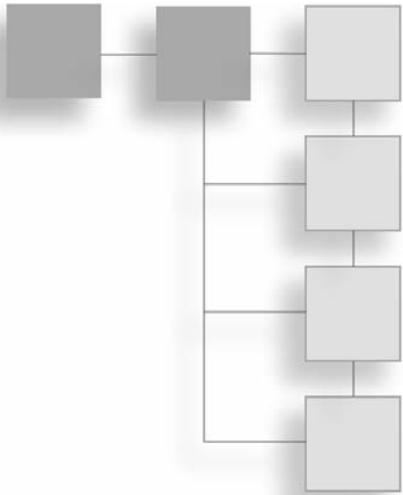


Mathematics and Physics for Programmers

Second Edition

John P. Flynt and Danny Kodicek



MATHEMATICS AND PHYSICS FOR PROGRAMMERS

SECOND EDITION

John Patrick Flynt
Danny Kodicek

Course Technology PTR
A part of Cengage Learning



Australia, Brazil, Japan, Korea, Mexico, Singapore, Spain, United Kingdom, United States

Mathematics and Physics for Programmers, Second Edition

John Patrick Flynt
Danny Kodicek

Publisher and General Manager,
Course Technology PTR:
Stacy L. Hiquet

Associate Director of Marketing:
Sarah Panella

Manager of Editorial Services:
Heather Talbot

Marketing Manager:
Mark Hughes

Senior Acquisitions Editor:
Emi Smith

Project Editor/Copy Editor:
Kim Benbow

Technical Reviewer: Ben Sherman

Interior Layout Tech:
Judy Littlefield

Cover Designer: Mike Tanamachi

Indexer: Valerie Haynes Perry

Proofreader: Sue Boshers

© 2012 Course Technology, a part of Cengage Learning.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance,
contact us at **Cengage Learning Customer &
Sales Support, 1-800-354-9706**

For permission to use material from this text or product,
submit all requests online at **cengage.com/permissions**.
Further permissions questions can be e-mailed to
permissionrequest@cengage.com.

All trademarks are the property of their respective owners.

All images © Cengage Learning unless otherwise noted.

Library of Congress Control Number: 2010933074

ISBN-13: 978-1-4354-5733-1

ISBN-10: 1-4354-5733-1

eISBN-10: 1-4354-5783-8

Course Technology, a part of Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**.

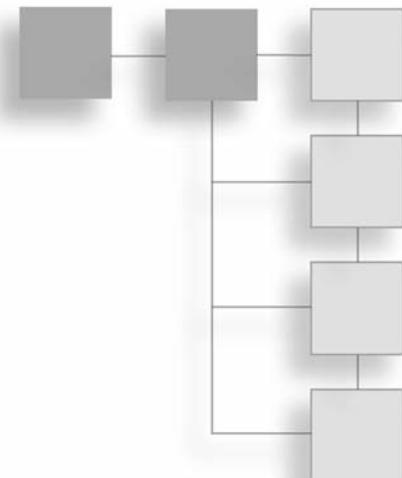
Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**.
Visit our corporate Web site at **cengage.com**.

*The first edition of this book was dedicated to
Tony Revel and David Hepburne-Scott, two teachers sadly missed.*

The second edition of this book is dedicated to its readers.

This page intentionally left blank



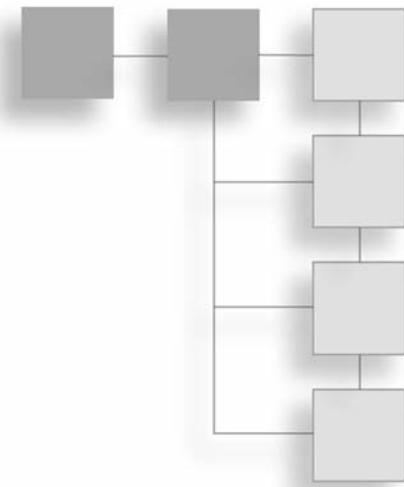
ACKNOWLEDGMENTS

The First Edition

For the production of the book, Danny Kodicek conveys his gratitude to Dave Pallai, Jennifer Blaney, Ania Wieckowski, Bryan Davidson, and Meg Dunkerley. Likewise, to James Newton and Pim, and Robert Tweed. For web-based tutorials, Jim Andrews, Jonas Beckeman, Thomas Higgins, Martin Kloss, Lucas Meijer, Allen Partridge, Ben Pitt (RobotDuck), Simeon Rice (NoiseCrime), Barry Swan, and Alex (Zav) Zavatone. For mathematical and other grounding, Michael Davies, John Field, Jim Cogan, Jonathan Katz, and Tony Revell. For physics grounding, David Hepburne-Scott. And to his children Anna and Matthew for letting him snooze on the sofa in the morning, to his wife Catherine for putting up with this huge endeavor for so long and her unfailing support, and to his mother and father for their encouragement and advice.

The Second Edition

For the production of the book, John Flynt conveys his gratitude to Danny Kodicek, Emi Smith, Kim Benbow, Lisa Staib, Judith Littlefield, and others at Cengage Learning. For technical concerns, to Ben Sherman, Toby Jones, John Hart, Kevin Claver, and Rob Johnson. For a general perspective, thanks go to Jim Curry and the prospect of doing more math. And to his family, Marcia, for being amazing, and to Amy and Adrian, for being equally amazing. And to Java for being Java and guarding the door: *Woof-Woof!*



ABOUT THE AUTHORS

John Flynt is the author and co-author of several books on programming, math, software engineering, and game development. Among his books are *Software Engineering for Game Developers*; *In the Mind of a Game*; *Perl Power!: The Comprehensive Guide*; *Java Programming for the Absolute Beginner, Second Edition*; *Beginning Math Concepts for Game Developers*; and *Java ME Game Programming, Second Edition*.

Danny Kodicek works with Sunflower Learning in the UK developing science simulations and tools for schools. His software has been translated into over fifteen languages and is sold worldwide. As a freelancer, his previous clients include the BBC and the Royal Air Force, and he was co-creator of the award-winning TimeHunt website.

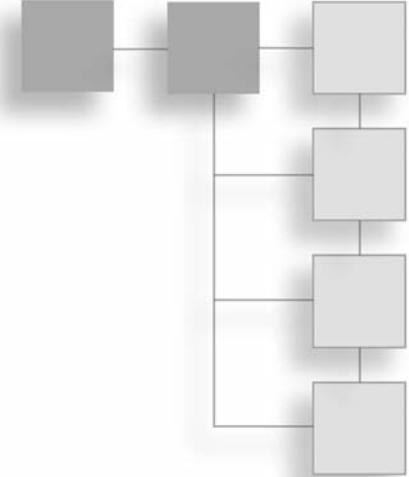


TABLE OF CONTENTS

Part I	Essential Topics in Mathematics	1
Chapter 1	Numbers	3
	Overview.....	3
	Numbers as Written.....	4
	How Computers Represent Numbers	9
	Exercises.....	20
	Summary.....	21
	You Should Now Know	21
Chapter 2	Arithmetic.....	23
	Overview.....	23
	Fractions	24
	Proportions, Ratios, and Percentages.....	35
	Exponentials	43
	Logarithms.....	47
	Exercises.....	51
	Summary.....	52
	You Should Now Know	52
Chapter 3	Algebra	53
	Overview.....	53
	Basic Algebra.....	53
	Working with Equations	59
	Factoring and Solving Quadratic Equations	64
	Functions and Graphs	76

Exercises.....	87
Summary.....	88
You Should Now Know	88
Chapter 4 Geometry and Trigonometry	89
Overview.....	89
Angles	89
Triangles	94
Calculations with Triangles	105
Rotations and Reflections	110
Exercises.....	118
Summary.....	118
You Should Now Know	119
Chapter 5 Vectors	121
Overview.....	121
Getting from Here to There.....	121
Vector Motion	130
Vector Calculations	138
Matrices	147
Exercises.....	155
Summary.....	155
You Should Now Know	156
Chapter 6 Calculus.....	157
Overview.....	157
Differentiation and Integration.....	157
Differential Equations	170
Approximation Methods	173
Exercises.....	179
Summary.....	180
You Should Now Know	180
Part II Essential Topics in Physics	181
Chapter 7 Acceleration, Mass, and Energy	183
Overview.....	183
Ballistics	183
Mass and Momentum.....	190
Energy.....	192
Exercises.....	196
Summary.....	197
You Should Now Know	198

Chapter 8	Detecting Collisions Between Simple Shapes	199
	Overview.....	199
	Ground Rules.....	200
	When Circles Collide	201
	When Squares Collide.....	211
	When Ellipses Collide.....	220
	When Things Collide.....	226
	Exercises.....	227
	Summary.....	228
	You Should Now Know	228
Chapter 9	Collision Resolution	229
	Overview.....	229
	Resolving a Single Collision.....	230
	Multiple Collisions	238
	Exercises.....	242
	Summary.....	243
	You Should Now Know	243
Chapter 10	Detecting Collisions Between Complex Shapes.....	245
	Overview.....	245
	Problems with Complex Shapes.....	246
	Some Reasonable Problems	259
	Built-In Solutions.....	273
	Exercises.....	274
	Summary.....	274
	You Should Now Know	275
Chapter 11	A Simple Pool Game	277
	Overview.....	277
	Primary Elements of a Simulation	278
	Taking a Shot	285
	Exercises.....	293
	Summary.....	294
	You Should Now Know	294
Part III	Complex Motion	295
Chapter 12	Force and Newton's Laws	297
	Overview.....	297
	Force	298
	Gravity	302
	Rockets and Satellites	305

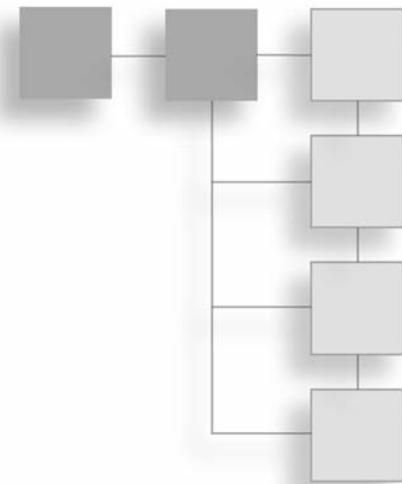
Exercise	308
Summary	308
You Should Now Know	308
Chapter 13 Angular Motion	309
Overview.....	309
The Physics of a Lever.....	309
Spin	316
Spinning Collisions.....	319
Spin Applied to the Pool Game	334
Exercises.....	335
Summary.....	336
You Should Now Know	336
Chapter 14 Friction.....	337
Overview.....	337
How Friction Works	338
Friction and Angular Motion	343
Exercise	347
Summary.....	347
You Should Now Know	348
Chapter 15 Strings, Pulleys, and Conveyor Belts.....	349
Overview.....	349
Pulling Things Around.....	349
Continuous Momentum	355
Exercise	359
Summary.....	359
You Should Now Know	359
Chapter 16 Oscillations	361
Overview.....	361
Springs	362
Simple Harmonic Motion	364
Damped Harmonic Motion	369
Complications of Springs.....	374
Calculating Spring Motion	376
Waves.....	382
Exercise	390
Summary.....	390
You Should Now Know	390

