0 \\ 0 & \frac{1}{12}m(x_{dim}^2 + z_{dim}^2) & 0 \\ 0 & 0 & \frac{1}{12}m(x_{dim}^2 + y_{dim}^2) \end{bmatrix}$$

For many purposes, these can be reasonable approximations. If necessary, it is possible to compute an inertia tensor and center of mass for a generalized model, assuming a constant density. An initial description of how do to do this is provided in [78], while more detail and code optimized for triangular data can be found in [30].

12.4.5 Integrating Rotational Quantities

Source Code Demo Torque

As with linear dynamics, we use our angular velocity to update to our new orientation. Ideally, we could use Euler's method directly and compute our new orientation as

$$\mathbf{\Omega}_{i+1} = \mathbf{\Omega}_i + h\omega_i$$

However, this won't work, mainly because we are trying to combine vector and matrix quantities. What we need to do is compute a matrix that represents the derivative and use that with Euler's method.

Recall that the column vectors of a rotation matrix are three orthonormal vectors. We need to know how each vector will change with time; that is, we need the linear velocity at each vector tip. What we want to do is convert the angular velocity into a linear velocity for each of our basis vectors. We can apply equation 12.6 to each of our basis vectors to compute this, and then use the matrix generated to integrate orientation. One way would be to take the cross product of ω with each column vector, but instead we can take our three angular velocity values, and create a skew symmetric matrix $\tilde{\omega}$, where

$$\tilde{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{12.9}$$

If we multiply this by our current orientation matrix, this will take the cross product of ω with each column vector, and we end up with the derivative of orientation in matrix form. Using this with Euler's method, we end up with

$$\mathbf{\Omega}_{n+1} = \mathbf{\Omega}_n + h(\tilde{\omega}_n \mathbf{\Omega}_n) \tag{12.10}$$

If we're using a quaternion representation for orientation, we use a similar approach. We take our angular velocity vector and convert it to a quaternion $\mathbf{w}$, where

$$\mathbf{w} = (0, \omega)$$

We can multiply this by one-half of our original quaternion to get the derivative in quaternion form, giving us, again with Euler's method,

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\left(\frac{1}{2}\mathbf{w}_n\mathbf{q}_n\right) \tag{12.11}$$

A derivation of this equation is provided by Witken and Baraff [119] and Eberly [30], for those who are interested.

Using either of these methods allows us to integrate orientation. As far as updating angular velocity, computing acceleration for rotational dynamics is rather complicated, so we won't be using angular acceleration at all. Instead, since torque is the derivative of angular momentum, we'll integrate the torque to update angular momentum, and then compute the angular velocity from that. As when we integrated force, we'll need a function to compute total torque across the entire interval, called `CurrentTorque()`. For both methods, we'll have to modify our input variables to take into account orientation and angular velocity as well as position and velocity.

To find the angular velocity, we rewrite equation 12.8 to solve for ω:

$$\omega = \mathbf{J}^{-1}\mathbf{L} \tag{12.12}$$

When computing the angular velocity in this way, there is one detail that needs to be managed carefully. The inertial tensor is in the local space of the object. However, angular momentum is integrated from torque, which is computed in world space, and we want our resulting angular velocity to also be in world space. To keep things consistent, we need a way to convert our local $\mathbf{J}^{-1}$ to world space. If we're using a rotation matrix to represent orientation, we can use it to transform $\mathbf{L}$ from world to local space, apply the inverse inertial tensor, and then transform back into world space. So, for a given time step:

$$\omega_{i+1} = \mathbf{\Omega}_{\mathrm{i+1}}\mathbf{J}^{-1}\mathbf{\Omega}^T_{\mathrm{i+1}}\mathbf{L}_{\mathrm{i+1}} \tag{12.13}$$

If we're using quaternions, the most efficient way to handle this is to convert our quaternion to a matrix, and then compute equation 12.13.

Using Euler's method and quaternions, the full code for handling rotational quantities looks like:

```
// compute new orientation, angular momentum
IvQuat w = IvQuat( 0.0f, mAngVelocity.x,
                   mAngVelocity.y, mAngVelocity.z );
mRotate += h*0.5f*w*mRotate;
mRotate.Normalize();
mRotate.Clean();
```

```
    mAngularMomentum += h*CurrentTorque( mTranslate, mVelocity,
                                         mRotate, mAngVelocity);
    mAngularMomentum.Clean();

    // update angular velocity
    IvMatrix33 rotateMat(mRotate);
    IvMatrix33 worldMomentsInverse =
               rotateMat*mMomentsInverse*::Transpose(rotateMat);
    mAngularVelocity = worldMomentsInverse*mAngularMomentum;
    mAngularVelocity.Clean();
```

12.5 Collision Response

Up to this point, we haven't considered collisions. Our objects are moving gracefully through the world, speeding up or slowing down as we adjust our forces. All of which is accurately modeled, except that the objects go right through each other. Not a very realistic or fun game. Instead, we'll need a way to simulate the two objects bouncing away from each other due to the collision. We can do so by using the methods we've discussed in the previous chapter in combination with some new techniques.

12.5.1 Locating the Point of Collision

For the purposes of this discussion, we'll assume a simple collision model, where the objects are mostly convex and there aren't multiple collision points. To perform our collision response properly, we have to know two things about the collision. The first is the exact point of collision between the two objects A and B—in other words, the point on the objects where they just touch (Figure 12.13). Since the two objects are just touching, there is a tangent plane which passes between the two, which also intersects both at that point. This is represented in the figure as a line. The second thing we need to know is the normal $\hat{\mathbf{n}}$ to that plane. We'll choose our normal to point from A, the first object; to B, the second.

Our main problem in figuring out collision location is that we're trying to detect collisions within an interval of time. In one time step, two objects may be completely separate; in the next, they are colliding. In fact, in most cases when collision is detected, we have missed the initial point of collision and the objects are already interpenetrating (Figure 12.14). Because of this, there is no single point of collision.

One possibility for finding the exact point when initial collision occurs is to do a binary search within the time interval. We begin by running our

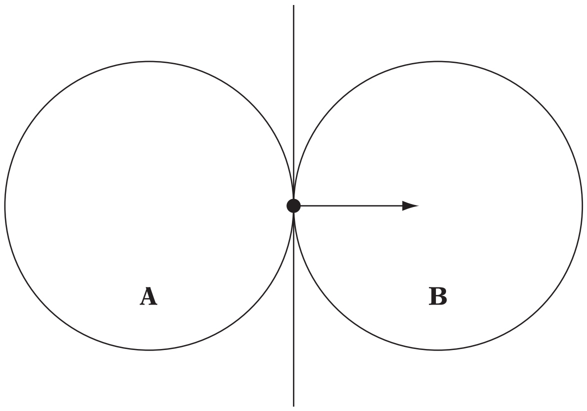


FIGURE 12.13 Point of collision. At the moment of impact between two convex objects, there is a single point of collision. Also shown is the collision plane and its normal.

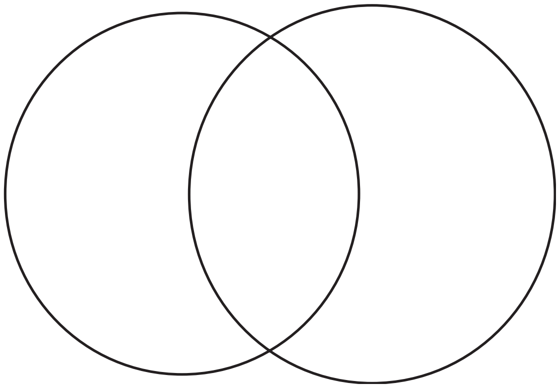


FIGURE 12.14 Interpenetrating objects. There is no single point of collision.

simulation, and then testing for collisions. If we find one, and the two objects involved are interpenetrating, we start our binary search:

```
dt = h/2
diff = h/4
while (dt > VerySmallNumber)
{
    Integrate from current time to current time+dt
    if just touching
      break
```

```
        else if intersecting
          dt -= diff
        else
          dt += diff
           diff /= 2
    }
```

At the end of the search, we'll either have found the exact collision point or will be reasonably close.

This technique has a few flaws. First of all, it's slow. Chances are that every time you get a collision, you'll need to run the simulation at least two or three additional times to get a point where the objects are just touching. In addition, in order for detection to be perfectly accurate, you need to rerun the simulation for all the objects, because their position at the time of the collision will be slightly different than their position at the end of the time interval. This may affect which objects are colliding. So you need to run the simulation back, determine the collision point, apply the collision response, and then run the simulation forward until you hit another collision, do another binary search, and so on. In the worst case, with many colliding objects, your simulation will get bogged down, and you'll end up with long frame times. The accuracy of this method may be suitable for offline simulation, but it's not good for interactivity.

Another possibility is to ignore it, approximate the collision location and normal, and let the collision response push the two objects apart. This can work, but if the response is too slow, the two objects may remain interpenetrated for a while. This looks quite odd and ruins the illusion of reality (Figure 12.15).

The third alternative begins by looking at the overlap between the two objects. The longest distance along that overlap is known as the penetration distance. We can push the two objects apart by the penetration distance until they just touch, and then use the point and normal from that intersection for collision calculations.

For example, take two spheres (Figure 12.16), with centers C_a and C_b, and radii r_a and r_b. If we subtract one center C_a from the other center C_b, we get the direction for our collision normal. The penetration distance p is then the sum of the two radii minus the length of this vector, or

$$p = (r_a + r_b) - \|C_b - C_a\| \tag{12.14}$$

We can move each sphere in opposite directions along this normal by the distance $p/2$, which will move them to a position where they just touch. This assumes that both objects can move—if one is not expected to move, like a

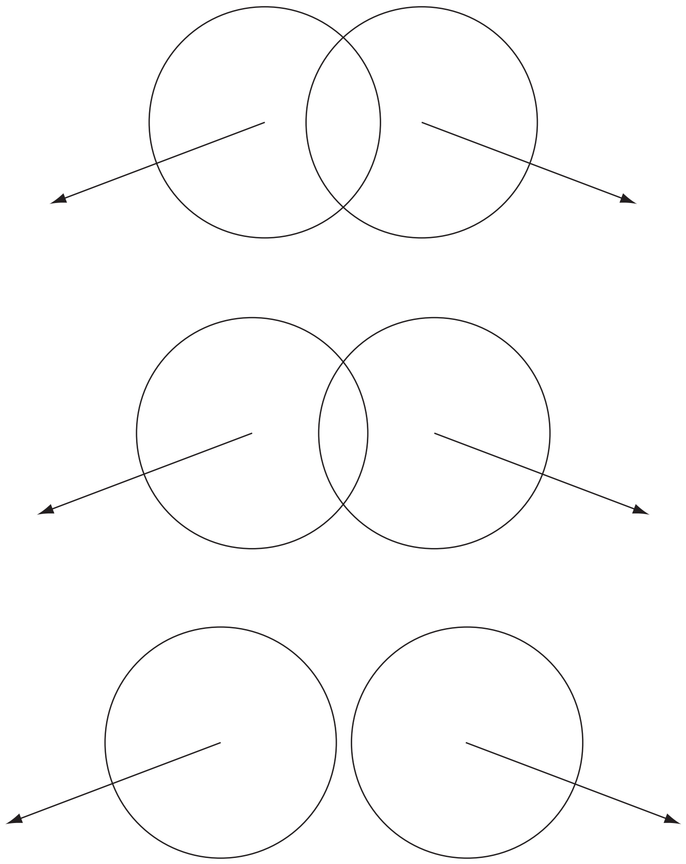


FIGURE 12.15 Allowing collision response to separate objects over time.

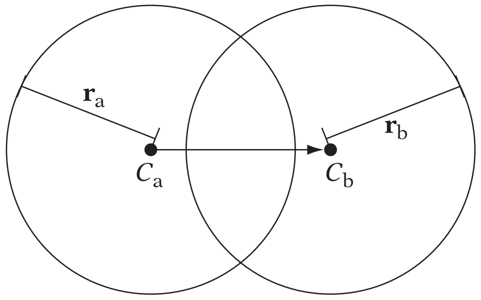


FIGURE 12.16 Determining penetration distance and collision normal.

boulder or a church, we translate the other object by the entire normal length. So for two moving objects A and B, the formula is

```
mTranslate -= 0.5f*penetration*centerDiff;
other->mTranslate += 0.5f*penetration*centerDiff;
```

Once we've pushed them apart, the collision point is where our center difference vector crosses the boundary of the two spheres. We can compute this point by halving the difference vector and adding it to the old C_a. We finish up by normalizing the difference vector to get our collision normal.

Handling penetration distance for capsules is just as simple. Instead of using the center points to compute the collision normal, we use the closest points on the line segments that define each capsule. The penetration distance becomes the sum of the radii minus the distance between these points. For bounding boxes, Eberly [27] provides a method that computes the penetration distance between two oriented boxes.

This technique does have some flaws. First, pushing the two objects apart by the entire penetration distance may look too abrupt. Instead, we can push them apart by a fraction of the penetration distance and assume that the collision response will separate them the rest of the way. The slight interpenetration will only be noticeable for one or two frames. Second, if objects are moving fast enough and the collision is detected too late, the two objects may pass through each other. If this case is not handled in the collision detection, we will get some very odd results when the objects are pushed apart. Finally, because we're pushing objects away from each other instantaneously, we may end up with situations where two objects collide, and one of them is moved into a third, causing a new interpenetration. Because we may have already tested for collision between the second pair of objects, we'll miss this collision. If we're expecting a large number of collisions between close objects, this system may not be practical.