Part IV	3-D Mathematics.....	391
Chapter 17	3-D Geometry.....	393
	Overview.....	393
	3-D Vectors	393
	Rendering	402
	Casting a Ray.....	409
	Exercise	414
	Summary.....	414
	You Should Now Know	414
Chapter 18	Transforms.....	415
	Overview.....	415
	Describing Locations in Space	415
	Applying Transforms.....	421
	Exercise	430
	Summary.....	430
	You Should Now Know	430
Chapter 19	3-D Collision Detection.....	431
	Overview.....	431
	Colliding Worlds	432
	Colliding Footballs	435
	Colliding Boxes.....	437
	Colliding Cans	442
	Varieties of Collisions.....	448
	Resolving Collisions in Three Dimensions	449
	Exercise	449
	Summary.....	450
	You Should Now Know	450
Chapter 20	Lighting and Textures.....	451
	Overview.....	451
	Light.....	451
	Materials.....	455
	Shading.....	468
	Exercises.....	470
	Summary.....	471
	You Should Now Know	471

Chapter 21	Modeling Techniques	473
Overview.....	473	
Mathematical 3-D Modeling	474	
Animated Surfaces	483	
Bone Animations	487	
Exercises.....	493	
Summary.....	494	
You Should Now Know	494	
Part V	Game Algorithms.....	495
Chapter 22	Speeding Things Up	497
Overview.....	497	
Cheap and Expensive Calculations	497	
Pseudo-Physics	504	
Culling.....	507	
Exercises.....	514	
Summary.....	514	
You Should Now Know	514	
Chapter 23	Tile-Based Games.....	515
Overview.....	515	
Generating a Game from Bits	516	
Advanced Tiling	523	
Exercises.....	529	
Summary.....	529	
You Should Now Know	529	
Chapter 24	Mazes.....	531
Overview.....	531	
Classifying Mazes	531	
Creating Mazes	537	
Navigating Within Mazes.....	547	
Exercises.....	556	
Summary.....	556	
You Should Now Know	557	

Chapter 25	Game Theory and Artificial Intelligence	559
Overview.....	559	
Introduction to Game Theory	560	
Tactical AI	572	
Top-Down AI	576	
Bottom-Up AI	580	
Exercises.....	586	
Summary.....	587	
You Should Now Know	587	
Chapter 26	Search Techniques.....	589
Overview.....	589	
Problem Solving	590	
Case Study	594	
Genetic Algorithms.....	598	
Exercises.....	605	
Summary.....	605	
You Should Now Know	606	
Appendix A	Glossary of Mathematical Terms	607
Appendix B	Code References	629
Appendix C	The Greek Alphabet	633
Appendix D	Learning Resources.....	635
Appendix E	Answers to Exercises.....	639
Index		647

This page intentionally left blank



INTRODUCTION

Who Is This Book For?

Math and Physics for Programmers, Second Edition, is for many different people. If you thumb through the pages, you can gain a sense of the topics covered and the presentation of the material. The discussions of mathematics, physics, biology, and other topics are meant to help you understand how such topics can assist you as a programmer who is interested in developing games. But even if your interests don't center on games, it is hoped that you can find that when the material is presented in the context of games, it is more fun and easier to understand.

Generally, to benefit from this book, it will help a great deal if you know how to program in a language such as Java, JavaScript, Python, ActionScript, Lingo, C/C++, Java, Visual Basic, Perl, or C#. These are some of the languages, in any event, the authors have played with over the years. This can even extend to the scripting languages for MATLAB, Mathematica, or Maple. Any grounding will do, since the code examples in the book are presented as pseudocode, allowing you to put what you learn to work in contexts that are familiar to you.

This book is for you if you are curious about how mathematics can be applied to programming projects. While the focus of the book is on the use of mathematics in game contexts, it remains that what applies to games applies to a multitude of other things as well. Generally, to prime the pump a little, have you considered how a program based on neural nets or genetics might be used for a game? How about maze theory and tessellations? And what about linear algebra and understanding how vectors and matrices can be used to create and manipulate graphical objects? On the other hand, how do you get from

basic arithmetic learned in middle or high school to a level of math that allows you to work with 3-D objects in sophisticated programming contexts? Or suppose you went through a lot of this at one point and are just looking for a refresher? If any of these questions catches your attention, then it is likely that this book is worth considering.

What Is This Book About?

Obviously, as the title suggests, this book is about math and physics for programming. It is not a textbook, however. Nor does it provide step-by-step procedures (in most cases) for doing the thing it presents. Certainly, in many instances, you can copy the pseudocode examples and alter them for your own purposes. But the book has been written to be read, to serve as a guide to what can be done with mathematics and physics in the context of game development. From this, it is hoped that you will gain a sense of the scope and depth of the possibilities.

The topics covered in the book fall into four broad categories:

- **Essential topics.** You are likely to encounter these topics in everyday life. They are the foundation concepts underlying the whole of mathematics. Most of these are covered in Part I of the book. They concern the basic principles of numbers, algebra, and geometry. These are also dealt with in the first chapters of Parts II, III, and IV, where the basics of physics and 3-D mathematics are discussed. As much as possible, essential topics are covered in depth, using derivations and code samples.
- **Advanced topics.** These are drawn from complex areas of mathematics and physics. While you are less likely to meet them on a day-to-day basis, they are necessary for a complete understanding of the subject and to solve other problems. Such topics include calculus (Chapter 6), complex physics concepts (Part III), and detailed 3-D math, especially the workings of 3-D renderers (Part IV). Advanced topics are covered in a more discursive fashion, with explanations of the principles and terminology, a few examples, and some derivations of important results.
- **Applied topics.** With applied topics, you will look at how essential concepts can be applied in more complex situations. Portions of the book that involve applied topics include most of Part II, as well as one or two chapters in Parts III and IV. In these chapters, you'll look at a few particular examples of the topic, exploring them in depth, with detailed code samples and derivations of mathematical results, but we will not attempt to cover the whole topic. However, the examples given should help you tackle similar problems yourself. You will try a range of different methods for achieving the same goals.

- **Extension topics.** These are more advanced or obscure areas of the subject, particularly those that can be used to illustrate some broader principles. Every effort is made to present things in a simple and engaging manner. Most of these crop up in Part V, although some of Parts III and IV fit into this category, too. The aim is to make you comfortable with the essential concepts and terminology of the topic without worrying too much about the fine detail. A full explanation of any one of these topics would require another book—or many books—but after reading through the chapters, you should at least know what it is you are trying to achieve and where to look for the solutions. These chapters have fewer code examples and equations, with more of an emphasis on verbal explanations and images.

Naturally, these categories overlap to some extent, and every chapter has an element of all of them, but this explanation should make the overall philosophy of the book clearer. The principal aim of the book is to leave you understanding the mathematical principles and concepts involved in programming—not to know everything there is to know on any one subject. It is hoped that by reading this book you will understand the concepts well enough to know what to look for when you encounter some new problem. In this respect, it is a book of mathematical techniques, physical principles, examples of the math in action, and general concepts and terminology.

Each chapter ends with one or more programming exercises for you to try yourself. You'll find many examples of the code on the book's companion website. For the most part, the examples are available in chapter files, but you'll find the longer examples under their specific names. Everything has been placed and indexed in HTML files, so you should find accessing examples to be an easy, intuitive activity.

How Should I Use This Book?

Math and Physics for Programmers is presented as a cumulative sequence. Most chapters make reference to topics covered earlier. In this respect, it is worthwhile to at least quickly skim over the whole book. Even if you are fairly confident with basic mathematics and physics, there are likely to be some parts of the early sections that you are unsure about.

Mathematics is a practical subject, and unless you have worked through a problem, you are unlikely to really understand it. Toward this end, it is worthwhile to at least take on the challenges listed in the exercises for the chapters. In fact, don't stop with the exercises alone. Think about how you might generalize and incorporate your work into a larger project. Can you think of a game that uses the principles you have learned? Can you think

how to improve an earlier game by incorporating these new elements? As a programmer, you are in a much better position to really come to grips with the principles behind what you are learning.

Finally, don't get scared. People are unreasonably afraid of some of these subjects even though they are perfectly able to tackle far more complex tasks (like arguing about international politics down at the pub). If you don't understand something, first read it again. If it still doesn't make sense, then think about why—is there some terminology you don't understand? See if the glossary refreshes your memory, or find an earlier chapter where the concept is explained. Try a practical example, draw a little diagram, or see if you can rephrase it. If all else fails, skip over it, and see if a later explanation throws some light on it. But don't do that too often—that is how problems start.

About the Pseudocode

The aim of this book is to introduce you to essential mathematics, not to teach good programming style. You don't need to get bogged down in the specifics of programming languages, implementation, or user interfaces, or (except very briefly) in graphics or sound. So you will find few references to variables, objects, arrays, or sprites. Instead, you will look at particular functions and algorithms, which perform particular tasks. Because they are entirely mathematical, they should be easy to implement in whatever language you are using and can be treated as black boxes that just do what you want. This does not mean that the same mathematics could not be implemented better or faster by integrating it with the rest of your code or by making use of shortcuts in the language you use. In fact, in the vast majority of cases, this is very possible.

To avoid worrying about programming style, all code examples in this book are presented as pseudocode. This term means different things to different people, but in this book it designates fairly detailed versions of the code, written in a human-readable language. The pseudocode is essentially a programming language written for a human interpreter rather than a computer.

Occasionally, some parts of the code are left out in order to concentrate on the important points. The sections that are left out are indicated by a double bar (||). There are also frequent comments, preceded by a double slash (//).

Most of the code examples on the book's companion website were composed in Lingo. This is a fairly flexible, high-level, uncompiled language that employs untyped variables and can be used in both an object-oriented and a procedural style.

Whenever it is possible, we avoid specifics about variables, arrays, and so on, but they are sometimes necessary. The book uses the Lingo convention of saying the first element of an array (list) A is A[1], rather than A[0] as most other languages have it.

How to Locate the Code Samples

As noted already, the code samples are in many instances pseudocode, which means, of course, that you cannot expect to copy and paste them into a compiler and have them run.

Most of the code samples are presented in HTML documents. They are sorted in two ways:

- **In chapter folders.** If the code pertains to themes presented in one chapter only, then you will find it by accessing one or more HTML files stored in a folder with the same name as the chapter. In many instances, you will find a file named the same as a chapter, such as Chapter 7.html. In other instances, however, you will find differently named files, and you can use your browser to inspect these for programs that pertain to the chapter.
- **From a main index.** A second approach is likely to prove more useful and is recommended above the first. In a folder called Indexed Programs, access the index.html file. This file will help you to access specific chapter themes and weave together the variant themes that are presented throughout the book. In this way, you can both reinforce your understanding of specific themes presented in the book and gain a sense of how the themes are tied together.

Since the index.html file might prove a little scant when you first open it, here is a list of the links you will see and where they will connect you:

- **Castlib tileScroller.** Connects you with a number of programs relating to the later chapters of the book, which deal with tiling (Chapter 23), searches, and optimization.
- **Castlib math.** Gives you access to programs relating to the first 18 chapters of the book but with specific reference to the math operations in these chapters.
- **Castlib mazes.** Links you to programs relating to Chapters 23, 24, and 25, which deal with mazes and AI.
- **Castlib pool.** Provides access to files that relate to the pool game, which is first presented in Chapter 11 but is referred to in discussions in several chapters that follow.
- **Castlib simple3D.** Connects you to programs that relate to 3-D graphics. This discussion formally begins in Chapter 17, but you then return to it in different contexts for all subsequent chapters.

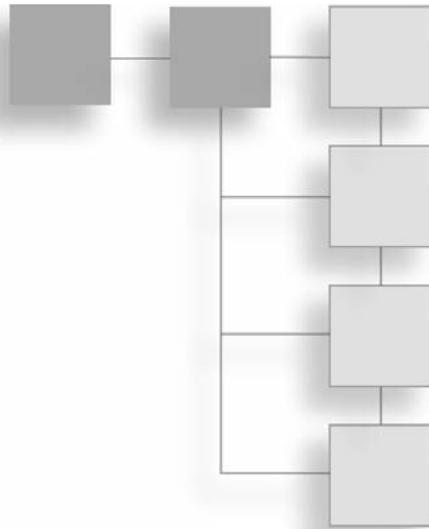
Note that the term Castlib is used to group items. A *cast* refers to a group of objects in a game that are related to common tasks. In this context, use of the term is more or less a way of saying that sample code is grouped according to themes. Get used to opening the index.html file as you read. After a time, you will acquaint yourself with the vocabulary items and visit code samples from various chapters as you go.

Errors, Omissions, and Comments

In a book of this length and scope, it's inevitable that errors have been missed. Most of these, it is hoped, will be minor, but it's possible that some will be more dramatic. In either case, comments can be submitted to the publisher and apologies, from the start, are offered.

Companion Website Downloads

You may download the companion code sample files from www.courseptr.com/downloads. Please note that you will be redirected to our Cengage Learning site.



PART I

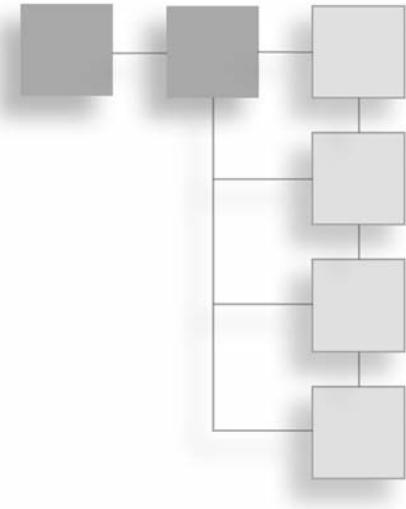
ESSENTIAL TOPICS IN MATHEMATICS

In this first part of the book, you will cover the most basic and essential elements of mathematics. In all likelihood, much of the material here will already be familiar to you. However, covering this material is important so that you can deal with the more complex concepts presented in later chapters.

This page intentionally left blank

CHAPTER 1

NUMBERS



In This Chapter

- Overview
- Numbers as Written
- How Computers Represent Numbers
- Standards and Computed Numbers
- Common Functions
- Rounding Errors and Performance

Overview

Most people would say that numbers are the foundation of mathematics. This is quite a modern view, because for most of the past two thousand years, students began with geometry. On the other hand, numbers are certainly the foundation of computers, and a thorough understanding of numbers and how they work is vital to programming.

In this respect, then, the journey begins with a look at the way computers represent numbers and what you can do with numbers using a computer. The goal here is to think about what a number is, especially the distinction between the number and the way you can represent it in writing or electronically.

Note

Code samples pertaining to this chapter can be found in the Chapter 1 folder on the companion website for this book, which is located at www.courseptr.com/downloads.

Numbers as Written

Numbers were not discovered in one fell swoop. The set of numbers commonly recognized today (positive counting numbers, negative counting numbers, zero, integers, rational numbers, irrational numbers, and imaginary numbers) required thousands of years to be discovered. While not reviewing the history of their discovery, a few points concerning their general definitions are worthwhile.

Integers, Rationals, and Irrationals

Mathematicians divide the world of numbers into several subsets. The first numbers to be recognized were the *counting numbers*, or *natural numbers*, which are represented by the symbol \mathbb{N} . These are the numbers that children usually learn (1, 2, 3, 4, . . .). In addition, there is the number zero (0). Each of these can represent a distinct number of “things.” When you add the negative numbers, you have the set of *integers*, represented by the symbol \mathbb{Z} . With the integers, however, a new domain opens out, for negative numbers are brought into the picture.

What, then, is a negative number? You can think about negative numbers in terms of a transaction. Consider a situation in which you have 5 marbles. You are playing with another person, Anne, who has 4 marbles. When you give Anne 2 marbles, she gains 2 marbles, but it is also the case that she is giving you -2 marbles. Either way, you add 2 marbles to one side and subtract 2 from the other.

Given this beginning, you can extend activities to consider multiplication. Think about negative numbers in terms of an operation of “the other way around.” Then you can see why two negative numbers multiplied together give a positive number. You give Anne 2 marbles, which is “the other way round” to her giving you 2 marbles. Since she gave you -2 marbles, if you do this -2 times, it’s the other way round from her doing it 2 times, so it’s the other way round from -4 , giving you $+4$.

If this sounds confusing, the confusion explains why it took centuries for people to accept the idea of negative numbers. The problem was so difficult that *fractions*, the quotients of two integers, were accepted sooner. First there were the fractions of 1, such as $\frac{1}{2}$, and $\frac{1}{3}$. These are in many cases the *reciprocals* of the natural numbers, as when $\frac{2}{1}$ becomes $\frac{1}{2}$. After the reciprocals, other simple fractions arose. These were fractions between 0 and 1, such as $\frac{2}{3}$. Then came the *vulgar fractions*, fractions greater than 1, such as $\frac{5}{4}$. At the end of the line, you have the complete set of *rational numbers*, symbolized by \mathbb{Q} , which includes both positive and negative fractions, as well as the integers. The integers are included with the rational numbers, because they are viewed as a special kind of fraction that has a denominator of 1.

Note

The *quotient* of two numbers is what you get when you divide one number by another. When one integer is divided by another, the result is a fraction, where the first (top) number is called the numerator and the second (bottom) the denominator. Sometimes, quotient means just the integer part of a division, but this will be covered later on. Similarly, the terms *sum*, *difference*, and *product* refer to, respectively, the result of adding, subtracting, and multiplying two numbers.

Irrational and Real Numbers

Beyond the rational numbers, you have the *irrational* numbers. Such numbers cannot be expressed by the quotient of two integers. For a long time it was thought that such numbers did not exist, but Pythagoras proved that at least one number, the square root of 2, was irrational. Since those days (roughly 550 B.C.E.), the field has expanded. Mathematicians now acknowledge that there are fundamentally *more* irrational numbers than rational ones.

The rational and irrational numbers together form the set of *real* numbers, represented by the symbol \mathbb{R} . In addition, however, there are other kinds of numbers, such as the *complex numbers*. These numbers include a multiple of the imaginary number i , which represents the square root of -1 . While these numbers are beyond the scope of this chapter, they are dealt with in Chapter 18, which discusses quaternions.

Note

The square root of a number n , is represented mathematically by \sqrt{n} . In a programming language, it is usually represented by a function, such as `sqrt(n)`. The square root of a number n is a number m , such that $m \times m = n$. When you multiply a number by itself in this way, the result is called the *square* of the number, denoted by n^2 , $n^{\wedge}2$, or `power(n, 2)`. A square is a special case of a *power* or an *exponent*. As long as p is positive, when you take a number n to the power p , written n^p , or $n^{\wedge}p$, this means to multiply the number n by itself p times. Because two negative numbers multiplied together give a positive number, every positive number has two square roots, one positive and one negative, while negative numbers have no real square root. (What happens when p is negative will be dealt with later on.)

The Number as a String of Digits

Numbers can be represented many different ways. Consider the number 5. You can represent it geometrically, as a pentagon, but you can also represent it as a set of 5 dots. You can present it physically, as 5 beads on an abacus. Still another (and common) approach is to use the symbol “5,” which is defined to mean what you want it to mean. Each of these representations has its own advantage. For example, if you use beads, to perform operations such as adding and subtracting, you simply count beads back and forth. On the other hand, when you get to larger numbers involving many beads, using symbols has an advantage.

As it is, it is not possible to have a unique symbol for each number. Instead, you use a limited set symbols, combining them to represent a given number, no matter how big. You do this by means of a *base* system. To create a base system, you choose a number b as a base. Then to represent a number n in the base b , you employ a recursive algorithm. Here is a pseudocode example of how this happens:

```

function numberToBaseString(Number, Base)
    if Number is less than Base then set Output to String (Number)
    otherwise
        || Find the remainder Rem when you divide Number by Base
        set Output to Rem
        set ReducedNumber to (Number - Rem) / Base
        set RestOfString to numberToBaseString (ReducedNumber, Base)
        append RestOfString to the front of Output
    end if
    return Output
end function

```

Note

In the `numberToBaseString ()` function, the remainder, when dividing n by m , is the smallest number r such that for some integer a , $m \times a + r = n$. Accordingly, the remainder when dividing 7 by 3 is 1, because $7 = 3 \times 2 + 1$. In the next chapter, a close look will be taken at how to find the remainder when dividing one number by another.

In the `numberToBaseString ()` function, recursion is used. As the name implies, this function converts a number into a string. The function takes two arguments, `Number` and `Base`. If you supply 354 as the argument for `Number` and 10 for the argument for `Base`, the remainder, when 354 is divided by 10, is 4. This is the number of “units” in 354. The number 4, as the remainder, is written into the output. Next, the remainder is subtracted, resulting in 350, and this result is divided, once again, by 10. By the definition of a remainder, the division is exact, and the answer is 35. This result, 35, is fed into the algorithm again. The remainder is 5, which is the number of “tens” in 354. With another iteration, 3 results, the number of “hundreds.” Putting everything back together, a string is generated, “354,” which is the number first introduced as an argument. While all of this might sound unnecessary, it is important to keep in mind that its purpose is to explore recursively the difference between the number 354 and its representation in the base 10. (Appendix A offers additional discussion of this topic.)