12.5.2 Linear Collision Response

Source Code
Demo
LinCollision

Whatever method we use, we now have two of the properties of the collision we need to compute the linear part of our collision response: a collision normal $\hat{\mathbf{n}}$ and a collision point P. The other two elements are the incoming velocities of the two objects, $\mathbf{v}_a$ and $\mathbf{v}_b$. Using this information, we are finally ready to compute our collision response.

The technique we'll use is known as an *impulse-based* system. The idea is that near the time of collision, the forces and position remain nearly constant, but there is a discontinuity in the velocity. At one point in time, the velocities of the objects are heading towards one another — in the next infinitesimal

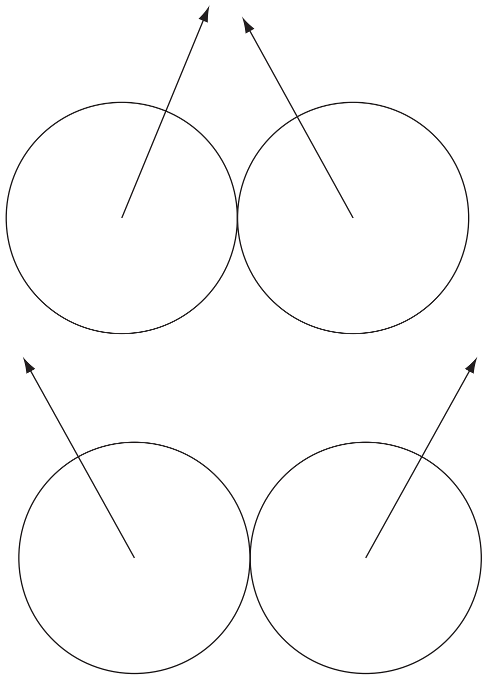


Figure 12.17 Instantaneous change in velocity at time of collision.

moment later, they are heading away (Figure 12.17). How much and in what relation the velocities change depends on the magnitude and direction of the incoming velocities, the direction of the collision normal, and the masses of the two objects.

Let's look again at the simple case of our two spheres A and B (Figure 12.18a). For now, let's assume their masses are equal. We again see our two incoming velocities $\mathbf{v}_a$ and $\mathbf{v}_b$, and our collision normal $\hat{\mathbf{n}}$. The idea is that we want to modify our velocity by an impulse velocity normal to the point of collision. The impulse will act to push the two objects apart—if the masses are equal, it will be equal in magnitude, but opposite in direction for each object. So we need to generate a scale factor j for our collision normal, and then add the scaled collision normal $j\hat{\mathbf{n}}$ and $-j\hat{\mathbf{n}}$ to $\mathbf{v}_a$ and $\mathbf{v}_b$ to get our outgoing velocities. So in order to compute the impulse vector, we need to compute this factor j.

To begin our computation, we need the relative velocity $\mathbf{v}_{ab}$, which is just $\mathbf{v}_a - \mathbf{v}_b$ (Figure 12.18a). From that, we'll compute the amount of relative velocity that is applied along the collision normal (Figure 12.18b). Recall that the dot product of any vector with a normalized vector gives the projection

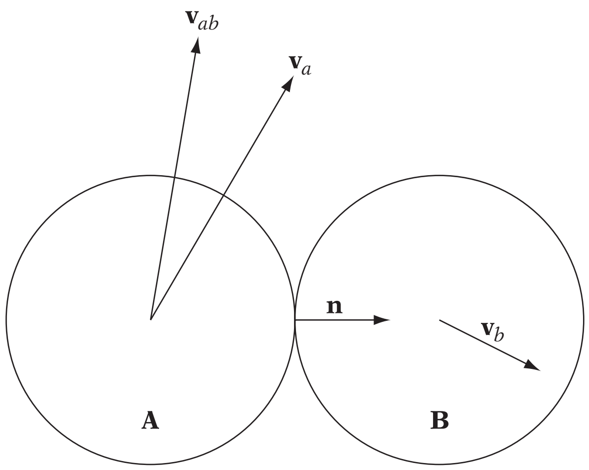


FIGURE 12.18a Computing collision response. Calculating relative velocity.

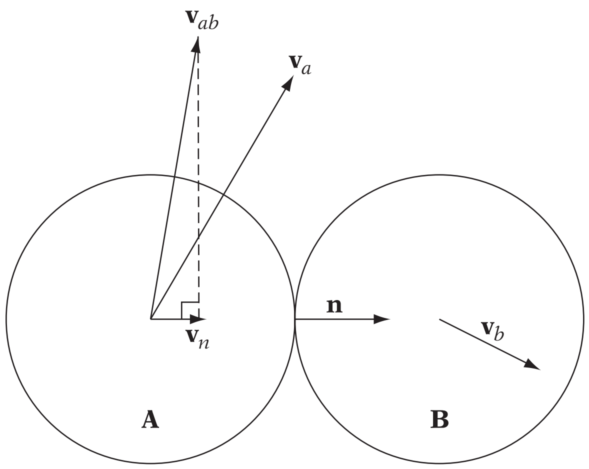


FIGURE 12.18b Collision response. Computing relative velocity along normal.

along the normal vector, which is just what we want. So

$$\mathbf{v}_n = (\,\mathbf{v}_{ab} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

At this point, we do one more test to see if we actually need to calculate an impulse vector. If the relative velocity along the collision normal is negative, then the two objects are heading away from each other and we don't need to compute an impulse. We can break out of the collision response code and proceed to the next collision. Otherwise, we continue with computing j.

In order to compute a proper impulse, two conditions need to be met. First of all, we need to set the ratio of the outgoing velocity along the collision normal to the incoming velocity. We do this by using a *coefficient of restitution* ϵ:

$$\mathbf{v}'_n = -\epsilon\, \mathbf{v}_n$$

or

$$(\mathbf{v}'_a - \mathbf{v}'_b) \cdot \hat{\mathbf{n}} = -\epsilon(\mathbf{v}_a - \mathbf{v}_b) \cdot \hat{\mathbf{n}} \tag{12.15}$$

This simulates two different physical properties. First of all, when two objects collide some energy is lost, usually in the form of heat. Second, if two objects are somewhat soft and/or sticky, or *nonelastic*, the bonding forces between the objects will decrease the outgoing velocities. Elastic in this case doesn't refer to the stretchiness of the object, but how resilient it is. A superball is relatively hard, but has very elastic collisions. So the quantity ϵ represents how much energy is lost and how elastic the collision between the two objects is. If ϵ is 1, then the two objects will bounce away from each other with the same relative velocities they had coming in. If ϵ is 0, they will stick together like two clay balls and move as one. Values in between will give a linear range of elastic responsiveness. Values greater than 1, or less than 0, are not permitted. An ϵ greater than 1 would add energy into the system, so a ball bouncing on a flat surface would bounce progressively higher and higher. An ϵ less than 0 means that the objects would be highly attracted to each other upon collision and would lead to undesirable interpenetrations.

Even if energy is not quite conserved (technically it is, but we're not tracking the heat loss), then momentum is. Because of this, the total momentum of the system of objects before and after the collision needs to be equal. So

$$m_a\, \mathbf{v}_a + j\hat{\mathbf{n}} = m_a \mathbf{v}'_a$$

or

$$\mathbf{v}'_a = \mathbf{v}_a + \frac{j}{m_a}\hat{\mathbf{n}} \tag{12.16}$$

Similarly,

$$m_b\, \mathbf{v}_b - j\hat{\mathbf{n}} = m_b\, \mathbf{v}'_b$$

or

$$\mathbf{v}'_b = \mathbf{v}_b - \frac{j}{m_b}\hat{\mathbf{n}} \tag{12.17}$$

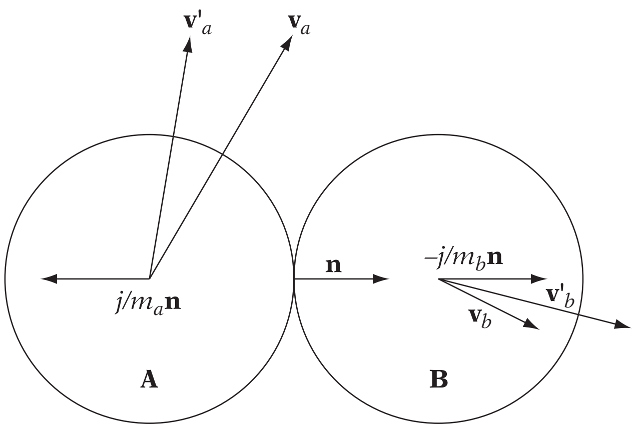


Figure 12.18c Collision response. Adding impulses to create outgoing velocities.

With this, we finally have all the pieces that we need. If we substitute equations 12.16 and 12.17 into equation 12.15 and solve for j, we get the final impulse factor equation

$$j = \frac{-(1+\epsilon)\,\mathbf{v}_{ab}\cdot\hat{\mathbf{n}}}{\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}\left(\frac{1}{m_a}+\frac{1}{m_b}\right)} \tag{12.18}$$

Now that we have our impulse value, we substitute this back into equations 12.6 and 12.17 to get our outgoing velocities (Figure 12.18c). Note the effect of mass on the outgoing velocities. As we expect, as the mass of an object grows larger, it grows more resistant to changing its velocity due to an incoming object. This is counteracted by j, which grows as relative velocity increases, or as the combined masses increase.

Our final algorithm for collision response between two spheres is as follows:

```
float radiusSum = mRadius + other->mRadius;
collisionNormal = other->mTranslate - mTranslate;
float distancesq = collisionNormal.LengthSquared();
// if distance squared < sum of radii squared, collision!
if ( distancesq <= radiusSum*radiusSum )
{
     // handle collision
     // penetration is distance - radii
     float distance = ::IvSqrt(distancesq);
     penetration = radiusSum - distance;
```

```
    collisionNormal.Normalize();

    // collision point is average of penetration
    collisionPoint = 0.5f*(mTranslate + mRadius*collisionNormal)
                + 0.5f*(other->mTranslate - other->mRadius*collisionNormal);

    // push out by penetration
    mTranslate -= 0.5f*penetration*collisionNormal;
    other->mTranslate += 0.5f*penetration*collisionNormal;

    // compute relative velocity
    IvVector3 relativeVelocity = mVelocity - other->mVelocity;

    float vDotN = relativeVelocity*collisionNormal;
    if (vDotN < 0)
         return;

    // compute impulse factor
    float numerator = -(1.0f+mElasticity)*vDotN;
    float denominator = (collisionNormal*collisionNormal);
    denominator *= (1.0f/mMass + 1.0f/other->mMass);
    float j = numerator/denominator;

    // update velocities
    mVelocity += j/mMass*collisionNormal;
    other->mVelocity -= j/other->mMass*collisionNormal;
}
```

In this simple example, we have interleaved the sphere collision detection with the computation of the collision point and normal. This is for efficiency's sake, since both use the sum of the two radii and the difference vector between the two centers for their computations. In a more complex collision system it is usually better to separate intersection detection from calculation of collision parameters. This is particularly true with hierarchical systems, where we may encounter many intersections between the bounding hierarchies of two objects. Only when we determine actual collision between leaf nodes do we calculate the collision normal and penetration distance.

12.5.3 Rotational Collision Response

Source Code
Demo
RotCollision

This is all well and good, but most objects are not spheres, which means that they have a visible orientation. When one collides with another at an offset to the center of mass, we would expect some change in angular velocity as well as linear velocity. In addition, any incoming angular velocity should affect the collision as well. A cue ball with spin (or English) applied causes a much different effect on a target pool ball than a cue ball with no spin.

As with linear and rotational dynamics, the way we handle rotational collision response is very similar to how we handle linear collision response. We need to modify only a few equations and recalculate our impulse factor j.

One modification we have to make is the effect of angular velocity on the incoming velocity. Up to this point, we've assumed that when the two objects strike each other, their surfaces are not moving, so the velocity at the collision point is simply the linear velocity. However, if one or both of the objects are rotating, then there is an additional velocity factor applied at the point of collision, as one surface passes by the other. Recall that equation 12.6 allows us to take an angular velocity ω and a displacement from the center of rotation $\mathbf{r}$ and compute the linear velocity contributed by the angular velocity at the point of displacement. Adding this to the original incoming velocities, we get

$$\bar{\mathbf{v}}_a = \mathbf{v}_a + \omega_a \times \mathbf{r}_a$$
$$\bar{\mathbf{v}}_b = \mathbf{v}_b + \omega_b \times \mathbf{r}_b$$

So now the relative velocity $\mathbf{v}_{ab}$ at the collision point becomes

$$\mathbf{v}_{ab} = \bar{\mathbf{v}}_a - \bar{\mathbf{v}}_b$$

and equation 12.15 becomes

$$(\bar{\mathbf{v}}'_a - \bar{\mathbf{v}}'_b) = -\epsilon(\bar{\mathbf{v}}_a - \bar{\mathbf{v}}_b) \tag{12.19}$$

The other change needed is that in addition to handling linear momentum, we also need to conserve angular momentum. This is a bit more complex compared to the equations for linear motion, but the general concept is the same. The outgoing angular momentum should equal the sum of the incoming angular momentum and any momentum imparted by the collision. For object A, this is represented by

$$\mathbf{I}_a\omega_a + \mathbf{r}_a \times j\hat{\mathbf{n}} = \mathbf{I}_a\omega'_a \tag{12.20}$$

or

$$\omega'_a = \omega_a + \mathbf{I}_a^{-1}(\mathbf{r}_a \times j\hat{\mathbf{n}}) \tag{12.21}$$

For object B, this is

$$\mathbf{I}_b\omega_b - \mathbf{r}_b \times j\hat{\mathbf{n}} = \mathbf{I}_b\omega'_b \tag{12.22}$$

or

$$\omega_b' = \omega_b - \mathbf{I}_a^{-1}(\mathbf{r}_b \times j\hat{\mathbf{n}}) \tag{12.23}$$

Just as with linear collision response, we can substitute equations 12.21 and 12.23 into 12.19, and solve for j to get

$$j = \frac{-(1+\epsilon)\,\mathbf{v}_{ab}\cdot\hat{\mathbf{n}}}{\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}\left(\frac{1}{m_a}+\frac{1}{m_b}\right)+\left[(\mathbf{I}_a^{-1}(\mathbf{r}_a\times\hat{\mathbf{n}}))\times\mathbf{r}_a+(\mathbf{I}_b^{-1}(\mathbf{r}_b\times\hat{\mathbf{n}}))\times\mathbf{r}_b\right]} \tag{12.24}$$

Using this j we calculate new angular momenta using equations 12.20 and 12.22 and from that calculate angular velocity as we did with angular dynamics, using equation 12.8. We use this same j for our linear collision response as well.

We change our linear collision handling code in three places to achieve this. First of all the relative velocity collision incorporates incoming angular velocity:

```
// compute relative velocity
IvVector3 r1 = collisionPoint - mTranslate;
IvVector3 r2 = collisionPoint - other->mTranslate;
IvVector3 vel1 = mVelocity + Cross( mAngularVelocity, r1 );
IvVector3 vel2 = other->mVelocity + Cross( other->mAngularVelocity, r2 );
IvVector3 relativeVelocity = vel1 - vel2;
```

Then we add angular factors to our calculation for j:

```
// compute impulse factor
float numerator = -(1.0f+mElasticity)*vDotN;
float denominator = (1.0f/mMass
     + 1.0f/other->mMass)*(collisionNormal.Dot(collisionNormal));

// compute angular factors
IvVector3 cross1 = Cross(r1, collisionNormal);
IvVector3 cross2 = Cross(r2, collisionNormal);
cross1 = mWorldMomentsInverse*cross1;
cross2 = other->mWorldMomentsInverse*cross2;
IvVector3 sum = Cross(cross1, r1) + Cross(cross2, r2);
denominator += (sum.Dot(collisionNormal));
```

Finally, in addition to linear velocity, we recalculate angular velocity:

```
// update angular velocities
mAngularMomentum += Cross(r1, collisionNormal);
mAngularVelocity = mWorldMomentsInverse*mAngularMomentum;
other->mAngularMomentum -= Cross(r2, collisionNormal);
other->mAngularVelocity = other->mWorldMomentsInverse*other->mAngularMomentum;
```

12.5.4 Other Response Techniques

There are some other techniques that have been used for collision response, with mixed results. The first is called the *penalty method*. Instead of generating an instantaneous change in velocity at a collision, the penalty method uses spring forces to push the objects away from each other. The more the two objects are interpenetrated, the larger the force. The problem with this method is that if you use small forces to avoid problems with stiff systems, your collisions look rather soft, and objects stay interpenetrated too long. And if you increase the forces to avoid the soft collisions, you end up with stiff systems and have to use implicit methods to solve your equations.

A *constraint system* is another technique which uses forces. Suppose we have a collection of particles, and want to keep each of them a fixed distance away from their neighbors, say in a grid (Figure 12.19). After any other force calculations are done, the constraint system analyzes the forces and velocities applied to each particle and computes exact forces to maintain the distance between the particles. Similar calculations can be done to keep particles on a wire or three particles at a relative angle. Constraint systems are very good for modeling chains, rope, cloth, or dead bodies. The downside is that in order to compute the exact forces, you have to solve large but sparse systems of

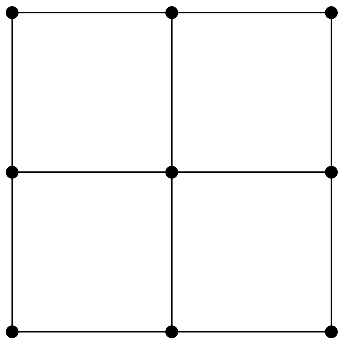


Figure 12.19 Mesh of particles constrained by distance.

linear equations. Also, constraint forces have trade-offs that are similar to those for penalty methods, in that you either end up with a stiff system or rather spongy simulations. Details for building a constraint system can be found in [119], [65], and [30].

12.6 Efficiency

Now that we have a simple simulation system, some notes on using it efficiently may be appropriate. The first rule is that this is a game. Don't waste time with any more processing power than you need to get the effect you want. While a fully realistic simulation may be desirable, it can't take too much processing power away from the other subsystems, for instance, graphics or AI. How resources are allocated among subsystems in a game depends on the game's focus. If a simpler solution will come close enough to the appearance of realism, then it is sometimes better to use that instead.

One way to reduce the amount of resources used is to simplify the problem. So far we've been assuming that we're building a truly 3D game, where the objects need to move in three degrees of freedom. If, however, you were building a tank game, it's highly unlikely that the tank would leave the ground. In most cases, land warfare games take place on a 2D map, with some height variation, so with the exception of projectiles the entire situation is really a 2D problem. You don't have to consider gravity, angular dynamics is constrained to just rotation around z, and thus you really need only one factor for your moments of inertia. This considerably simplifies the angular dynamics equations. The same is true for a first-person shooter; in general, characters will interact as cylinders sliding on a flat floor, with vertical walls as boundaries. In this case, we can simplify the collision problem to circles on a 2D plane.

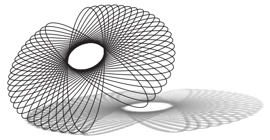
Another way to improve efficiency is to run simulation code only on some of the objects in the world. For example, we could restrict full simulation to those objects that are visible or near the player. We could use a simplified simulation model for the other objects or not move them at all. We could also not simulate objects that aren't currently moving, and begin simulation only when forces are applied or another object collides with them. When using this technique, we need to be careful about discontinuities in the simulation. We don't want a falling object that passes out of view to stop in midair, only to start falling again when it's visible again. Nor do we want objects to jerk, move strangely, or jump position as one simulation model ceases and another takes over. While managing these discontinuities can be tricky, using such restrictions can also gain quite a performance boost.

Simplifying the forces computed during simulation is another place to find speed improvements. We've alluded to this before. In a truly complete simulation we would compute a gravitational force, a normal force to keep the object from sinking through the ground, and a static frictional force to keep the object from sliding down any inclines. In most cases we can assume that the sum of all these forces is zero and ignore them completely. Friction is a similar case. We could compute a complex equation for an object that handles all contact points, current surface area, and whether we are moving or at rest — or we could just use a drag coefficient multiplied by velocity. If your game calls for the full friction model, then by all means do it, but in many cases it is overkill.

12.7 Chapter Summary

The use of physical simulation is becoming an important part of providing realistic motion in games and other interactive applications. In this chapter, we have described a simple physical simulation system, using basic Newtonian physics. We covered some techniques of numeric integration, starting with Euler's method, and discussed their pros and cons. Using these integration techniques, we have created a simple system for linear and rotational rigid body dynamics. Finally, we have shown how we can use the results of our collision system to generate impulses for collision response.

The system we've presented is a very simple one — we've barely scratched the surface of what is possible in terms of physical simulation. For those who are interested in proceeding further, Eberly [30] presents a more complete look at game physics, including the use of physics in graphics shaders. Burden and Faires [17] and Golub and Ortega [47] have more description of numerical integration techniques and managing error bounds. Finally, Witken and Baraff [119] and Jakobson [65] describe different methods for building constraint systems, useful for soft-body simulations such as cloth and rag doll.

Appendix A


Trigonometry Review

A.1 Basic Definitions

A.1.1 Ratios on the Right Triangle

The trigonometric functions sine, cosine, and tangent are based on ratios of the sides of a right triangle, relative to one acute angle θ (Figure A.1):

$$\begin{aligned} \sin\theta &= \text{opp}/\text{hyp} \\ \cos\theta &= \text{adj}/\text{hyp} \\ \tan\theta &= \text{opp}/\text{adj} = \sin\theta/\cos\theta \end{aligned}$$

We also define the reciprocal functions secant, cosecant, and cotangent as follows:

$$\begin{aligned} \sec\theta &= \text{hyp}/\text{adj} = 1/\cos\theta \\ \csc\theta &= \text{hyp}/\text{opp} = 1/\sin\theta \\ \cot\theta &= \text{adj}/\text{opp} = 1/\tan\theta \\ &= \cos\theta/\sin\theta = \sec\theta/\csc\theta \end{aligned}$$

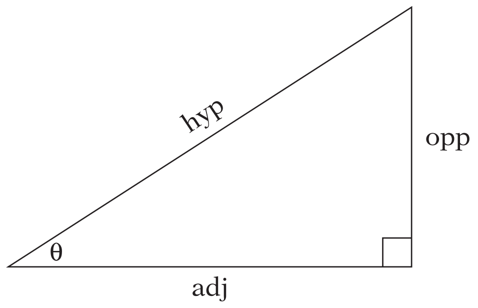


Figure A.1 Computing trigonometric functions on the right triangle.

A.1.2 Extending to General Angles

Consider a standard Cartesian frame for $\mathbb{R}^2$. We place a line segment, or radius, with length r and one endpoint fixed at the origin. The other endpoint is located at a point (x, y). We define θ as the angle between the radius and the positive x-axis. The angle is positive if the direction of rotation from the x-axis to the radius is counterclockwise, negative if clockwise. A full rotation is broken into 2π radians, or 360 degrees. The coordinate axes divide the plane into four quadrants: they are numbered in the order of rotation. Within this we can inscribe a right triangle, with the radius as hypotenuse and one side incident with the x-axis (Figure A.2).

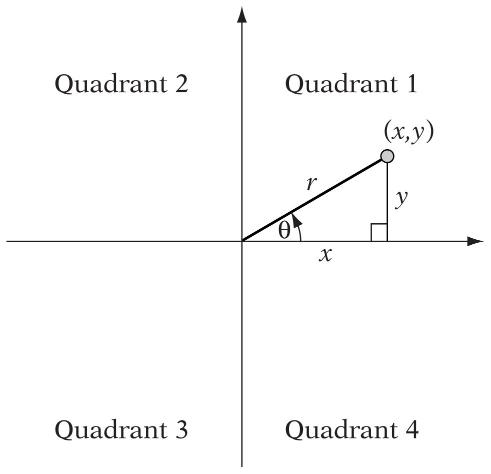


Figure A.2 Computing trigonometric functions on the standard Cartesian frame, showing the four ordered quadrants.

We can represent the sine and cosine based on the length r of the radius and the location (x, y) of the free endpoint:

$$\sin\theta = y/r$$
$$\cos\theta = x/r$$

In this case the tangent becomes the slope of the radius:

$$\tan\theta = y/x$$

For angles greater than $\pi/2$, the magnitude of the result is the same, but the sign may be negative depending on which quadrant the angle is in:

Functions	**Quadrant**	**Sign**
sin, csc	1,2	$+$
	3,4	$-$
cos, sec	1,4	$+$
	2,3	$-$
tan, cot	1,3	$+$
	2,4	$-$

The tangent, cotangent, secant, and cosecant all involve divisions by x or y, which may be 0. This leads to singularities at those locations, which can be seen in the function graphs in Figures A.3 through A.8. This sequence of figures shows the six trigonometric functions graphed against θ (in radians).

Also note that these functions are periodic. For example, $\sin(0) = \sin(2\pi) = \sin(-4\pi)$. In general, $\sin(x) = \sin(n \cdot 2\pi + x)$, for any integer n. The same is true

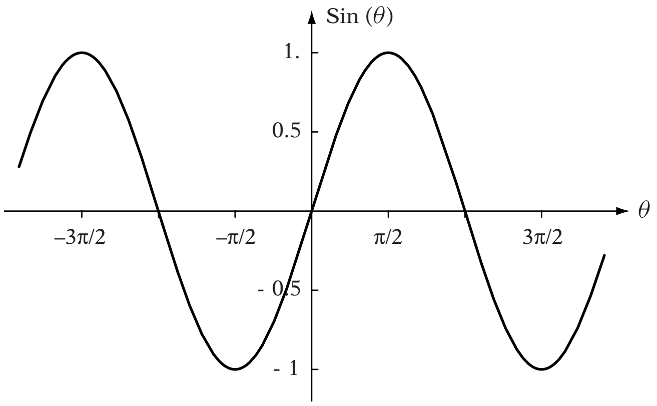


FIGURE A.3 Graph of $\sin\theta$.

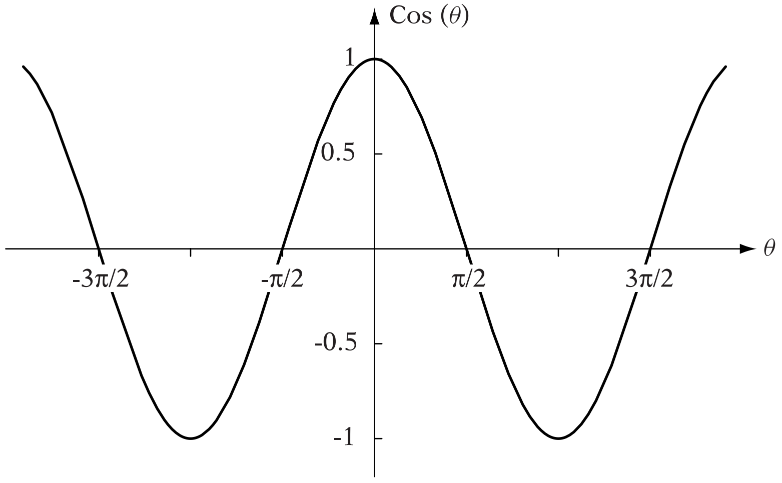


FIGURE A.4 Graph of $\cos\theta$.