If you perform the operation in reverse, using a string as an argument, you find the value of the number. The `baseStringToValue()` function shows how this is so:

```
function baseStringToValue(DigitString, Base)
    if DigitString is empty then set Output to 0
    otherwise
        set Output to the last digit of DigitString
        set RemainingString to all but the last digit of DigitString
        set ValueOfRemainingString to
            baseStringToValue (RemainingString, Base))
        add Base * ValueOfRemainingString to Output
    end if
    return output
end function
```

When you write a number in base notation, you designate a slot, or base position, for each digit in the number. Consider the number 2631, base 10. The first number, 1, is given by the base to the zero power (which is 1) times the number. The second number, 30, is given by the base to first power (10) times the number (3). The third number, 600, is given by the base to the second power (100), again times the number (6). The process goes on. Rendered as a sequence of additions, you arrive at an expression of this form: $2 \times 1000 + 6 \times 100 + 3 \times 10 + 1 \times 1$.

Decimal, Binary, and Hexadecimal

There's nothing special about using 10 as a base, but since computers use base 2, base 10 might not be the best choice to use as an example in the current context. Still, base 10 has its virtues, one of which is that humans have 10 fingers to count with. Because 10 is a multiple of 2 and of 5, it's fairly easy to determine whether a number written in decimal notation is divisible by these two numbers. This is an advantage. On the other hand (no pun intended), it's much harder to tell if it's divisible by 4 or 7. However, there are some fairly simple tests for divisibility by 3 or 9, and it's also easy to divide a number written in base 10 by 2 or 5.

Note

Consider that a number is divisible by 3 if and only if the sum of its digits in base 10 is divisible by 3.

What if you use some other base? Consider, for example, base 12. This base is familiar, in part, because it has been extensively used for currency and measuring, resulting in the familiar system of feet and inches. It makes it easy to divide numbers by 2, 3, 4, or 6. Another is base 60, used in clocks, along with 12 hours. Still another is base 360, used in measuring angles. However, if the last two are to be used properly, they require a lot of individual symbols. The Babylonians used this base, but apparently responding to the complexity of the undertaking, they subdivided its values using a base 10 system. Although very logical, base 12 never caught on except in limited ways.

Base 2, or *binary*, on the other hand, turns out to be useful in many ways and has found many advocates. In base 2, you represent numbers by powers of two. Using the base positions discussed earlier, the number 11 in base 2 is written as 1011, or $1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$. As with base 10, as long as you work with integers, representing numbers in base 2 can be done uniquely. Another feature of base 2 is that while you need far more digits to represent any given number, you only need two symbols to do it. For example, 1 is 1, 3 is 11, and 5 is 101. As people learn in a computer architecture course, this way of relating numbers allows you to speak of 1 as “on” and 0 as “off,” and this is the basic (binary) language of digital circuits.

Base 2 requires two symbols, but another base common in computer operations requires 16. This is hexadecimal, which uses the standard decimal symbols 0–9, added to six others, the letters A–F, which stand for 10–15. In hexadecimal, F1A represents $15 \times 256 + 1 \times 16 + 10 \times 1$, or 3866. Converting between binary and hexadecimal is quite simple, since each hexadecimal symbol can be translated directly into four binary symbols: F1A translates to 1111 0001 1010.

How Computers Represent Numbers

Given that the binary system is at the heart of how numbers are represented in a computer, and this book is about mathematics for programming, it is beneficial to examine in greater detail how a computer uses binary numbers.

Representing Integers

Because binary can be conveniently represented by two symbols, 1 or 0, or *on* and *off*, it’s an ideal system for computers. As has been said many times, a computer thinks only in numbers, and numbers are represented by sets of “switches.” Each switch represents one bit of information, and each bit can be either “on” or “off.” (The word *bit* is a shortened term for the expression “binary digit.”) With eight such switches (bits), you can represent any number from 0 to 255. With 32 switches (bits), you can represent any number up to 4294967295. With 64 switches (bits), you can represent any number up to 18,446,744,073,709,551,615. This is also known as a “quad word.”

The computer's native environment is designed to perform calculations very fast with any number using base 2, and this is relative to the number of bits of the processor. A 64-bit machine can handle far bigger calculations than a 32-bit one. Not all bits are used, however. Usually, one bit is set aside to represent whether the number is positive or negative. As a result, a 32-bit machine processes numbers from -2147483647 to 2147483647 , instead only a positive range of numbers up to 4294967295 .

Since performing operations like addition and multiplication is so easy, binary computations are especially fast. Here is an algorithm that adds two numbers represented as strings in binary:

```
function addBinaryStrings(b1, b2)
    //pad out the smaller number with leading
    //zeroes so they are the same length
    set Output to ""
    set CarryDigit to 0
    repeat with Index = the length of b1 down to 1
        set k1 and k2 to the Index'th characters of b1 and b2
        if k1 = k2 then
            // the digits are the same, so write the current carry digit
            set WriteDigit to CarryDigit
            set CarryDigit to k1
            // if k1 and k2 are 1 then you are going to be carrying a digit
            // if they are 0 then you won't.
        otherwise
            // the digits are different,
            // so write the opposite of the current carry digit
            set WriteDigit to NOT(CarryDigit)
            // leave the carry digit alone as it will be unchanged
        end if
        append WriteDigit to the front of Output
    end repeat
    if CarryDigit = 1 then append 1 to the front of Output
    return output
end function
```

This algorithm is worth examining because the operations it presents are the key to how computers work. In binary, you have $01 + 01 = 10$, and $01 + 10 = 11$. Speed is augmented because each can be added according to whether digits are the same or different (either 0 or 1). It is not necessary to worry about their values within a large range of values. Since it's easy to implement in the computer's hardware, manipulating integers is a relatively simple operation for a computer. Fundamental to this process are the logical operators AND, OR, and NOT, which combine binary (or *Boolean*) digits together in different ways.

If two values, A and B, are binary digits, as Figure 1.1 illustrates, the AND, OR, and NOT switches result in three primary ways of controlling a computer's logic. With A AND B, the resulting bit is 1 if and only if both A and B are 1. With A OR B, the resulting bit is 1 if A or B is 1, or if both are 1. With the NOT operation, the result is 1 only if the bit is set to 0.

AND

A	B	Result
1	0	0
0	1	0
1	1	1
0	0	0

OR

A	B	Result
0	1	1
1	0	1
1	1	1
0	0	0

NOT

Bit	Result
0	1
1	0

Figure 1.1

Fundamental Boolean operations on bits.

Multiplication in binary is also straightforward because it combines two simple operations, multiplying by 2 and adding. Just as you can multiply by 10 in decimal, you can multiply by 2 in binary by shifting the digits of the number by one place. Consider, for example, that 6 in binary is 110. If the bits are shifted one place to the left, 12 results, for 12 in binary is 1100.

Representing Rational and Irrational Numbers

Thus far, the focus has been on integers, but base notation can be used to represent non-integers, also. The essential activity in this respect is to count down or up from 1. To accomplish this, a *marker* is used. The marker is referred to as the *radix point*. The radix point establishes the point at which to start counting. To work with decimal numbers, the radix point is called the *decimal point*, and the first column after the decimal point represents tenths, the second is hundredths, and so on. In binary, the first column after the radix point represents halves, the second quarters, and so on. Given this scheme of things, the number $1\frac{3}{8}$ is represented in binary as 1.011 ($1 + \frac{1}{4} + \frac{1}{8}$).

But there is a problem with this system, which is that not all fractions can be represented this way. In decimal, it is troublesome representing $\frac{1}{3}$, because this fraction cannot be expressed as a sum of tenths, hundredths, and so on. It turns out to be the limit of the infinite series $\frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \dots$. You express this by writing $\frac{1}{3}$ in decimal as $0.333\dots$.

Note

The term “limit of the series” has a precise and important meaning in mathematics, as is reviewed in the study of calculus. However, it should be clear what it means even in the context of numbers and arithmetic. The key is that the more terms of the series you take, the closer the sum approaches $\frac{1}{3}$, without ever increasing beyond it.

In binary, the situation is even worse. You cannot express any fraction whose denominator is divisible by any number other than 2. Consider, for example, that in decimal, $\frac{1}{5}$ is exactly 0.2. In binary, however, it is the infinitely repeating number, $0.001100110011\dots$. Because such numbers have no definitive termination point, computers have trouble with them.

There are two ways to get around this problem. One is to forget radix points and instead to represent the fraction in terms of its constituent integers, the numerator and denominator. Then every rational number can be represented exactly, without any troublesome repeating digits. Unfortunately, there are disadvantages to this as well. If you add $\frac{1}{2}$ and $\frac{1}{3}$, the answer is expressed as follows: $\frac{5}{6}$. $\frac{5}{6} + \frac{1}{7} = \frac{41}{42}$. $\frac{41}{42} + \frac{1}{11} = \frac{493}{462}$. This is known as an *incompatible fraction*. With such a fraction, the denominators have no common factor (covered more extensively in Chapter 2), so as you work with the fraction, you end up increasing the complexity of the denominator. As a result, it is easy to go above the computer’s maximum size for integers. Even if the maximum is not reached, the computer slows down as it carries out its calculations with increasingly bulky numbers. Except in very special cases, then, approaching fractions in this way is not a sensible move.

Another problem with this approach is the existence of the irrational numbers, which can't be expressed exactly as a fraction. Instead, you are forced to find an approximation. To arrive at an approximation, computers use the base system to represent non-integers, and round the number to the nearest digit.

There can also be problems with this, however. Consider what occurs if a *fixed-point representation* of numbers is used. With this approach, a fixed set of digits (eight, for example) is used after the radix point. With only a limited number of bits to play with for each number, each bit added to the end means one less bit to use at the top. As a result, you are limiting how large and how small numbers can be. Allowing the radix point to be altered changes this situation, and this is what most programming languages do. This alternative is called *floating-point representation*, similar to what is called *scientific notation*.

In scientific notation, a number is represented by giving the first “significant” (non-zero) digit, and then as many digits as required after that. Then you give an indication of where to place the radix. For example, the number 201.49 could be represented by (20149; 3). In fact, it is represented as “2.0149 e2.” By convention, you start with the radix after the first digit and count from there. You can count forward or backward (for example, 2.0149 e-3 is 0.0020149). The index (e3) number is called the *exponent*.

Standards and Computed Numbers

Floating-point numbers use a system similar to scientific notation. There are a number of standard formats. One has been established by the Institute of Electrical and Electronics Engineers (IEEE), an organization that has created many standard formats that are used commonly by computer manufacturers. These standards are important for creating a common language between different hardware.