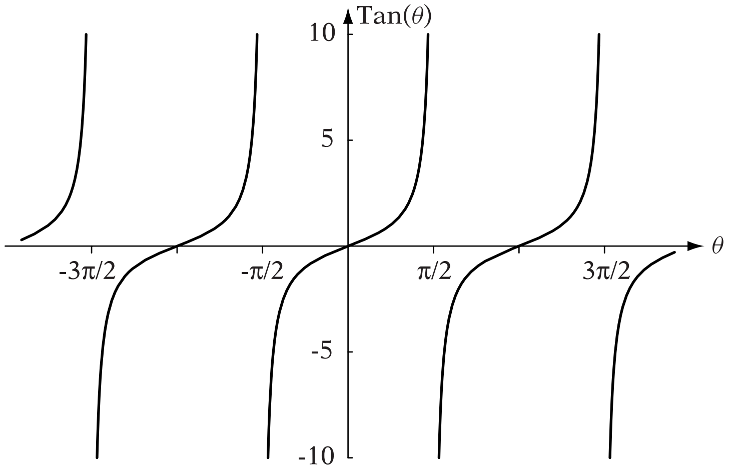


FIGURE A.5 Graph of $\tan\theta$.

for cosine, secant, and cosecant. Tangent and cotangent are periodic with period π: $\tan(x) = \tan(n \cdot \pi + x)$.

A.2 Properties of Triangles

There are three laws that relate angles in a triangle to sides of a triangle, using trigonometric functions. Figure A.9 shows a general triangle with sides of length a, b, and c, and corresponding opposite angles α, β, and γ.

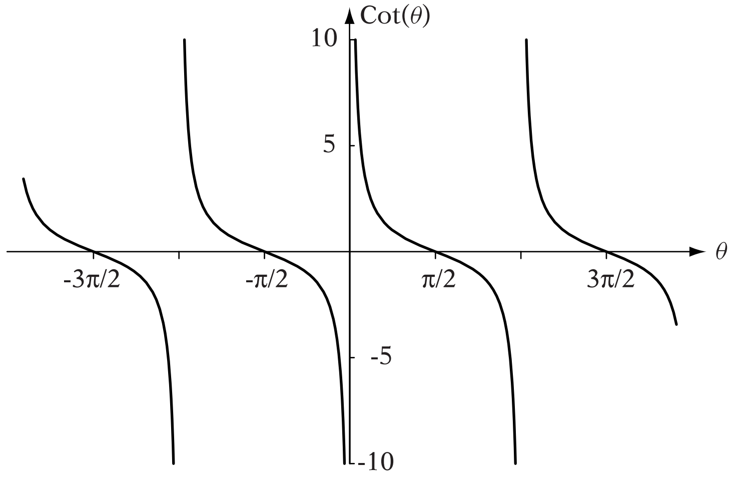


FIGURE A.6 Graph of $\cot\theta$.

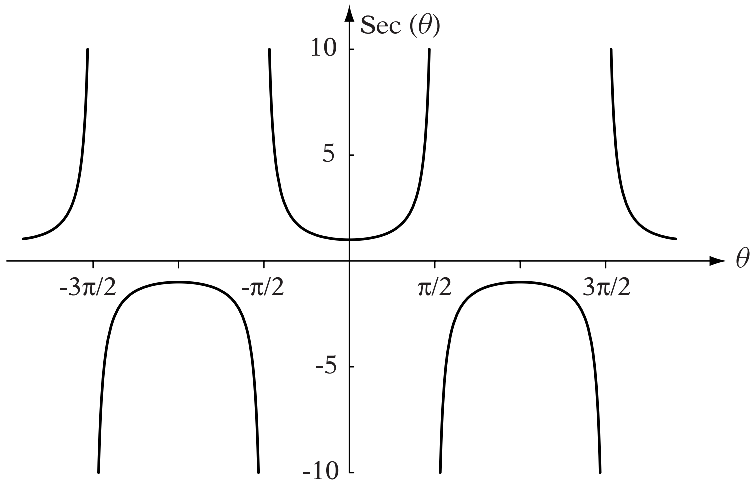


FIGURE A.7 Graph of $\sec\theta$.

The *law of sines* relates angles to their opposing sides as a constant ratio for each pair:

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} \tag{A.1}$$

Recall the Pythagorean theorem:

$$c^2 = a^2 + b^2$$

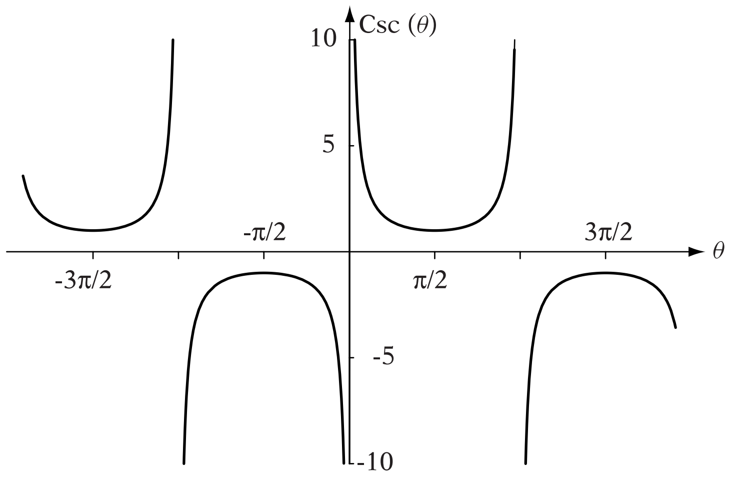


FIGURE A.8 Graph of $\csc\theta$.

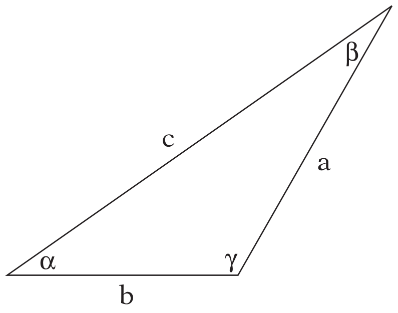


FIGURE A.9 General triangle, with sides and angles labeled.

which relates two sides of a right triangle to the hypotenuse. The *law of cosines* is an extension to this, which can be used to compute the length of a side from the length of two other sides and the angle between them:

$$c^2 = a^2 + b^2 - 2ab\cos\gamma \tag{A.2}$$

Substituting $\pi/2$ for γ produces the specific case of the Pythagorean theorem.

The *law of tangents* relates two angles and their corresponding opposite sides:

$$\frac{a-b}{a+b} = \frac{\tan(\frac{1}{2}(\alpha-\beta))}{\tan(\frac{1}{2}(\alpha+\beta))} \tag{A.3}$$

All of these can be used to construct information about a triangle from partial data.

While not specifically one of the laws, a related set of formulas computes the area of a triangle:

$$\frac{ab\sin\gamma}{2} = \frac{bc\sin\alpha}{2} = \frac{ac\sin\beta}{2} \tag{A.4}$$

A.3 Trigonometric Identities

A.3.1 Pythagorean Identities

Again, from the Pythagorean theorem we know that

$$a^2 + b^2 = c^2$$

where c is the length of the hypotenuse and a and b are the lengths of the other two sides. In the case where the length of the hypotenuse is 1, the length of the other two sides are $\cos\theta$ and $\sin\theta$, so

$$\sin^2\theta + \cos^2\theta = 1 \tag{A.5}$$

where $\sin^2\theta = (\sin\theta)(\sin\theta)$, and similarly for $\cos^2\theta$.

Dividing equation A.5 through by $\cos^2\theta$:

$$\frac{\sin^2\theta}{\cos^2\theta} + \frac{\cos^2\theta}{\cos^2\theta} = \frac{1}{\cos^2\theta}$$

$$\tan^2\theta + 1 = \sec^2\theta \tag{A.6}$$

If we instead divide equation A.5 by $\sin^2\theta$:

$$\frac{\sin^2\theta}{\sin^2\theta} + \frac{\cos^2\theta}{\sin^2\theta} = \frac{1}{\sin^2\theta}$$

$$\cot^2\theta + 1 = \csc^2\theta$$

A.3.2 Complementary Angle

If we consider one acute angle θ in a right triangle, the other acute angle is its complement $\frac{\pi}{2} - \theta$. We can compute trigonometric functions for the complementary angle by changing the sides we use when computing the ratios, for example,

$$\sin\left(\frac{\pi}{2}-\theta\right) = \text{adj/hyp} = \cos(\theta)$$

The complementary angle identities are

$$\cos\theta = \sin\left(\frac{\pi}{2}-\theta\right) \tag{A.7}$$

$$\sin\theta = \cos\left(\frac{\pi}{2}-\theta\right) \tag{A.8}$$

$$\cot\theta = \tan\left(\frac{\pi}{2}-\theta\right) \tag{A.9}$$

$$\tan\theta = \cot\left(\frac{\pi}{2}-\theta\right) \tag{A.10}$$

$$\csc\theta = \sec\left(\frac{\pi}{2}-\theta\right) \tag{A.11}$$

$$\sec\theta = \csc\left(\frac{\pi}{2}-\theta\right) \tag{A.12}$$

A.3.3 Even-Odd

Two of the trigonometric functions, cosine and secant, are symmetric across $\theta = 0$ and are called *even* functions:

$$\cos(-\theta) = \cos\theta \tag{A.13}$$

$$\sec(-\theta) = \sec\theta \tag{A.14}$$

The remainder are antisymmetric across $\theta = 0$ and are called *odd* functions:

$$\sin(-\theta) = -\sin\theta \tag{A.15}$$

$$\csc(-\theta) = -\csc\theta \tag{A.16}$$

$$\tan(-\theta) = -\tan\theta \tag{A.17}$$

$$\cot(-\theta) = -\cot\theta \tag{A.18}$$

A.3.4 Compound Angle

For two angles α and β, the sines of the sum and difference of the angles are, respectively,

$$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta \tag{A.19}$$

$$\sin(\alpha-\beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta \tag{A.20}$$

Similarly, the cosines of the sum and difference of the angles are

$$\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta \tag{A.21}$$

$$\cos(\alpha-\beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta \tag{A.22}$$

These can be combined to create the compound angle formulas for the tangent:

$$\tan(\alpha+\beta) = \frac{\tan\alpha + \tan\beta}{1-\tan\alpha\tan\beta} \tag{A.23}$$

$$\tan(\alpha-\beta) = \frac{\tan\alpha - \tan\beta}{1+\tan\alpha\tan\beta} \tag{A.24}$$

A.3.5 Double Angle

If we substitute the same angle θ for both α and β into the compound angle identities, we get the double angle identities:

$$\sin 2\theta = 2\sin\theta\cos\theta \tag{A.25}$$

$$\cos 2\theta = \cos^2\theta - \sin^2\theta \tag{A.26}$$

The latter can be rewritten using the Pythagorean identity as

$$\cos 2\theta = 1 - 2\sin^2\theta \tag{A.27}$$

$$= 2\cos^2\theta - 1 \tag{A.28}$$

The double angle identity for tangent is

$$\tan 2\theta = \frac{2\tan\theta}{1-\tan^2\theta} \tag{A.29}$$

A.3.6 Half Angle

Equations A.27 and A.28 can be rewritten as

$$\sin^2\alpha = \frac{1-\cos 2\alpha}{2} \tag{A.30}$$

$$\cos^2\alpha = \frac{1+\cos 2\alpha}{2} \tag{A.31}$$

Substituting $\theta/2$ for α and taking the square roots gives

$$\sin\left(\frac{\theta}{2}\right) = \pm\sqrt{\frac{1-\cos\theta}{2}} \tag{A.32}$$

$$\cos\left(\frac{\theta}{2}\right) = \pm\sqrt{\frac{1+\cos\theta}{2}} \tag{A.33}$$

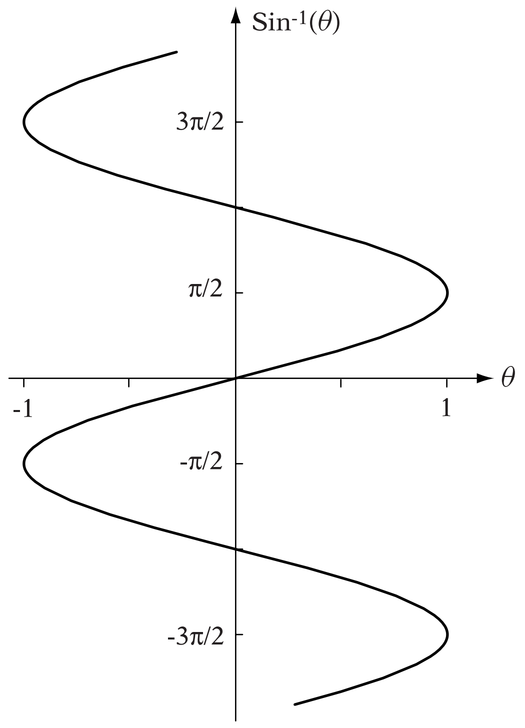


Figure A.10 Graph of arcsin θ.

Note that due to the square root, there are two choices for each identity, positive and negative—the one chosen depends on what quadrant $\theta/2$ is in.

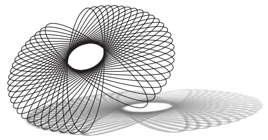
A.4 Inverses

The trigonometric functions invert to multivalued functions because they are periodic. For example, the graph of the inverse $\sin^{-1}\theta$, or *arcsine* can be seen in Figure A.10. Its domain is the interval $[-1, 1]$ and its range is $\mathbb{R}$.

Because of this, it is common to restrict the range of an inverse trignometric function so that it maps only to one value, given a value in the domain. Standard choices for these restrictions are as follows:

Function	**Domain**	**Range**
$\sin^{-1}$	$[-1, 1]$	$[-\pi/2, \pi/2]$
$\cos^{-1}$	$[-1, 1]$	$[0, \pi]$
$\tan^{-1}$	$\mathbb{R}$	$[-\pi/2, \pi/2]$

Appendix B
Calculus Review



B.1 Limits and Continuity

B.1.1 Limits

The expression

$$L = \lim_{x \to a} f(x)$$

is read as, L is the *limit* of a function f as x approaches a given value a. Informally, this represents that as x gets closer to a, $f(x)$ will get closer to L. We can more formally represent the notion of "closeness" to L and a by using the following definition:

> A function $f(x)$ has a limit L at a if given any $\epsilon > 0$ there exists $\delta > 0$ such that $|f(x) - L| < \epsilon$ when $0 < |x - a| < \delta$.

In other words, for each value of ϵ larger than zero, $f(x)$ is less than ϵ away from L for all x sufficiently close to a. The value of δ provides a measure of what "sufficiently close" means.

In many cases, the limit is just the value of the function at a. For example, if we have the function

$$f(x) = x^2 \tag{B.1}$$

then

$$\lim_{x \to a} f(x) = \lim_{x \to a} x^2 = a^2 = f(a)$$

for all values of a. However, consider:

$$g(x) = \frac{x^2 - 1}{x - 1} \tag{B.2}$$

At $x = 1$, the value of $g(x)$ is undefined since the resulting denominator is 0. But if we graph g, as in Figure B.1, it appears that as we get close to 1 the function value gets close to 2. As it happens, 2 is the limit of $g(x)$ as x approaches 1. In this case we can say that while at $x = 1$, the function value is undefined, however:

$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1} = 2$$

Note that there may not necessarily be a limit at a given a. For example, as graphed in Figure B.2, the step function:

$$h(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{B.3}$$

has no limit at 0. In this case we can talk about a right-hand limit (approaching only from the positive direction) or left-hand limit (approaching from

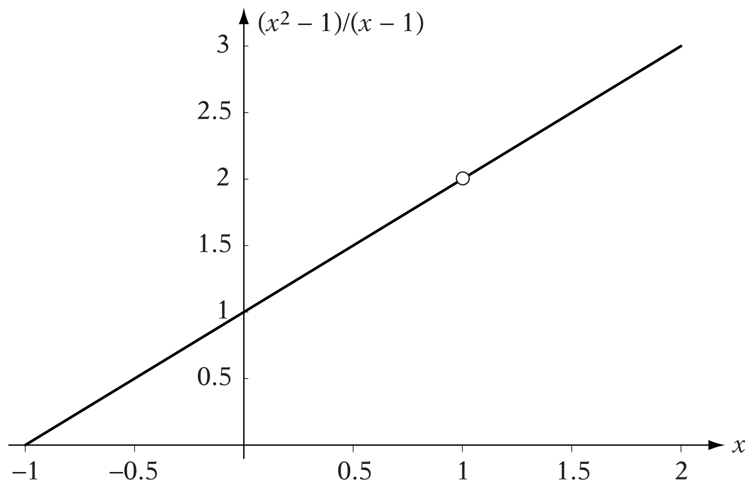


FIGURE B.1 Part of function with discontinuity but valid limit at $x = 1$.

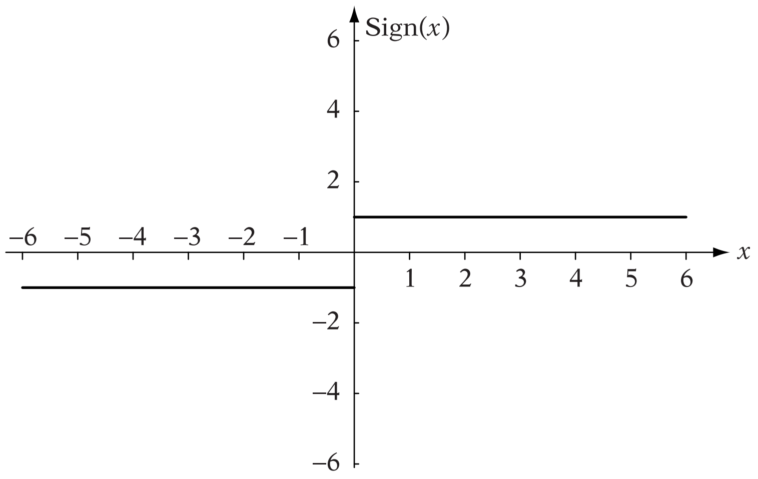


Figure B.2 Function with discontinuity and no two-sided limit at $x = 0$.

the negative direction) or, respectively,

$$\lim_{x \to 0^+} h(x) = 1$$
$$\lim_{x \to 0^-} h(x) = -1$$

B.1.2 Continuity

There are three possibilities with regard to the limit of a function $f(x)$ as x approaches a:

1. $\lim_{x \to a} f(x)$ exists and equals $f(a)$ (e.g., equation B.1)
2. $\lim_{x \to a} f(x)$ exists and does not equal $f(a)$ (e.g., equation B.2)
3. $\lim_{x \to a} f(x)$ does not exist (e.g., equation B.3)

In the first case, we say that f is *continuous* at a. Otherwise, it is *discontinuous* at a.

We also say that a function $f(x)$ is continuous over an interval (a, b) (or $[a, b]$) if it is continuous for every value x in the interval. Informally, we can think of a continuous function as one that we can draw without ever lifting the pen from the page.

B.2 Derivatives

B.2.1 Definition

Suppose we have a function $f(x)$. If we take two points on the curve at time x and time $x+h$, then we can compute the slope of the secant that passes through the points by the function

$$\frac{f(x+h)-f(x)}{h} \tag{B.4}$$

As the value of h approaches 0, the limit (if it exists) approaches the slope of a line tangent to the function at the point x. We can use this to create a new function of x, which computes slopes of $f(x)$ for every value of x where the limit exists:

$$f'(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \tag{B.5}$$

This function is called the *first derivative*, or simply the *derivative*, which we have represented as $f'(x)$. Other common representations are df/dx (also known as *Leibnitz notation*), or when taken with respect to time we place a dot over the function, as $\dot{f}(t)$.

The derivative $f'(x)$ describes the instantaneous rate of change of $f(x)$ at the value x. If $f'(x)$ is positive, $f(x)$ is said to be *increasing* at that point. Correspondingly, if $f'(x)$ is negative, $f(x)$ is said to be *decreasing*. The magnitude of $f'(x)$ describes how great the rate of change is.

A derivative may not necessarily exist for every value in the domain of a function. If this is the case for a particular value x, we say the function is not differentiable at x. If a function is discontinuous, it is not differentiable at the discontinuity. However, even if it is continuous, it may not be possible. For example, Figure B.3, the absolute value function:

$$|x| = \begin{cases} x; & x \geq 0 \\ -x; & x < 0 \end{cases}$$

has no derivative at $x = 0$. This discontinuity represents a sudden change in slope, or if our function represents a path in space, a sudden change in direction.

A function f is differentiable on an open interval (a, b) if it is differentiable at each point in (a, b). It is differentiable on a closed interval $[a, b]$ if it is

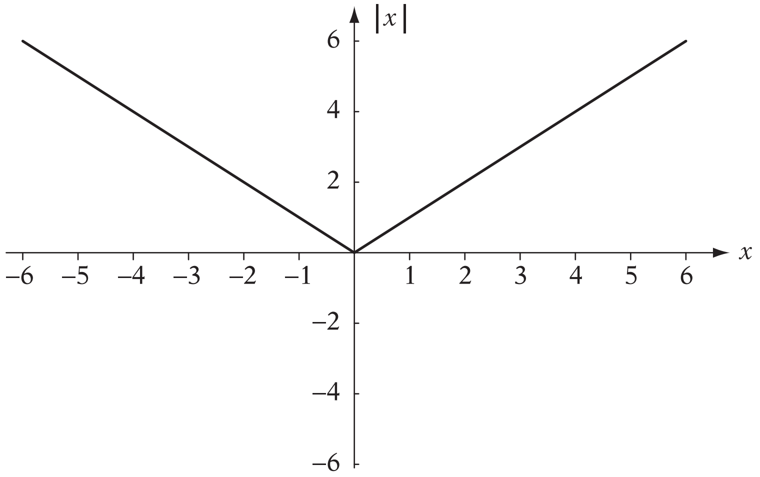


FIGURE B.3 Function that is continuous but has discontinuity in its first derivative at $x = 0$.

differentiable on (a, b) and the limits

$$\lim_{h \to 0^+} \frac{f(a+h) - f(a)}{h}$$

and

$$\lim_{h \to 0^-} \frac{f(b+h) - f(b)}{h}$$

exist. If either limit exists at a point x, then we say that f has a one-sided derivative at x. For example, the absolute value function is differentiable on the intervals $[c, 0)$ and $(0, d]$, where $c < 0$ and $d > 0$, despite not being differentiable at 0.

Since the derivative is itself a function, assuming it is differentiable we can take its derivative to get the *second derivative*, represented by $f''(x)$. If the second derivative is positive, it represents a part of the function which is concave-up (the cross section of a bowl). If it is negative, that part of the function is concave-down (an arch). If the first derivative is continuous but there is a discontinuity in the second derivative, then this represents a sudden change in concavity.

So long as a function and its subsequent derivatives are differentiable, we can continue this process of taking the derivative of derivatives. In general, the nth derivative of a function f at x is represented as $f^{(n)}(x)$, and if such a derivative exists, we say that f is differentiable to order n. If we can keep differentiating in perpetuity, we say that f is infinitely differentiable.

B.2.2 Basic Derivatives

Power of a Variable

The derivative for the power of a variable x, or $f(x) = x^k$ is

$$f'(x) = kx^{k-1} \tag{B.6}$$

By this, the derivative for a linear function $g(x) = x$ is just

$$g'(x) = 1 \cdot x^0 = 1$$

The derivative of a constant term $f(x) = a$ is

$$f'(x) = 0$$

Arithmetic Operations on Functions

The derivative of the sum of two functions is the sum of the derivatives:

$$\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x) \tag{B.7}$$

The derivative of the difference of two functions is the difference of the derivatives:

$$\frac{d}{dx}(f(x) - g(x)) = f'(x) - g'(x) \tag{B.8}$$

The derivative of the product of two functions is

$$\frac{d}{dx}(f(x)g(x)) = f'(x)g(x) + g'(x)f(x) \tag{B.9}$$

The derivative of the quotient of two functions is

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - g'(x)f(x)}{g(x)^2} \tag{B.10}$$

Composite Functions

If we have the composite of two functions

$$h(x) = f(g(x)) = (f \circ g)(x)$$

then the derivative is found by using the *chain rule*. We take the derivative of f with respect to the function g, and multiply that by the derivative of g with respect to the variable x, or

$$h'(x) = f'(g(x))g'(x) \tag{B.11}$$

For example, suppose we have

$$h(x) = (2x^2 + 1)^5$$

We change variables to set $f(u) = u^5$ and $g(x) = 2x^2 + 1$, so that $h(x) = f(g(x))$. Then

$$\begin{aligned} h'(x) = f'(g(x))g'(x) &= 5(2x^2 + 1)^4 \cdot 4x \\ &= 20x(2x^2 + 1)^4 \end{aligned}$$

General Polynomials

If we have a general polynomial

$$f(x) = \sum_{i=0}^{n} a_i x^i$$

we can combine equations B.6, B.7, and B.9 to find its resulting derivative:

$$f'(x) = \sum_{i=0}^{n} a_i i x^{i-1}$$

B.2.3 Derivatives of Transcendental Functions

Trigonometric Functions

The derivatives of the standard trigonometric functions are

$$\begin{aligned} \frac{d}{dx} \sin x &= \cos x \\ \frac{d}{dx} \cos x &= -\sin x \end{aligned}$$

$$\frac{d}{dx}\tan x = \sec^2 x = 1 + \tan^2 x$$
$$\frac{d}{dx}\cot x = -\csc^2 x = -(1 + \cot^2 x)$$
$$\frac{d}{dx}\sec x = \sec x \tan x$$
$$\frac{d}{dx}\csc x = -\csc x \cot x$$

Trigonometric Inverses

The derivatives of the trigonometric inverses are

$$\frac{d}{dx}\sin^{-1} x = \frac{1}{\sqrt{1-x^2}}; \qquad |x| < 1$$
$$\frac{d}{dx}\cos^{-1} x = -\frac{1}{\sqrt{1-x^2}}; \qquad |x| < 1$$
$$\frac{d}{dx}\tan^{-1} x = \frac{1}{1+x^2}$$
$$\frac{d}{dx}\cot^{-1} x = -\frac{1}{1+x^2}$$
$$\frac{d}{dx}\sec^{-1} x = \frac{1}{x\sqrt{x^2-1}}; \qquad |x| > 1$$
$$\frac{d}{dx}\csc^{-1} x = -\frac{1}{x\sqrt{x^2-1}}; \qquad |x| > 1$$

Exponentials and Logarithms

The derivative of the natural exponential function $f(x) = e^x$ is

$$\frac{d}{dx}e^x = e^x$$

That is, the exponential is its own derivative.