As established in the IEEE standards, if you are working with 32 bits, you use 1 bit for the sign (positive or negative) and 8 bits for the position of the radix point, from -127 (relative to a value of zero) to +127. This gives a total value of 255. The value of the actual number can then go as high as a little under 2^{128} . The remaining 23 bits represent the *mantissa*, or actual digits of the number. Rather ingeniously, because the first significant digit of a binary number is always 1, there is no need to include this digit in the computer's representation of the number. It is just implicitly there, which provides one more digit of precision. The actual digits of the floating-point representation are correctly called the *significand*, although the word *mantissa* is used for this, too.

Note

It's important to distinguish between the *size* of the number, which is given by the exponent, and the *precision* of the number, which is determined by the number of bits in the significand. Although you can represent very large numbers, they are no more precise than the small ones. For example, it is impossible to distinguish between 987874651253 and 987874651254 using a 32-bit floating-point number.

To explore how numbers are processed by a computer, consider the actions that are taken with two numbers.

The value 10.5

Consider these operations with the value 10.5.

- To represent the value 10.5, you first translate it into binary: 1010.1.
- In scientific notation, this is 1.0101 e3.
- Translate the 3 into a binary number and add 127 (this value is called the bias—it allows us to represent numbers from -127 to 127 rather than 0 to 255), to get 10000010 for the exponent bits.
- Pad out the 1.0101 with zeroes to give 24 digits of the mantissa:
10101000000000000000000000.
- Ignore the first 1, which you can take as read, leaving a 23-bit significand: 010100000000000000000000.
- And the final bit is zero, representing a positive number rather than a negative one.

The value -1 e35

Consider these operations with the value -1 e35.

- To represent the value given in decimal by -1 e35.
- In binary scientific notation, this is -1.10101001010110100101101... e-117.
- Translate the -117 into binary and add 127 to get 1010 for the exponent, which you pad out with zeroes to get 8 bits: 00001010.
- The significand is the first 23 bits after the decimal point:
10101001010110100101101.
- And the final bit is one, representing a negative number.

Considerations

In the operations just examined, the only problem with omitting the initial 1 of the mantissa is that it apparently leaves no way to represent a floating-point value of exactly zero. Fortunately, one extra trick can be added, which is to say that when the exponent part of the number is zero, the leading 1 is not to be assumed. This also means that numbers smaller than 2^{-127} can be represented as long as it is accepted that these numbers will be less precise than usual. Using this trick of denormalization, numbers as low as 2^{-149} can be represented.

As you might expect, zero is represented by the floating-point number given by a zero significand and a zero exponent. The positive/negative bit can be either value. There are also a number of other special cases, which represent *overflow* and *underflow*, as well as the value NaN (“Not a Number”). The value NaN is sometimes returned when you attempt to do something that has no valid answer. An example of this is trying to find the square root of -1 .

How this pans out can be shown by considering the value given in decimal by 2×10^{-40} . Here are the main features of the operations:

- In binary scientific notation, 2×10^{-40} is represented as 1.00010110110000100110001 $\times 10^{-132}$.
- Because -132 is less than -127 , it is not possible to represent this number with the usual precision. But if you shift the exponent up by five places, you get a value of 0.0000100010110110000100110001 $\times 10^{-127}$.
- You can represent this by using an exponent part of 00000000 and a significand of 00000100010110110000100. This gives you five fewer digits than usual, but it's better than not being able to represent the number at all.

In Exercise 1.2 toward the end of this chapter, you are asked to write a set of functions to work with floating-point numbers. The discussion has concerned 32-bit systems. Computers are now commonly 64-bit. With a 32-bit processor, some languages use *double precision* numbers, which combine two 32-bit numbers into one to give a 64-bit number. This allows a much wider range of numbers to be represented, with much greater precision.

Common Functions

This section explores a few functions that are common, in one form or another, to most computer languages: `abs()`, `floor()`, `ceil()`, and `round()`. Some languages provide these functions with different names, but they work the same, and the fact that they are so commonly provided testifies to their usefulness. In this context, the discussion involves how they are implemented.

Absolute Values

The simplest of them is the `abs()` function, which returns the *absolute value* of a number (float or integer). This is the value of the number, ignoring whether it is positive or negative. A function that finds the absolute value can be implemented this way:

```
function abs(n)
  if n>=0 then return n
  otherwise return -n
end function
```

The absolute value of a number is its value made positive. In the `abs()` function, if n is negative then $-n$ is positive. Of course, the computer doesn't natively need to do anything even this complicated. It simply sets the value of the sign bit in the floating-point or integer representation to 0.

Floats to Integers

Three functions are commonly used to convert float to integer values. Their specific purposes are as follows:

- For a number n , the `floor()` function finds the largest integer less than n .
- For a number n , the `ceil()` function finds the smallest integer greater than n .
- For a number n , the `round()` function finds the integer nearest to n .

Figure 1.2 illustrates the actions of these three functions. Each range is represented by a solid line at one end and a broken line at the other. The solid and broken lines represent inclusion. For example, with `floor(n)=3`, n is greater than or equal to 3 but less than 4.

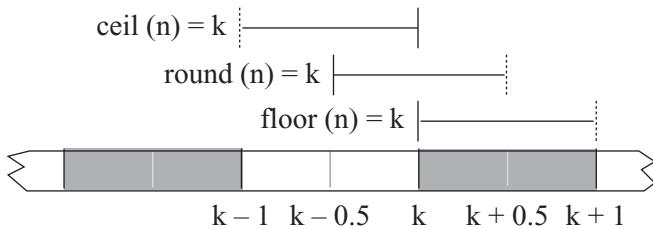


Figure 1.2

The ranges of floating point numbers that give the answer 3 using the `floor()`, `ceil()`, and `round()` functions.

Each of these functions is useful in different circumstances. The `floor()` and `ceil()` functions are easy for the computer to calculate using floating-point representations. If the exponent of the number is 5, then the functions can find the positive integer part (call it p) of the number by taking the first five digits of the significand. If it's less than 1, the integer part is zero. If it is greater than the number of digits in the significand, then the function does not know the exact integer part. All that is then necessary is to check the sign of the number. If it is positive, then `floor(n)` is equal to p and `ceil(n)` is $p + 1$. If it's negative, then `ceil(n)` is $-p$ and `floor(n)` is $-(p + 1)$.

The `round()` function rounds to the nearest whole number. This version of rounding is what most people learn in their early studies of math. The `round()` function finds an integer such that $\text{abs}(n - \text{round}(n)) < 1$. Numbers exactly halfway between two integers, such as 1.5, are ambiguous. You define the `round()` function to round up in such cases. Given that the `floor()` and `ceil()` are defined, the `round()` function can be implemented as follows:

```
function round(n)
    Set f to floor(n)
    Set c to ceil(n)
    If n - f > c - n then return c // n is nearer c than f
    If c - n > n - f then return f // n is nearer f than c
    Otherwise return c // n is half-way between c and f
End function
```

While the `round()` function suffices to illustrate how a number can be rounded, it remains that it is simpler to calculate the value directly from the floating-point representation. Using this approach, you find the integer part p as before. To decide whether to round up or down, you look at the next digit of the significand, which in binary represents the value $\frac{1}{2}$. If this digit is 0, then the fractional part of the number is less than $\frac{1}{2}$, which means that you round down. If it's 1, then the fractional part is greater than or equal to $\frac{1}{2}$. If it is greater than or equal, you round up. The arbitrary decision to round up halves turns out to be quite handy. If you round them down, and the first digit of the fractional part is a 1, you still do not know whether to round up or down without looking at the rest of the digits.

Rounding Errors and Performance

No matter how precise you make floating-point numbers, they are still not exact except in special circumstances (such as exact powers of 2). This means that a danger exists of introducing errors into calculations that depend on exact precision. In most circumstances, this is not a major problem, but it is not hard to come up with examples where it is an issue.

For example, ask the computer for the result of $\left(\frac{7}{50} + 1\right) \times 50 - 57$. Performing the calculation

on paper, you see that the answer should be 0, because $\frac{7}{50} \times 50 = 7$ and $1 \times 50 = 50$. However, the computer reports that the answer is 7.10542735760100 e-15. Why does this happen? The answer is that when you divide by 50, you end up with a number with a negative exponent. Adding 1 creates a number with an exponent of 0, which means that a whole lot of bits are dropped off the end of the significand. Multiplying back up by 50 does not restore the lost bits.

Rounding errors are a major source of problems when performing complex and repeated calculations, especially “stretch and fold” calculations that involve adding small numbers to larger ones. In general, there is not a great deal you can do about such problems, but in some cases, doing some pre-calculations alleviates the problem a little.

If you know that you are going to drop and then restore bits, to avoid the problem, you might be able to make your calculations in a different order. In the previous example, consider what happens if you perform the calculation as $\left(\frac{7}{50} \times 50\right) + (1 \times 50) - 57$. In this case, dividing by 50 and then multiplying again results in little loss of precision. Simply adding and subtracting a small integer like 50 is not going to change or worsen the situation.

One other issue to notice here is that testing whether two floating-point calculations are equal may also prove to be a problem. Consider what happens if you ask whether $\frac{7}{50} + 1 \times 5000 = 5700$. The answer is no, but it should be yes. In general, it is best when checking for equality of two floats to call them equal when their absolute difference is less than a small value. So, instead of saying `if x = y`, a preferable approach is to say `if abs(x - y) < 0.00001`.

None of these workarounds solves the rounding error problem completely. If you are doing calculations that require lots of precision, you might have to use specialized calculation algorithms that allow you to retain as much precision as possible. These will be considered more closely in a moment.

Floating-point calculations are also more expensive in terms of computer time than are calculations with integers. For example, you might find that a given processor using floats calculates 2×2 a million times over and requires 1.2 seconds. Integer calculations might take under a second. This is a fairly significant difference for such a simple task. One conclusion is that if at all possible, it's beneficial to use integer calculations to perform the task you are trying, especially in games or other situations where there are a lot of calculations and speed is an issue.

You can use integer calculations even if you are working with fractions. To do so, multiply all numbers by a sufficiently large number at the start, convert to integers, and complete all your calculations before dividing again by the number at the end. Still, while most calculations will work in this way, caution is the best policy. Test your outcomes as you go.

Having said this, it is certainly the case that the difference between floating-point calculations and integer calculations is not as big a deal as it once was. In the early days of floats, the difference in speed was often critical. These days, processors are geared to deal with floats on a hardware level just as they are with integers, so the difference is not usually important. Your time is probably better spent optimizing other areas of your code.

Big Integers

One way to deal with functions requiring great precision is to work with `BigInteger` classes. Such classes generate specialized objects that store numbers in long arrays instead of using the computer's processor to store and work with them. Something similar to this action was taken when, previously in this chapter, a binary number was represented with a long string of digits.