The inverse of an exponential function is a logarithmic function. For example, the inverse of the natural exponential function e^x is $\log_e x$, usually written as $\ln x$ and called the natural logarithm. The derivative of the natural logarithm is

$$\frac{d}{dx} \ln x = \frac{1}{x}$$

A general exponential function a^x can be represented in terms of the natural exponential as $a^x = e^{x \ln a}$. So by the Chain rule:

$$\frac{d}{dx} a^x = \ln a \cdot a^x$$

A logarithm with an arbitrary base a can be represented in terms of the natural logarithm as

$$\log_a x = \frac{\ln x}{\ln a}$$

Using this, the derivative is

$$\frac{d}{dx} \log_a x = \frac{1}{x \ln a}$$

B.2.4 Taylor's Series

A *power series centered on* h is an infinite summation of the form

$$\sum_{k=0}^{\infty} a_k (x-h)^k$$

Suppose it is possible to represent a function f as a power series centered on h. Expanding terms, we can then write $f(x)$ as

$$f(x) = a_0 + a_1(x-h) + a_2(x-h)^2 + \cdots$$

To solve for $a_0, a_1, \ldots$, we begin by finding the value at $f(h)$:

$$f(h) = a_0 + a_1(h-h) + a_2(h-h)^2 + \cdots$$

All terms but the first cancel, and so $a_0 = f(h)$. Assuming that f is differentiable at h, we can differentiate both sides and again evaluate at h to get

$$f'(h) = a_1 + 2a_2(h-h) + 3a_3(h-h)^2 \cdots$$

So $a_1 = f'(h)$. Differentiating one more time (again, assuming that it is possible) gives us

$$f''(h) = 2a_2 + 6a_3(h-h) + 12a_4(h-h)^2 \cdots$$

giving $a_2 = f''(h)/2$. Continuing this process gives us a general formula for a_k of

$$a_k = \frac{f^{(k)}(h)}{k!}$$

Assuming that f is infinitely differentiable, the *Taylor series expansion* for f is

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(h)}{k!}(x-h)^k \tag{B.12}$$

The first few terms of this look like

$$f(x) = f(h) + f'(h)(x-h) + \frac{f''(h)}{2}(x-h)^2 + \frac{f'''(h)}{6}(x-h)^3 + \cdots$$

In general, a function f may not be infinitely differentiable, so another form is used. Suppose f is differentiable to degree $n+1$ within an interval I, and h lies within I. Then we can approximate f with p_n, the nth Taylor polynomial

$$f(x) \approx p_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(h)}{k!}(x-h)^k$$

The error of the approximation is given by r_n, the nth Taylor remainder, where

$$r_n(x) = f(x) - p_n(x)$$

It can be proved that for every x in I, there is a value $\xi(x)$ between x and h which allows us to represent $r_n(x)$ as

$$r_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x-h)^{n+1}$$

This is also known as the *Lagrange remainder formula*.

B.3 Integrals

B.3.1 Definition

Given a function $f(x)$, the *indefinite integral* (also known as the *antiderivative*) of $f(x)$ is represented as

$$\int f(x)\,dx$$

The term dx, or differential, represents the fact that we are integrating with respect to the variable x; any other variables will be considered constant. The result of the indefinite integral for $f(x)$ is a function $F(x) + C$, where $F'(x) = f(x)$.

The arbitrary constant C is appended to indicate a possible constant term, the value of which will differentiate to 0. For example, differentiating the functions $f(x) = x^2+x+1$ and $g(x) = x^2+x-12$ produces $f'(x) = g'(x) = 2x+1$. Integrating $2x + 1$ with respect to x gives the result $x^2 + x + C$.

The *definite integral* of a function $f(x)$ across an interval $[a, b]$ is represented as

$$\int_a^b f(x)\,dx$$

We say in this case that we are integrating from a to b. The result of the definite integral is a quantity. In particular, when $f(x) \geq 0$ it equals the area between the curve and the axis represented by the differential—in this case, the x-axis. For example, the following definite integral

$$\int_0^1 x^2\,dx$$

computes the area (also known as the *area under the curve*) shown in Figure B.4. The result is 1/3.

If any of the curve being evaluated is negative along the interval, the area computed by the definite integral between that section of curve and the axis in question is also negative. For example, the following definite integral

$$\int_{-1}^0 x\,dx$$

computes the area shown in Figure B.5. The result of the definite integral is $-1/2$.

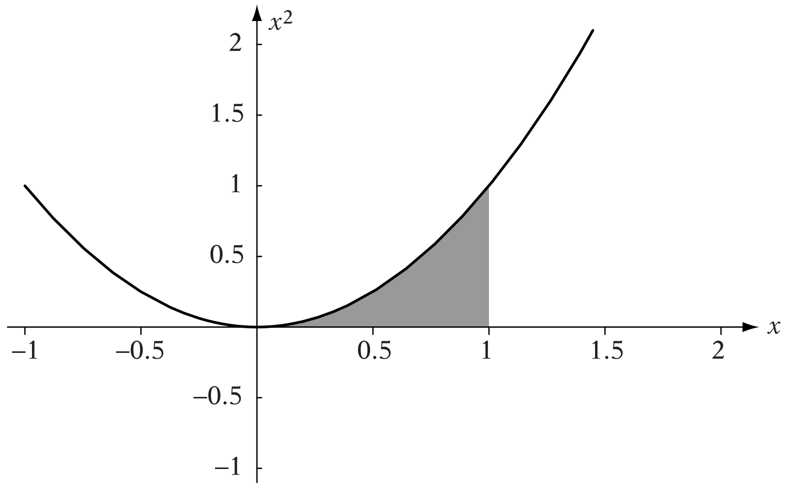


FIGURE B.4 Definite integral returns area between curves and x-axis. The result in this case is $\frac{1}{3}$.

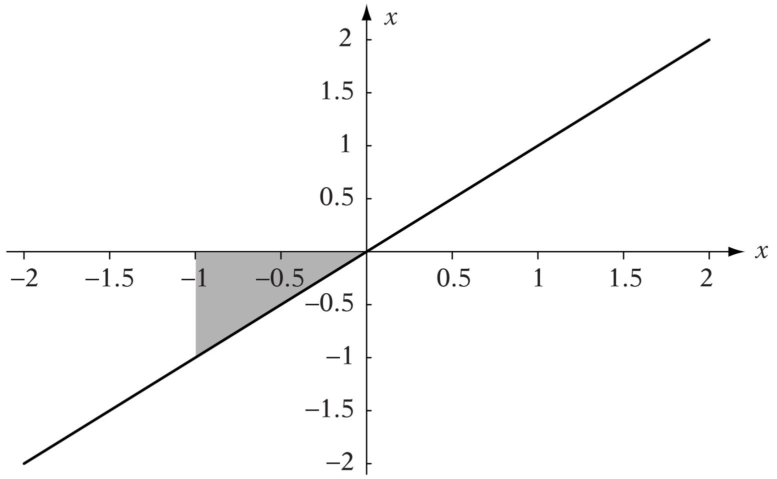


FIGURE B.5 Definite integral of areas of curve below axis produces negative results. The result in this case is $-\frac{1}{2}$.

The *fundamental theorem of calculus* states that a definite integral can be computed from two evaluations of the indefinite integral. More specifically, if $f(x)$ is a continuous function on a closed interval $[a, b]$, and an antiderivative $F(x)$ can be found such that $F'(x) = f(x)$ for all x in $[a, b]$, then

$$\int_a^b f(x)dx = F(x)|_a^b = F(b) - F(a)$$

B.3.2 Evaluating Integrals

Computing an integral for a general function is often not easy, if it can be done at all. Most of the time in games numerical methods are used for evaluation of definite integrals. However, knowing some simple integrals can be useful. For more complex forms the reader is directed to a more detailed calculus reference such as [32].

The integral of the sum of two functions is the sum of the integrals of the functions:

$$\int f(x) + g(x)\ dx = \int f(x)\ dx + \int g(x)\ dx$$

If a function is multiplied by a constant, we can pull the constant out of the integral:

$$\int a \cdot f(x)\ dx = a \int f(x)\ dx$$

If the limits of integration are reversed, then the result is negated:

$$\int_b^a f(x)\ dx = -\int_a^b f(x)\ dx$$

The integral of a polynomial term x^k, where $k \neq -1$, is

$$\int x^k\ dx = \frac{x^{k+1}}{k+1} + C$$

If $k = -1$, then we note that

$$\frac{d}{dx} \ln x = \frac{1}{x}$$

so

$$\int \frac{1}{x}\ dx = \ln x + C$$

Tables of integrals can be found in many places, in particular [122]. A few selected examples are

$$\int \cos x\ dx = \sin x + C$$

$$\int \sin x \; dx = -\cos x + C$$

$$\int \tan x \; dx = -\ln|\cos x| + C$$

$$\int \cot x \; dx = \ln|\sin x| + C$$

$$\int \sec x \; dx = \ln|\sec x + \tan x| + C$$

$$\int \csc x \; dx = -\ln|\csc x + \cot x| + C$$

$$\int e^x \; dx = e^x + C$$

$$\int a^x \; dx = \frac{a^x}{\ln a} + C$$

$$\int \ln x \; dx = x\ln x - x + C$$

$$\int \frac{1}{\sqrt{a^2 - x^2}} \; dx = \sin^{-1}\frac{x}{a} + C$$

$$\int \frac{1}{a^2 + x^2} \; dx = \frac{1}{a}\tan^{-1}\frac{x}{a} + C$$

$$\int \frac{1}{x\sqrt{x^2 - a^2}} \; dx = \frac{1}{a}\sec^{-1}\left|\frac{x}{a}\right| + C$$

B.3.3 Trapezoidal Rule

In many cases it is either inconvenient or impossible to compute the integral directly. For example, the sinc function $f(x) = \sin x/x$ cannot be integrated analytically. In these cases numerical methods are used to approximate the value of a definite integral. One of the simplest such methods is the trapezoidal rule.

Figure B.6 shows a function which we want to integrate. We can approximate the curve between a and b by using a line segment, and the area under the curve is approximated by the area of a trapezoid:

$$\int_a^b f(x)\; dx \approx \frac{1}{2}(b-a)[f(b) + f(a)]$$

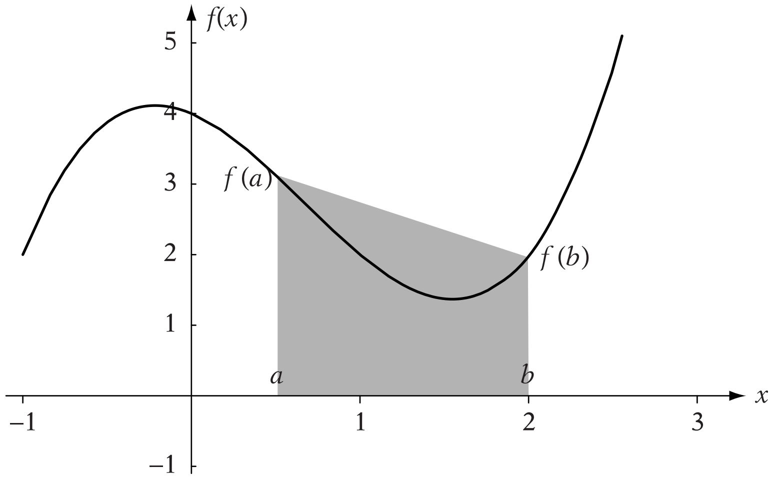


Figure B.6 Approximating the definite integral using a single trapezoid.

We can get a better approximation by slicing the interval into n equally spaced subintervals, computing the areas of the resulting trapezoids and adding them together (Figure B.7). This is equal to

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{b-a}{2n}\sum_{i=0}^{n-1}[f(x_{i+1})+f(x_i)]\\ &= \frac{b-a}{2n}[f(b)+f(a)]+\frac{b-a}{n}\sum_{i=1}^{n-1}f(x_i)\end{aligned}$$

where each $x_i = a + (b-a)i/n$.

B.3.4 Gaussian Quadrature

While the trapezoid rule provides reasonable approximation of a definite integral for little cost, we can get a better approximation using a method called Gaussian quadrature.

The trapezoid rule can be rewritten as a summation of the form

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} c_i f(x_i)$$

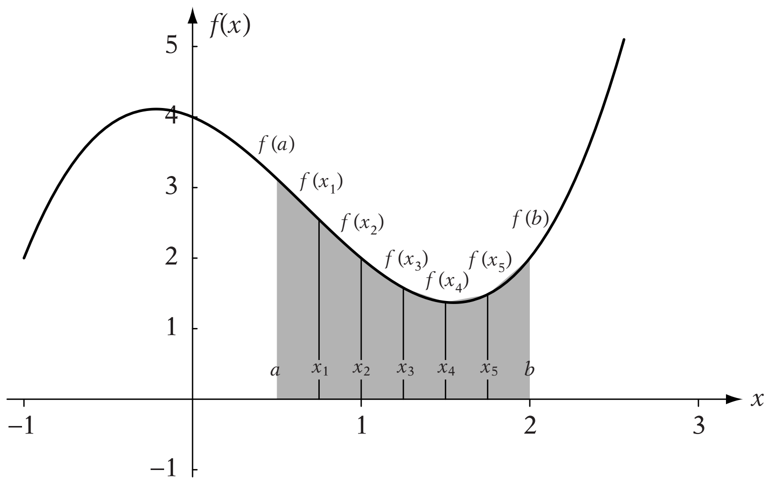


FIGURE B.7 Approximating the definite integral using multiple trapezoids.

where our c_i and x_i are

$$c_i = \begin{cases} (b-a)/2n; & i = 0, i = n \\ (b-a)/n; & 0 < i < n \end{cases}$$

$$x_i = a + (b-a)i/n$$

Gaussian quadrature uses a similar form, except that it uses nonuniform samples and calculates weights to minimize error and get a better approximation. The error is measured relative to a polynomial; using Gaussian quadrature with n samples, we want the exact result when integrating a polynomial P of degree $2n-1$ or less.