When you perform calculations in this way, you are no longer limited by the processor's built-in number storage. Your array can be as long as you like, limited only by the computer's memory. As a result, instead of having 32 or 64 bits to represent a number, you might have a trillion bits or more, which should be sufficiently precise to deal with any calculation you could conceivably want to perform. Of course, since you must perform the calculations at the level of your programming language rather than directly within the processor and each calculation involves many more digits, this approach is much slower. However, in situations in which you need great precision (such as detailed physics simulations of hair, water, or fabric), generally you are thinking in terms of letting the computer do the calculations overnight rather than at lightning runtime speed.

`BigInteger` classes are available for most languages used by scientists. Consider, for example Java, C#, and C/C++. That they are not dealt with specifically in this context does not make a great deal of difference. The mathematical explorations in this book do not require them.

Exercises

EXERCISE 1.1

Write a function `convertBase` (`NumberString`, `Base1`, `Base2`), which takes a string (or array) `NumberString`, representing an integer in the base `Base1` and converts it into the base `Base2`, returning the new string (or array). Make it handle any base from 2 to 16.

You might want to make special cases for binary and hexadecimal translation to speed up the function.

EXERCISE 1.2

Write a series of functions for calculating addition, subtraction, multiplication, and division using IEEE-style floating-point numbers. Represent the floats as 32-bit strings (or arrays) of 1s and 0s.

Each function should take two floating-point numbers and return another. Don't forget the special case when the exponent is zero. You will probably find division to be the most difficult case.

Note

Hints for the solutions of the exercises appear in Appendix E. Some answers, in pseudocode, can be found on the book's companion website.

Summary

This chapter has covered a lot of ground. Numbers have been dealt with in theory, and an attempt has been made to establish how it is that computers approach different computational situations. How computers deal with problems involving fractions can lead to situations in which precision becomes questionable, and floating-point numbers are more costly to compute than are integers. In the next chapter, the theory of this chapter is extended to basic arithmetic operations.

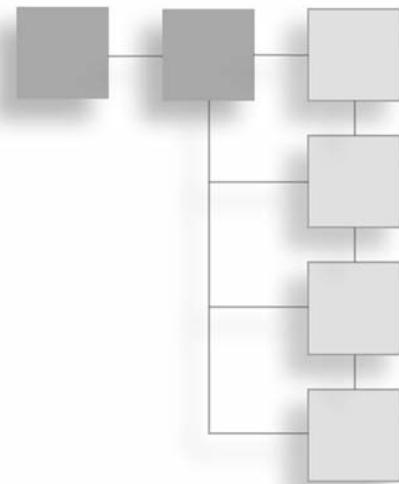
You Should Now Know

- The meaning of the terms *integer*, *fraction*, *rational*, *numerator*, *denominator*, *fixed-point number*, *floating-point number*, *exponent*, *mantissa*, *significand*, *base*, *binary*, *decimal* and *decimal point*, among others
- The difference between the *value* of a number and its *representation* as a string of digits
- How computers *represent* and calculate with integers and floating-point numbers
- How to convert a float to an integer in various ways
- Why *rounding errors* are an essential consequence of working with floats and how to avoid them in some circumstances

This page intentionally left blank

CHAPTER 2

ARITHMETIC



In This Chapter

- Overview
- Fractions
- Proportions, Ratios, and Percentages
- Exponentials
- Logarithms

Overview

This chapter and the next explore basic mathematics. It is necessary to cover this ground to prepare for topics presented later in the book. As it is, many people come up against problems in more advanced settings because they do not understand or have forgotten basic concepts. Without the basics, it is sometimes possible to find workaround strategies or use techniques that have been learned by rote, but in the end, understanding how things work ensures that you will be able to continue indefinitely.

This chapter examines the essentials of arithmetic, working with what in most computer languages are data types identified as integers and floats. While working with these data types does not cover the whole of arithmetic, it does provide pointers on a few topics that are often troublesome.

You can find pseudocode samples relating to this chapter on the book's companion website.

Fractions

You learned some of the details of working with fractions in the previous chapter, but because so many people seem to have trouble with them, this chapter presents them in more detail, starting with some basic arithmetic and moving a few steps beyond.

Calculating Fractions

A fraction is something like an instruction: it tells you to divide a by b . Fractions tend to appear in different ways in print. For example, the fraction $\frac{2}{5}$ is exactly the same as the value $2 \div 5$. In fact, the division symbol \div is a representation of this process: the two dots represent the numerator (a) and the denominator (b) of the fraction.

Note

In this book, fractions are represented two ways. In some instances, you see them represented using a diagonal line, as in $\frac{1}{2}$. In other instances, you see them with a horizontal line, as in $\frac{1}{2}$. A horizontal line is used to save space.

Multiply Two Fractions

To multiply two fractions, you multiply the numerators and multiply the denominators:

$$\frac{2}{5} \times \frac{3}{7} = \frac{2 \times 3}{5 \times 7} = \frac{6}{35}$$

Multiplication is represented by the English word “of,” as in “two fifths of three sevenths,” just as with integers you might say “five of these apples” and mean $5 \times$ apple. It’s easier to see how it works by breaking the process into two steps: dividing by the denominator (“take one fifth of three sevenths”) and then multiplying by the numerator (“now you want two of those”). When multiplying a fraction by an integer, remember that an integer is a fraction with a denominator of 1, which means that you can multiply the fraction’s numerator by the integer and leave the denominator alone (multiply it by 1).

Divide Fractions

To divide $\frac{2}{5}$ by $\frac{3}{7}$, first flip $\frac{3}{7}$ over so that it becomes $\frac{7}{3}$. The one is the reciprocal of the other. Then you multiply:

$$\frac{2}{5} \div \frac{3}{7} = \frac{2}{5} \times \frac{7}{3} = \frac{14}{15}$$

Why do this? Well, for starters, consider the behavior of integers. When you divide by 2, you halve the number (which means taking half of the number). As you just observed, this is the same as multiplying by $\frac{1}{2}$.

When you divide a number by $\frac{1}{2}$, you want to find out how many halves go into it (which is twice as many as the number of 1s that go into it). Since the number of 1s that go into a number is the number itself (anything divided by 1 is unchanged), this means that a number divided by $\frac{1}{2}$ is multiplied by 2.

As a general notion, division is represented by several words in English, such as “per”—your speed in miles *per* hour is the number of miles travelled divided by the number of hours taken—as well as “each” and “a.”

Addition and Subtraction of Fractions

Addition and subtraction of fractions is a little more complicated than the previously discussed activities. As long as two fractions have the same denominator, you can add and subtract them fairly easily. You leave the denominator unchanged and perform normal arithmetic with the numerators. With addition, $\frac{3}{5} + \frac{1}{5} = \frac{4}{5}$. With subtraction, $\frac{3}{5} - \frac{1}{5} = \frac{2}{5}$. In both cases, you are counting, and the objects with which you are counting happen to be units of $\frac{1}{5}$. It is only when you want to add fractions with different denominators that you hit problems, and this is so because the units you are trying to add are different.

Fortunately, every fraction can be represented an infinite number of ways, so if you multiply the numerator and denominator by any number at all, you end up with the same fraction. This is because any number divided by itself is 1. For example, the fractions $\frac{2}{2}$ and $\frac{3}{3}$ are both equal to 1. As a result, you can multiply any given number by such fractions and leave the number unchanged. Calculations can be made easier by choosing the most convenient representation of them.

Suppose you want to add $\frac{1}{2}$ to $\frac{1}{4}$. It's much easier to do this if you recognize that $\frac{1}{2}$ is equal to $\frac{2}{4}$ (multiplying the top and bottom by 2). Because both fractions have the same denominator, you can use this fact to make the problem into one involving arithmetic among the values of the denominators: $\frac{1}{4} + \frac{2}{4} = \frac{3}{4}$. This process even works if you are adding fractions that can't be easily converted, such as when $\frac{2}{3}$ is added to $\frac{3}{5}$.

In this instance, you can “cross-multiply,” which means that you multiply the numerator and denominator of each fraction by the denominator of the other. In the language of arithmetic, you “put them over a common denominator” (see Figure 2.1).

$$\frac{2}{3} + \frac{3}{5} = \frac{2 \times 5}{3 \times 5} + \frac{3 \times 3}{5 \times 3} = \frac{2 \times 5 + 3 \times 3}{3 \times 5} = \frac{19}{15}$$

Figure 2.1

Cross-multiplying.

Now consider how some of the arithmetic operations can be applied to code. To make things easier to understand, a pseudocode function can be created. The function encapsulates a problem involving fractions. Its name is `addFractions()`, and it takes two arguments, each a fraction represented by a two-element array:

```
function addFractions(f1,f2)
    set num1 to the numerator of f1
    set den1 to the denominator of f1
    set num2 to the numerator of f2
    set den2 to the denominator of f2
    if den1 = den2 then return the fraction(num1+num2, den1)
    otherwise
        set num3 to num1*den2
        set num4 to num2*den1
        return the fraction(num3+num4,den1*den2)
    end if
end function
```

A problem with this function is that the calculations it involves are potentially much more complicated than they need to be. Consider the expression presented earlier, $\frac{1}{2} + \frac{1}{4}$. With the `addFractions()` function, you calculate $\frac{4}{8} + \frac{2}{8}$, instead of the simpler $\frac{2}{4} + \frac{1}{4}$. What’s more, the final answer is given as $\frac{6}{8}$ rather than $\frac{3}{4}$. While this is technically correct, it is not what you really need.

Factors and Factorization

In the previous section, the technique for adding fractions was to change the representation of the fractions to a more complicated representation before adding them. The problem with this approach is that you end up with more complexity than is needed. A better approach is to switch them back to a simpler version at the end. This is called *reducing a fraction to its lowest terms*, and to do this, it's necessary to look at factors. A *factor* or divisor of an integer is an integer that can be exactly divided into the integer. So 1, 2, 3, and 6 are all factors of 6. If n is a factor of m , then you say that m is a *multiple* of n (or more strictly, an *exact multiple*, or *integer multiple*).

Note

The number 6 is called a “perfect number,” because it is the sum of all its factors other than itself. The numbers 1, 2, and 3 are the factors of 6, and $1 + 2 + 3 = 6$. The Pythagoreans were particularly fond of perfect numbers.

Consider the fraction $\frac{6}{8}$, introduced previously. You can see that it would be much simpler if you divided everything by 2, making it $\frac{3}{4}$. The number 2 is a factor of both the numerator and the denominator: it is a *common factor* of 6 and 8. In fact, because 6 and 8 have no common factor greater than 2, 2 is called the *highest common factor*, or more often, the *greatest common divisor*, or GCD. To reduce a fraction to its lowest terms, you need to divide the numerator and denominator by their GCD.

Note

You will sometimes see the GCD of two numbers n and m written as (n, m) . Since this notation is easily confused with other notations of the same kind, in this context, $\text{gcd}(n, m)$ will be used.

In early studies of arithmetic, you probably found GCDs by finding the prime factors of two or more numbers. There is a simpler method, but for now it is useful to continue considering prime factors. They are useful in their own right. To explain prime factors, it is first necessary to define *prime numbers*. A prime number (or just “a prime”) is a number whose only factors are itself and 1. Conventionally, you exclude the number 1 from the list of primes because it's something of a special case and makes for a less redundant

pattern. The list of primes goes 2, 3, 5, 7, 11, 13, 17, 19, The prime numbers are something like the building blocks of multiplication, and there is an enormous body of mathematical writing about them—most of which has been by mathematicians looking for patterns in their distribution or ways to test whether a number is prime. Such studies are particularly important in the field of cryptography.

Generating Prime Numbers

You can generate a list of prime numbers less than some number M quite easily using a technique known as *Eratosthenes' Sieve*. To do this, start with a list of all the numbers from 2 to M . Take the first number in the list, and delete all the numbers in the list that are divisible by this number. Remove the number from the list and add it to your list of primes. Then repeat this process. You need to do this only until the first number in the list is greater than \sqrt{M} . After you're past that point, all the remaining numbers must be prime, since they're not divisible by any number smaller than the square root, and any product of two numbers bigger than \sqrt{M} must be bigger than M .

Here's the algorithm in pseudocode:

```
function listOfPrimes(M)
    // make an array with all the numbers from 2 to M and call it nlist
    set maxM to floor(sqrt(M)) // the biggest number that must be checked
    set index to 1
    repeat while nlist[index] is less than maxM
        set prime to nlist[index]
        repeat for index2 = index+1 to the number of elements in nlist
            if nlist[index2] is divisible by prime then delete it from the list
        end repeat
        add 1 to index
    end repeat
    return nlist
end function
```

To return to the prime factors, a prime factor of a number n is, as you might expect, a prime number that is a factor of n . Every number can be represented in a unique way as a product of prime numbers—for example, $12 = 2 \times 2 \times 3$. This fact goes by the very grand name of the *Fundamental Theorem of Arithmetic*. A number with more than one prime factor is said to be *composite*.

Note

The Fundamental Theorem of Arithmetic is one of the reasons that mathematicians contend that 1 is not prime. If it were, it would be an exception to this rule of uniqueness. You could represent any number as a product of its prime factors and any number of 1s.

To find the prime factors of an arbitrary number n , the only absolutely reliable way is to keep dividing it by prime numbers until you find one. Of course, this can be a slow process for large numbers, but it's not too bad for small ones, especially if you have already generated a list of primes. Here's a function that will do it.

```
function primeFactors(n)
    // create a list of possible primes
    set plist to listOfPrimes(floor(n/2))
    set rlist to a blank array
    set index to 1
    repeat while index <= the number of elements in plist
        set prime to plist[index]
        if n is divisible by prime then
            set n = n/prime
            add prime to rlist
        otherwise
            add 1 to index
        end if
    end repeat
    return rlist
end function
```

Note

When people are searching for extremely large primes (as they often do for cryptographic or other purposes), they don't use this method because it would take an unfeasibly long time even on a very fast computer. Instead, they generally use various tricks to hone in on numbers that are extremely likely to be prime to within some particular degree of certainty. Depending on the application, this is usually enough.

GCD and Euclid's Algorithm

To return to the question of the GCD of two numbers, if you want to find $\text{gcd}(n, m)$, you can do so by finding the prime factors of each of the numbers. Say that n is 24 and m is 60. You use the function previously defined to find that the prime factors of n are 2, 2, 2, 3. Of m the prime factors are 2, 2, 3, 5. Then you just take as many elements of each list as you can that match, which are two 2s and one 3. So $\text{gcd}(n, m)$ is $2 \times 2 \times 3 = 12$.

On the other hand, you could use the phenomenally simple and ingenious method known as *Euclid's Algorithm*, which is approximately a gazillion times faster.

Euclid's Algorithm relies on a simple observation. If you have three numbers, a, b, c , such that $a = b + c$, and if some number d is a factor of both a and b , then it must be a factor of c as well.

Armed with this knowledge, you can design another function. To find $\text{gcd}(n, m)$, where $n \geq m$, divide n by m and find the remainder r . Then you know that $n = a \times m + r$, for some a , where $r < m$ (recall that this is the definition of the remainder). If $r = 0$, then m is a factor of n , so $\text{gcd}(n, m) = m$. Otherwise, because the GCD is a factor of both n and m by definition, it must also be a factor of r . And having gone this far, you repeat the process using the numbers m and r in place of n and m . Here is the pseudocode for the function:

```
function gcd(n, m)
    set r to the remainder of n/m
    if r = 0 then return m
    otherwise return gcd(m, r)
end function
```

Eventually, you bottom out with a number d that is an exact divisor of the two arguments, and you can then work back to find that it is a common divisor of both of the original numbers. Euclid's Algorithm is one of the most beautiful things in mathematics!

Lowest Common Multiple

After you have $\text{gcd}(n, m)$, you can also find another useful number, the *least (or lowest) common multiple* (LCM) of n and m . This is the smallest number that has both n and m as factors, and it is equal to $n * m / \text{gcd}(n, m)$. The LCM brings us back to the sum of two fractions.

When adding two fractions, instead of cross-multiplying, the most efficient system is to “put them over the lowest common denominator.” In other words, you multiply the top and bottom of each fraction by the smallest possible number so that they have the same denominator. This lowest common denominator is the LCM of the two denominators.

Consider how this works by calculating $\frac{7}{24} + \frac{7}{30}$. Instead of a bare cross-multiplication, you multiply each fraction, top and bottom, by the denominator of the other, *divided by the GCD of the two denominators*. Here is what happens:

- By Euclid’s Algorithm, $\gcd(24, 30) = 6$, you must multiply the first fraction, top and bottom, by $\frac{30}{6} = 5$, and the second by $\frac{24}{6} = 4$.
- This gives you $\frac{35}{120} + \frac{28}{120}$, which has a common denominator (the LCM of 24 and 30) of 120.
- Because of the common denominator, you can now simply add to get $\frac{63}{120}$.
- Now you need only to reduce this to its lowest terms by finding $\gcd(63, 120)$. Using Euclid’s Algorithm, you see that $120 = 63 + 57$, $63 = 57 + 6$, $57 = 9 \times 6 + 3$, and $6 = 3 \times 2$, so $\gcd(120, 63) = 3$.
- Therefore, you can divide the answer, top and bottom, by 3 to get $\frac{21}{40}$, which is now in its lowest terms.

Note

The fraction $21/40$ is in its lowest terms because $\gcd(21, 40) = 1$. In such a situation, you say that 21 and 40 are relatively prime or co-prime.

Modulo Arithmetic

As with several other techniques considered so far, Euclid’s Algorithm relies on a process of finding the remainder when dividing one number by another. This turns out to be such a useful procedure that there is a standard term for it. If you have two numbers n and m , such that when you divide n by m you get a remainder r , it is said that “ n is congruent to r modulo m .” Congruence is normally expressed with a special triple bar symbol (\equiv), so you write $n \equiv r \pmod{m}$.

Actually, the congruence relation is a little more subtle than that, for two numbers are congruent mod m whenever they differ by an integer multiple of m . So when working modulo 5, the numbers 4, 9, 14, . . . are all equivalent because they are all congruent to 4.

Most programming languages allow you to work with modulo congruence relations through a modulo function. The function takes two arguments, n and m , and returns the remainder r . The expression is coded along the lines of $\text{mod}(n, m) = r$.

If $n \geq 0$, then the `mod()` function returns a value between 0 and $m - 1$. If $n < 0$, it generally returns a value between $-(m - 1)$ and 0 (in the most commonly used languages, at least). Since in itself it is very useful, you can use a modified modulo function. Here is the pseudocode for this function:

```
function StrictModulo(n, m)
    set r = mod (n,m)
    if r<0 then return r + m
    otherwise return r
end function
```

The `StrictModulo()` function always returns a value between 0 and $m - 1$, and the congruence relationship $n1 \equiv n2 \pmod{m}$ is equivalent to this expression in code:

```
StrictModulo(n1, m) = StrictModulo(n2, m)
```

Although you might find any number of instances in which the modulo function can be applied, the most common use is to decide whether one number is divisible by another. If n is exactly divisible by m , then `StrictModulo(n, m) = 0`.

Note

The modulo function is not symmetric around zero. In general, `StrictModulo (n, m)` is not equal to `StrictModulo (-n, m)`. In fact, you generally have `StrictModulo (n, m) = m - StrictModulo(-n, m)`, unless n is a multiple of m , in which case both values are zero.

You can use another method to calculate the modulo. If you find `floor(n/ m)`, this gives the largest integer less than $\frac{n}{m}$. Then $m * \text{floor}(n/m)$ is the largest multiple of m less than n . This means that for a positive integer, $\text{mod}(n, m) = n - m * \text{floor}(n/m)$. Obviously, there are minor dangers in switching between floats and integers, however, so it's best to avoid this method as a general rule.

Cycling Through Data

Because it is commonly used in situations where something cycles through a small list of options, modulo arithmetic is sometimes known as *clock arithmetic*. A clock contains only 12 numbers, so when calculating hours, $8 + 6 = 2$. This is an example of modulo arithmetic in action.

In programming, you can use modulo calculations to deal with circumstances like these. For example, suppose you create a clock and want to be able to add times together. Without modulo arithmetic, a pseudocode function would look something like this:

```
function naiveClockAdd(oldHours, oldMinutes, addHours, addMinutes)
    set newMinutes = oldMinutes + addMinutes
    repeat while newMinutes>60
        // drop the number of minutes by 60 and add another hour
        subtract 60 from newMinutes
        add 1 to addHours
    end repeat
    set newHours = oldHours + addHours
    repeat while newHours>12
        // drop 12s until you're in the range 1-12
        subtract 12 from newHours
    end repeat
    return array(newHours,newMinutes)
end function
```

Obviously, this isn't a particularly complicated function. But still, using modulo arithmetic, it can be substantially simplified:

```
function cleverClockAdd(oldHours, oldMinutes, addHours, addMinutes)
    set newMinutes = strictModulo (oldMinutes + addMinutes, 60)
    add (oldMinutes + addMinutes - newMinutes) / 60 to addHours
    set newHours = 1+ strictModulo (oldHours + addHours - 1, 12)
    return array(newHours, newMinutes)
end function
```

Since such calculations take some getting used to, it's helpful to examine them in more detail. In the first line, the new minutes are worked out. These should be in the range 0–59. Next, the full number of new minutes (`oldMinutes + addMinutes`) is evaluated, with the result that their value is modulo 60. The next line works out how many hours were taken out in the calculation, and for this a trick is employed. Recall that `strictModulo(n, m)` is the remainder when n is divided by m . This means that, for some a , $n = a*m + \text{strictModulo}(n, m)$. To find a , then, you subtract `strictModulo(n, m)` from n and divide by m . In this case, since a is the number of hours (multiples of 60 minutes), you increase `addHours` by this amount.