It can be shown that for a given value of n and limits of integration of $[-1, 1]$, the values of x_i needed to meet this criteria are the roots of the nth member of a set of polynomials called the Legendre polynomials. The corresponding values of c_i are given by

$$c_i = \int_{-1}^{1} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}\, dx$$

The roots x_i and the associated constants c_i are easily precomputed for a given n. The first few are

n	x_i	c_i
2	$\pm\sqrt{1/3}$	1
3	0	8/9
	$\pm\sqrt{3/5}$	5/9
4	± 0.3399810436	0.6521451549
	± 0.8611363116	0.3478548451
5	0.0000000000	0.5688888889
	± 0.5384693101	0.4786286705
	± 0.9061798459	0.2369268850

Note that using these values is valid only when integrating from −1 to 1. If our integral has a general interval of $[a, b]$, we can use the following to transform it so it can be used with Gaussian quadrature:

$$\int_a^b f(x)\,dx = \int_{-1}^1 f\left(\frac{(b-a)t+b+a}{2}\right)\frac{b-a}{2}\,dt$$

B.4 Space Curves

A parametric curve is a function $Q(t)$ that maps a set of real values (represented by the parameter t) to a set of points. When mapping to $\mathbb{R}^3$, we commonly use a parametric curve broken into three separate functions, one for each coordinate: $Q(t) = (x(t), y(t), z(t))$. This is also known as a *space curve*.

The first derivative of a space curve is found by computing the derivatives of the functions $x(t)$, $y(t)$, and $z(t)$, so $\mathbf{Q}'(t) = (x'(t), y'(t), z'(t))$. The result of $\mathbf{Q}'$ at parameter t is a vector tangent to the curve at location $Q(t)$, instead of a single slope value. The magnitude of the vector represents the speed at which $Q(t)$ changes relative to time; the larger the vector, the faster the position changes. $\mathbf{Q}'$ is also known as the velocity $\mathbf{v}(t)$.

Computing the second derivative of $Q(t)$ is done similarly, by computing the second derivatives of the individual functions x, y, and z: $\mathbf{Q}''(t) = (x'(t), y'(t), z'(t))$. This represents the change in velocity and is also known as acceleration, or $\mathbf{a}(t)$.

If we normalize $\mathbf{Q}'(t)$ at each parameter t, we get the tangent $\mathbf{T}(t)$:

$$\mathbf{T}(t) = \frac{\mathbf{Q}'(t)}{\|\mathbf{Q}'(t)\|}$$

We can also compute the derivative of $\mathbf{T}(t)$ and normalize it to get the normal $\mathbf{N}(t)$:

$$\mathbf{N}(t) = \frac{\mathbf{T}'(t)}{\|\mathbf{T}'(t)\|}$$

Note that this is not the same as the acceleration. While the acceleration's direction may vary relative to the velocity, the result of $\mathbf{N}(t)$ is always perpendicular to $\mathbf{T}(t)$. By taking the cross product of $\mathbf{T}$ and $\mathbf{N}$, we get the binormal $\mathbf{B}(t)$:

$$\mathbf{B}(t) = \mathbf{T}(t) \times \mathbf{N}(t)$$

Using $\mathbf{T}(t)$, $\mathbf{N}(t)$, and $\mathbf{B}(t)$ as an orthonormal basis and $Q(t)$ as the origin, this gives us a coordinate frame for every parameter t, known as the Frenet frame.

As mentioned, $\mathbf{N}(t)$ is not the same as acceleration. The acceleration vector lies in the subspace formed by using $\mathbf{T}$ and $\mathbf{N}$ as basis vectors, or

$$\mathbf{a} = a_T\mathbf{T} + a_N\mathbf{N}$$

where

$$a_T = \frac{d\|\mathbf{v}\|}{dt}$$
$$a_N = \|\mathbf{v}\| \left\| \frac{d\mathbf{T}}{dt} \right\|$$

A parametric curve $Q(t)$ is *smooth* on an interval $[a, b]$ if it has a continuous derivative on $[a, b]$ and $Q'(t) \neq 0$ for all t in (a, b). A parametric curve $Q(t)$ is *piecewise smooth* on an interval $[a, b]$ if it can be broken into a finite number of subintervals, where it is smooth on each subinterval and Q has one-sided derivatives on (a, b).

For a given point P on a smooth curve $Q(t)$, we define a circle with radius ρ and first and second derivative vectors equal to those at P as the osculating circle. The *curvature* κ at P is $1/\rho$. We can also define the curvature of Q as

$$\kappa(t) = \frac{\|\mathbf{T}'(t)\|}{\|\mathbf{Q}'(t)\|} \tag{B.13}$$

The curvature at any point is always nonnegative. The higher the curvature, the more the curve bends at that point; the curvature of a straight line is 0.

We can compute the length $\mathcal{L}$ of a piecewise smooth space curve Q on an interval $[a, b]$ by

$$\mathcal{L} = \int_a^b \|\mathbf{Q}'(t)\| \, dt \tag{B.14}$$

If $Q(t)$ is smooth, we can also define the *arc length* function $s(t)$ as

$$s(t) = \int_a^t \|\mathbf{Q}'(u)\| \, du$$

If $t \geq a$, this measures the length of the curve from a given point $Q(a)$ to a variable point $Q(t)$. If we differentiate both sides with respect to t, we get

$$s'(t) = \|\mathbf{Q}'(t)\| = \|\mathbf{v}(t)\|$$

Since vector length is nonnegative, and we also know that $\mathbf{Q}'(t) \neq 0$ (since Q is smooth), we know that $s(t)$ is strictly increasing and thus invertible to a function $t(s)$. Based on this, we can reparameterize a curve represented by $Q(t)$ by s, by using $Q(t(s))$. This is known as reparameterization by arc length. Rather than mapping a time t to a position on the curve, we can map a length $\mathcal{L}$ to a position on the curve.

It is usually impossible to evaluate the integral in equation B.14, and hence the arc length, directly. Instead the length is approximated by using numerical methods, such as the trapezoid rule or Gaussian quadrature.

Bibliography

[1] AMD. AMD developer support website. *www.amd.com*.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[3] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*. Wiley, New York, 7th edition, 1994.

[4] ARM. Arm Ltd. developer support website. *www.arm.com*.

[5] James Arvo, editor. *Graphics Gems II*. Academic Press, Inc., San Diego, 1991.

[6] ATI. ATI developer support website. *www.ati.com*.

[7] Sheldon Axler. *Linear Algebra Done Right*. Springer-Verlag, New York, 2nd edition, 1997.

[8] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufman, San Francisco, 1987.

[9] J. F. Blinn and M. E. Newell. Clipping using homogeneous coordinates. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 245–251, New York, 1978.

[10] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[11] Jim Blinn. *A Trip Down the Graphics Pipeline*. Morgan Kaufmann, San Francisco, 1996.

[12] Jim Blinn. *Dirty Pixels*. Morgan Kaufmann, San Francisco, 1998.

[13] Jim Blinn. *Notation, Notation, Notation*. Morgan Kaufmann, San Francisco, 2002.

[14] Jonathan Blow. Hacking quaternions. *Game Developer,* March 2002.

[15] W. Boehm. Inserting new knots into B-spline curves. *Computer Aided Design*, 12(4):199–201, 1980.

[16] W. Boehm. On cubics: A survey. *Computer Graphics and Image Processing*, 19:201–226, 1982.

[17] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS, Boston, MA, 5th edition, 1993.

[18] Arthur Cayley. *The Collected Mathematical Papers of Arthur Cayley*. Cambridge University Press, Cambridge, 1889–1897.

[19] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufman, San Francisco, 1993.

[20] T. N. Cornsweet. *Visual Perception*. Academic Press, New York, 1970.

[21] R. Courant and D. Hilbert. *Methods of Mathematical Physics*, Volume One. Wiley, New York, 1989 (reprint).

[22] M. Cyrus and J. Beck. Generalized two- and three-dimensional clipping. *Computers and Graphics*, 3:23–28, 1978.

[23] Mark DeLoura, editor. *Game Programming Gems*. Charles River, Hingham, MA, 2000.

[24] Mark DeLoura, editor. *Game Programming Gems 2*. Charles River, Hingham, MA, 2001.

[25] Tony deRose. Three-dimensional computer graphics: A coordinate-free approach. Technical Report, University of Washington, 1993.

[26] Rene Descartes. *La Geometrie (The Geometry of Rene Descartes)*. Dover Publications, New York, 1954.

[27] David H. Eberly. *3D Game Engine Design*. Morgan Kaufmann, San Francisco, 2001.

[28] David H. Eberly. Personal communication with authors, 2002.

[29] David H. Eberly. Rotation representations and performance issues. Technical Report, Magic Software, 2002.

[30] David H. Eberly. *Game Physics*. Morgan Kaufmann, San Francisco, 2003.

[31] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[32] Robert Ellis and Danny Gulik. *Calculus With Analytic Geometry*. Harcourt, San Diego, CA, 2nd edition, 1982.

[33] Wolfgang Engel, editor. *Direct3D ShaderX*. Wordware, Plano, Texas, 2002.

[34] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, San Francisco, 2004 (forthcoming).

[35] Euclid. *The Elements*. Dover Publications, New York, 1956.

[36] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1992.

[37] Stephen H. Friedberg, Arnold J. Insel, and Lawrence E. Spence. *Linear Algebra*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[38] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, 1980.

[39] Andrew S. Glassner, editor. *Graphics Gems*. Academic Press, San Diego, CA, 1990.

[40] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, Boston, MA, 1989.

[41] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, 1994.

[42] Ronald N. Goldman. Matrices and transformations. In Andrew S. Glassner, editor, *Graphics Gems*, pages 472–475. Academic Press, San Diego, CA, 1990.

[43] Ronald N. Goldman. Some properties of Bézier curves. In Andrew S. Glassner, editor, *Graphics Gems*, pages 472–475. Academic Press, San Diego, CA, 1990.

[44] Ronald N. Goldman. Recovering the data from the transformation matrix. In James Arvo, editor, *Graphics Gems II*, pages 324–331. Academic Press, San Diego, CA, 1991.

[45] Ronald N. Goldman. Decomposing linear and affine transformations. In David Kirk, editor, *Graphics Gems III*, pages 108–116. Academic Press, San Diego, CA, 1992.

[46] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 1993.

[47] Gene H. Golub and James M. Ortega. *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*. Academic Press, Boston, MA, 1992.

[48] S. Gottschalk, M. C. Lin, and D. Manocha. OBBtree: A hierarchical structure for rapid interference detection. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 171–180, 1996.

[49] Jens Gravesen. The length of Bézier curves. In *Graphics Gems V*, pages 199–205. Academic Press, San Diego, CA, 1998.

[50] Gil Gribb and Klaus Hartmann. Fast extraction of viewing frustum planes from the world-view projection matrix, 2001. *www2.ravensoft.com/users/ggribb/plane%20extraction.pdf*.

[51] Brian Guenter and Richard Parent. Computing the arc length of parametric curves. *IEEE Computer Graphics and Applications*, 10(3):72–78, 1990.

[52] P. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, 1990.

[53] Halliday, David and Robert Resnik. *Fundamentals of Physics*. Wiley and Sons, New York, 1981, 2nd edition.

[54] William Hamilton. On quaternions, or on a new system of imaginaries in algebra. *Philosophical Magazine*, 1844–1850. (Available online). *www.maths.soton.ac.uk/EMIS/classics/Hamilton*.

[55] John C. Hart, George K. Francis, and Louis H. Kauffman. Visualizing quaternion rotation. *ACM Transactions on Graphics*, 13(3):256–276, 1994.

[56] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1996.

[57] Paul Heckbert. Texture mapping polygons in perspective. Technical Report, New Institute of Technology, 1983.

[58] Paul Heckbert and Henry Moreton. Interpolation for polygon texture mapping and shading. In David Rogers and Rae Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, New York, 1991.

[59] Paul S. Heckbert, editor. *Graphics Gems IV*. Academic Press, San Diego, CA, 1994.

[60] Chris Hecker. Under the hood/behind the screen: Perspective texture mapping (series). *Game Developer*, 1995–1996.

[61] Chris Hecker. Behind the screen: Physics (series). *Game Developer Magazine*, 1996–1997.

[62] Martin Held. ERIT — a collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[63] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2nd edition, 1996.

[64] Intel. Intel developer support website. *http://developer.intel.com*.

[65] Thomas Jakobson. Advanced character physics. In *Proceedings of Game Developers Conference*, 2001.

[66] Kenneth Joy. On-line geometric modeling notes: Affine combinations, barycentric coordinates and convex combinations. Technical report, University of California, Davis, 2000.

[67] Kenneth Joy. On-line geometric modeling notes: Points and vectors. Technical report, University of California, Davis, 2000.

[68] Kenneth Joy. On-line geometric modeling notes: Vector spaces. Technical report, University of California, Davis, 2000.

[69] William Kahan. Lecture notes on the status of IEEE-754. Postscript file accessible electronically through the Internet at the address *http://cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps*, 1996.

[70] David Kirk, editor. *Graphics Gems III*. Academic Press, San Diego, CA, 1992.

[71] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, 1986.

[72] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River, Hingham, MA, 2002.

[73] Yu-Dong Liang and Brian Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3(1):1–22, 1984.

[74] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE Press, Bellingham, WA, 2002.

[75] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Computer Graphics (SIGGRAPH '03 Proceedings)*, 2003.

[76] Stan Melax. Finding the shortest path quaternion. In Mark DeLoura, editor, *Game Programming Gems*. Charles River, Hingham, MA, 1998.

[77] Microsoft. Direct X SDK. Available for free download from *http://msdn.microsoft.com*.

[78] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.

[79] Tomas Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Natick, MA, 1999.

[80] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York, 1981.

[81] Hubert Nguyen. Casting shadows. *Game Developer*, March 1999.

[82] nVIDIA. nVIDIA developer support website. *http://developer.nvidia.com*.

[83] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, Boston, MA, 3rd edition, 1999.

[84] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, UK, 2000.

[85] Lewis Padgett. Mimsy were the borogroves. Reprinted in *Science Fiction Hall of Fame*, Volume One, Doubleday, NY, 1943.

[86] Alan W. Paeth, editor. *Graphics Gems V*. Academic Press, San Diego, CA, 1995.

[87] Rick Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann, San Francisco, 2002.

[88] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, 1994.

[89] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[90] Charles Poynton. Charles Poynton's color FAQ. *www.poynton.com/*.

[91] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1991.

[92] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*, 2nd edition. Cambridge University Press, New York, 1993.

[93] Anthony Ralston. *A First Course in Numerical Analysis*. McGraw-Hill, New York, 1965.

[94] David F. Rogers. *An Introduction to NURBS: With Historical Perspective*. Morgan Kaufmann, San Francisco, 2000.

[95] David F. Rogers and J. Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, 1990.

[96] Philip J. Schneider and David H. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, San Francisco, 2002.

[97] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, Reading, MA, 2nd edition, 2000.

[98] Ken Shoemake. Animating rotation with quaternion curves. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 245–254, 1985.

[99] Ken Shoemake. Quaternion calculus for animation. In *Math for SIGGRAPH (ACM SIGGRAPH '89 Course Notes 23)*, pages 187–205. 1989.

[100] Ken Shoemake. Quaternions and 4×4 matrices. In James Arvo, editor, *Graphics Gems II*, pages 351–354. Academic Press, San Diego, CA, 1991.

[101] Ken Shoemake. Euler angle conversion. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 222–229. Academic Press, San Diego, CA, 1994.

[102] Ken Shoemake. Polar matrix decomposition. In Paul S. Heckbert, editor, *Graphics Gems IV*, pages 207–221. Academic Press, San Diego, CA, 1994.

[103] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface '92*, pages 258–264, 1992.

[104] William Stallings. *Computer Organization and Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 5th edition, 2000.

[105] Dan Sunday. Distance between Lines and Segments with their Closest Point of Approach. Technical Report, *http://geometryalgorithms.com*, 2001.

[106] I. E. Sutherland. Sketchpad: A man-machine graphical communications system. In *IFIPS Proceedings of the Spring Joint Computer Conference*, 1963.

[107] I. E. Sutherland and G. W. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[108] Dante Treglia, editor. *Game Programming Gems 3*. Charles River, Hingham, MA, 2002.

[109] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, San Francisco, 2003.

[110] James M. Van Verth. Using the covariance matrix for better fitting bounding objects. In Andrew Kirmse, editor, *Game Programming Gems 4*. Charles River, Hingham, MA, 2004.

[111] David R. Warn. Lighting controls for synthetic images. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, 1983.

[112] Alan Watt. *3D Computer Graphics*, 2nd edition. Addison-Wesley, Wokingham, UK, 1993.

[113] Alan Watt and Fabio Policarpo. *3D Games: Real-Time Rendering and Software Technology*, Volume One. Addison-Wesley, Harlow, UK, 2001.

[114] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, Wokingham, UK, 1992.

[115] Eric W. Weisstein. Eric W. Weisstein's World of Mathematics. *http://mathworld.wolfram.com*.

[116] Eric W. Weisstein. Eric W. Weisstein's World of Physics. *http://scienceworld.wolfram.com/physics*.

[117] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *Lecture Notes in Computer Science, New Results and New Trends in Computer Science*, volume 555, pages 359–370. Springer-Verlag, New York, 1991.

[118] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, 1983.

[119] Andrew Witkin and David Baraff. Physically based modelling: Principles and practice. In *ACM SIGGRAPH 2001 Course Notes*, 2001.

[120] George Wohlberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, 1990.

[121] Hansong Zhang. Effective occlusion culling for the interactive display of arbitrary models. Technical Report TR99-027, UNC/Chapel Hill, January, 1998.

[122] Daniel Zwillinger. *CRC Standard Mathematical Tables and Formulae*. CRC Press, Boca Raton, FL, 1995.

Index

Trademarks

About the CD-ROM

Introduction

Many of the concepts in this book are visual, dynamic, or both. While static illustrations are used throughout the book to illuminate some of these concepts, the truly dynamic concepts can be best understood only via experiencing them in an interactive illustration. Computer-based examples serve this purpose quite well.

This book includes a CD-ROM that contains numerous interactive demonstration programs for concepts discussed in the book. The demos are supported on Windows (2000 and XP), MacOS (OS X), and Linux. The main contents of the CD-ROM are:

- Pre-compiled versions of the demos for Windows, ready to run. These are likely to be useful to the widest range of readers of the book, as they are ready to use as supplied, and can be experienced quickly, with book in hand.
- Source for all of the demos, ready to edit and recompile on all platforms. For many students, this is an excellent way to start tinkering with actual graphics, animation and simulation code. The demos can form excellent launching pads for further experimentation.
- Source for the graphics and math libraries used to create the demos. These libraries can form the basis of even more complex graphics applications, especially the low-level mathematics libraries. In addition, the source to these libraries is used as a set of design and implementation examples throughout the book.

Updates

To distribute updates and corrections to this code as well as new demos, a webpage has been established for this book at *www.essentialmath.com*. Please visit this site before using the included CD-ROM to read any important news or updates regarding the CD-ROM that were added following the production of the book's CD-ROM.

Installing the CD-ROM

In order to use the CD-ROM, simply insert the disc into a CD-ROM drive that is mounted on the computer and use the file explorer or command prompt to open the top-level directory of the disc.

Getting Started

There are two files that anyone planning to use the CD-ROM should read prior to copying and using the demos or any of the code. The first of these files is the license information, `LICENSE.PDF`.

This file details the license agreement that all users are bound by when using the demo code. The "grant" clause of this software license agreement ("SLA") are as follows:

The full details may be found in the license file on the CD-ROM.

The second set of files that any user should read are the "read me" files. The general "read me" file, `README_FIRST.TXT` relates information that is pertinent to all users of the code. In addition, there are README files for each of the supported platforms. Put together, these files contain a wide range of information, including:

- Descriptions of supported platforms, hardware, and development tools
- Instructions on how to prepare your computer to run the demos on each of the supported platforms
- Instructions on how to build the engine libraries and demos themselves (on each of the supported platforms)
- Known issues with any of the demos or libraries

The book makes many references in its text to these demos, where appropriate, using the icons described in the introduction to the book. However, there are additional, unreferenced demos that were written after the book text was finalized. These newer demos are available on the CD-ROM, but are not referenced in the text. Please refer to the README_FIRST.TXT file in the root directory of the CD-ROM, as well as the demo directories for each chapter for additional demos not referenced in the book text.

Contents of the CD-ROM

Shared Libraries Directory

`/common`

Contains the source and build configuration files for the support libraries (collectively known as `Iv`) described in the book and used to create the book's demos

`/Includes`

Contains copies of all of the headers from the `Iv` directories listed below

`/IvCollision`

Contains bounding volume classes and intersection methods

`/IvCurves`

Contains position-interpolating curve classes

`/IvEngine`

Contains classes and functions that support interactive application development

`/IvMath`

Contains foundation mathematical classes such as vectors

`/IvScene`

Contains classes implementing a basic hierarchical scene graph

`/IvUtility`

Contains low-level system support code such as file I/O

`/Libs`

Contains the libraries built by each of the `Iv` library directories

Examples Directory

`/Examples`

Contains all of the demo applications referenced by the book text

`/Ch03-Xforms`

Contains the demos for Chapter 3: Affine Transformations

```
/Transforms-01-Interaction
/Transforms-02-Centered
/Transforms-03-Separate
/Transforms-04-Tank
/Transforms-05-SceneGraph
```

`/Ch05-Viewing`

Contains the demos for Chapter 5: Viewing and Projection

```
/Viewing-01-LookAt
/Viewing-02-Rotation
/Viewing-03-Perspective
/Viewing-04-Stereo
/Viewing-05-Orthographic
/Viewing-06-Oblique
/Viewing-07-Clipping
/Viewing-08-Picking
```

Contents of the CD-ROM (*continued*)

/Ch06-GeometryColoring

Contains the demos for Chapter 6: Geometry, Shading, and Texturing

/Geometry-01-BasicSphere
/Geometry-02-IndexedGeom
/Geometry-03-BasicShading
/Geometry-04-BasicTexturing
/Geometry-05-TextureWrapping

/Ch07-Lighting

Contains the demos for Chapter 7: Lighting

/Lighting-01-DistanceAttenuation
/Lighting-02-Spotlight
/Lighting-03-Components
/Lighting-04-Edges
/Lighting-05-Textures

/Ch08-Raster

Contains the demos for Chapter 8: Rasterization

/Raster-01-DepthBuffering
/Raster-02-TextureFilter
/Raster-03-Mipmapping
/Raster-04-Blending

/Ch09-Curves

Contains the demos for Chapter 9: Curves

/Curves-01-Linear
/Curves-02-Lagrange
/Curves-03-Hermite
/Curves-04-AutoHermite
/Curves-05-Catmull
/Curves-06-Bezier
/Curves-07-B-Spline
/Curves-08-SpeedControl
/Curves-09-CameraControl

/Ch10-Orientation

Contains the demos for Chapter 10: Orientation Representation

/Orientation-01-Euler
/Orientation-02-Transform
/Orientation-03-LerpSlerp

/Ch11-Collision

Contains the demos for Chapter 11: Intersection Testing

/Collision-01-Hierarchy
/Collision-02-SweepPrune

/Ch12-Simulation

Contains the demos for Chapter 12: Rigid Body Dynamics

/Simulation-01-Force
/Simulation-02-Torque
/Simulation-03-Tank
/Simulation-04-LinCollision
/Simulation-05-AngCollision

/gl

Contains the GLUT headers and library files that are needed to build the Iv libraries and demo applications

/Win32System

Contains Windows runtime DLL(s) required by the GLUT system

Glossary of Notation

Scalars

$\mathbb{W}, \mathbb{Z}, \mathbb{R}$	whole numbers, integers, real numbers
$[a, b]$,(a, b)	closed interval, open interval
$\lvert a\rvert$, $\lfloor a \rfloor$, $\lceil a \rceil$	absolute value, floor, ceiling
$\min(a, ..., b)$, $\max(a, ..., b)$	minimum of a set of scalars, maximum of a set of scalars

Vectors, Points, and Lines

$\mathbb{R}^2$,$\mathbb{R}^3$,$\mathbb{R}^n$	pairs of real numbers, triples of real numbers, n-tuples of real numbers
$\mathbf{v}$, $\mathbf{v}^T$, $\mathbf{0}$, v_i	vector, vector transpose, zero vector, vector element
$\{\mathbf{i}, \mathbf{j}\}$, $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$	standard basis in $\mathbb{R}^2$, standard basis in $\mathbb{R}^3$
$\mathbf{u} \cdot \mathbf{v}$, $\mathbf{u} \times \mathbf{v}$, $\mathbf{u} \otimes \mathbf{v}$	dot product, cross product, tensor product
$\lVert\mathbf{v}\rVert$, $\hat{\mathbf{v}}$, $\mathbf{v}^{\perp}$	vector length or norm, unit length vector, vector perpendicular to $\mathbf{v}$
$\text{proj}_{\mathbf{w}}\mathbf{v}$	projection of vector $\mathbf{v}$ on vector $\mathbf{w}$
P, $\overline{PQ}$, $\text{dist}(P, Q)$	point, line segment, distance between two points
$L(t)$, $P(u, v)$, $Q(u)$	parameterized line, parameterized plane, parameterized curve

Matrices and Transformations

$\mathcal{T}$, $\mathcal{T}^{-1}$, $\mathcal{T} \circ \mathcal{S}$	transformation, inverse transformation, transformation composition
$\mathbf{M}$, $\mathbf{M}^{-1}$, $\mathbf{M}^T$, $\mathbf{I}$	matrix, matrix inverse, matrix transpose, identity matrix
$m_{i,j}$ or $(\mathbf{M})_{i,j}$	matrix element at row i and column j
$\tilde{\mathbf{w}}$	skew-symmetric matrix representing cross product by $\mathbf{w}$

$\det(\mathbf{M})$ or $\lvert\mathbf{M}\rvert$, $\mathbf{M}^{\text{adj}}$	matrix determinant, adjoint matrix
$R\mathbf{v}$, $\boldsymbol{\Omega}$	rotation of vector **v**, orientation
$\mathbf{v}_{\lvert\lvert}$, $\mathbf{v}_{\perp}$	part of **v** parallel to rotation, part of vector **v** orthogonal to rotation
(x, y, z, w), $\mathbb{R}P^3$	homogeneous point, homogeneous space

Functions and Calculus

$f(x)$, $f'(x)$, $f''(x)$	function, first derivative, second derivative
$\frac{dy}{dx}$, $\dot{y}$	first derivative with respect to x, first derivative with respect to time
$\frac{\partial y}{\partial x}$	partial derivative of y with respect to x
$\sum_{i=a}^{b}$, $\prod_{i=a}^{b}$, $\int$, $\int_a^b$	summation, product, indefinite integral, definite integral
C^0, C^1, C^2, G^1	positional, tangential, curvature, and geometric continuity

Orientation

$\mathbf{q}$, $\mathbf{q}^{-1}$, $\mathbf{qvq}^{-1}$	quaternion, inverse quaternion, rotation of vector **v** by quaternion

Simulation

X, $\mathbf{v}$, $\mathbf{a}$, $\mathbf{F}$, $\mathbf{P}$, m	position, velocity, acceleration, force, linear momentum, mass
ω, τ, $\mathbf{L}$, $\mathbf{J}$	angular velocity, torque, angular momentum, inertial tensor matrix