Note

Although this method is often useful and so worth exploring, in this case it would be simpler to find a using the function `floor(n/m)`.

In the third line of the `cleverClockAdd()` function, you find the value for the `newHours` variable. The only complication here is that the value of `newHours` must be in the range 1–12, rather than 0–11 (as the `strictModulo()` function gives). To fix this problem, you do something a bit sneaky. You subtract 1 from `newHours` before taking the modulo, which results in a value one less than the true number of hours in the range 0–11. Then you add 1 back on, rendering the true answer in the needed range.

Such calculations can be a little confusing, but most situations involving the modulo function tend to be variants on the three tricks just reviewed:

1. Use `mod(n, m)` to reduce n to the range 0 to $m - 1$ (cycling through the data).
2. Subtract `mod(n, m)` from n to make n exactly divisible by m .
3. Pre-subtract or add numbers from or to n before taking `mod(n, m)`; then add or subtract them back to bring the value into the desired range, especially 1 to m .

A further example of the use of the modulo function involves initializing large sets of data, especially when creating a grid. Suppose you have an array of $n \times m$ objects that represent a grid of squares n wide and m high. Each object knows its number in the array and wants to calculate its position in the grid. You can satisfy this requirement quickly with a modulo function:

```
function positionInGrid (squareNumber, numberColumns)
  set positionAcross to 1 + mod(squareNumber-1, numberColumns)
  set positionDown to (squareNumber-
    mod(squareNumber, numberColumns)) / numberColumns
  return array(positionAcross, positionDown)
end function
```

Proportions, Ratios, and Percentages

Returning to fractions, one of the most powerful interpretations of a fraction is as a ratio between the numerator and denominator. Another is as a percentage. This section considers these two concepts in detail.

Mapping Between Ranges of Values

“Ratio” (or proportion) actually means pretty much the same thing as “fraction,” but it is used in different circumstances, particularly when relating similar objects. (*Similar* here has a precise meaning, as well as its normal English one, and its precise meaning will be encountered in Chapter 5). Ratios are sometimes written using the colon. The ratio 4:3, for example, is equivalent to the fraction $\frac{4}{3}$. Two objects are in the size ratio 4:3 if one is $\frac{4}{3}$ times the size of the other. Frequently, ratios are used to describe how something is divided up; so you might choose to divide a cake in the ratio 1:2, making one piece twice as big as the other. To do this, you use the fact that $\frac{1}{1+2} + \frac{2}{1+2} = \frac{1+2}{1+2} = 1$. In other words, you divide the cake into two pieces, one $\frac{1}{3}$ the size, the other $\frac{2}{3}$ the size.

Paper

Ratios are most useful when dealing with scales of objects. If a piece of paper is 297 mm wide and 210 mm high (international standard A4 size), when you scale it to double its size, its sides remain in the same proportion. This isn’t surprising because, as has already been shown, multiplying both the numerator and the denominator of a fraction by the same number leaves the fraction unchanged.

This fact is quite useful when scaling an object. If you know the initial ratio of the height to the width, then to calculate the height given the width, all you need to do is to multiply the new width by the same ratio. Here is the pseudocode for a function that accomplishes this:

```
function newHeight(originalWidth, originalHeight, newWidth)
    return newWidth * originalHeight / originalWidth
end function
```

You can also think about this the other way round. When scaling any object, every linear measure of the object (its diagonal, height, width, perimeter, and so on) is multiplied by the same amount. If you know the original value and new value of any one of these, every other dimension must be multiplied by the proportion of the new value to the old. Consider, for example

$$\text{newDiagonalLength} = \text{originalDiagonalLength} * \text{newHeight} / \text{originalHeight}$$

This fact makes it easy to create a mapping function between two rectangles. If you know the coordinates of a point in one rectangle, you can relate them to the equivalent point in the other rectangle by multiplying them by the same proportion.

Incidentally, the A classification of paper sizes is chosen specifically so that when you cut a piece of paper in half horizontally (in portrait format), you get the next A-size up (see Figure 2.2). You can represent this algebraically as

$$\frac{aHeight}{aWidth} = \frac{aWidth}{(aHeight / 2)}$$

which translates to

$$aHeight^2 = 2 \times aWidth^2$$

and

$$\frac{aHeight}{aWidth} = \sqrt{2}$$

So for each size of paper, the height is $\sqrt{2} = 1.414 \dots$ times the width (or very close to it): $\frac{297^2}{210} = 2.0002 \dots$ Figure 2.2 illustrates how the rectangles diminish as the ratio is applied.

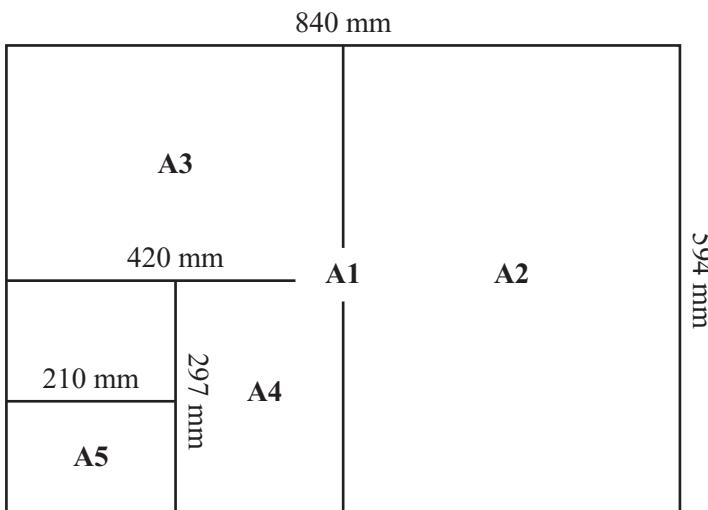


Figure 2.2

The A-series of paper sizes.

The Golden Ratio

Another useful proportion is called the *golden ratio*. A highly celebrated phenomenon among mathematicians, it is represented by the Greek letter Phi: φ (pronounced *fee*). This is the proportion of a rectangle such that, if you cut a square-sized portion from one end, the remaining piece is in the same proportions as the original (see Figure 2.3). The algebra will not be explained in this context, but it turns out that $\varphi = \frac{1+\sqrt{5}}{2} = 1.618 \dots$

As Figure 2.3 illustrates, a pattern similar to the one glimpsed in Figure 2.2 results.

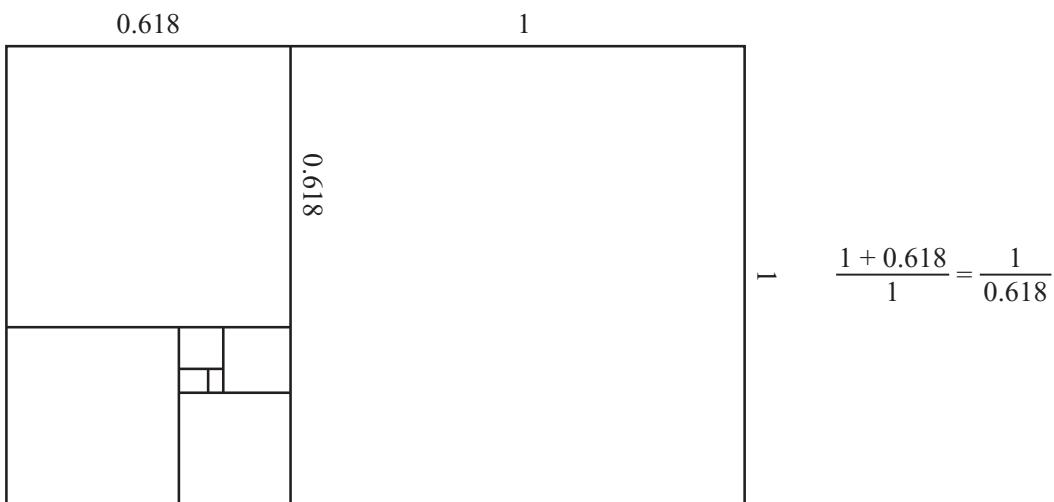


Figure 2.3
The Golden Ratio.

The ancient Greeks and other thinkers through the ages have ascribed all kinds of wonderful properties to φ , calling it the *Divine Proportion*, the *Golden Mean*, and so on. They have contended that objects placed on a canvas (among other things) with such proportions are the most aesthetically pleasing. This is true to an extent (consider the façade of the Parthenon, which is often flaunted as an example), but recent psychological tests on the subject have shown that this isn't universal. The proportion $1:\sqrt{2}$, as with the standard paper size, is much closer to the most popular. A further point, however, is that what people currently prefer could be explained by their continual exposure to A-series paper.

Fibonacci Sequence

When the Golden Ratio is mentioned, the *Fibonacci Sequence* is often the next topic. The Fibonacci Sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, . . . , where each number is the sum of the two previous numbers. The ratio between successive terms of this sequence rapidly approaches φ , and this happens with any sequence of numbers generated by the same rule as the Fibonacci Sequence. The Fibonacci Sequence has proven interesting in a number of areas of science, for it frequently crops up in nature, describing, for example, such things as the concavity of seashells or how leaves on a vine are distributed.

Making a Slider

Another common area where the notion of proportion is used involves sliders. A slider is a graphical representation of a range of numbers. In the standard slider, a line represents the range of numbers you are looking at (say from 10 to 100), and a movable pointer represents the current value (see Figure 2.4).

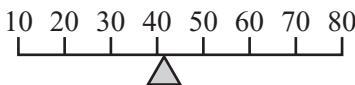


Figure 2.4

A standard slider.

To make a slider, you must know four values: the position of each end of the line and the maximum and minimum values they represent. For now, it is not necessary to know exactly what is meant by the “position” of the ends of the line. Just think of it as a number—say 100 for one end and 200 for the other.

When you initialize the slider, you can quickly calculate what might be expressed as its “intrinsic proportion.” You can think of this as the range of values represented by one unit of the slider. You calculate the proportion by finding the size of the total range divided by the total length of the slider.

$$\text{intrinsicProportion} = \frac{\text{maxValue} - \text{minValue}}{\text{endPoint2} - \text{endPoint1}}$$

To find the value represented by a particular point on the slider, you find how far the value is along the slider, multiply it by the intrinsic proportion, and then add on the minimum value:

$$Value = (thisPoint - endPoint1) \times intrinsicProportion + minValue$$

Notice how similar this is to calculating proportions of a scaled piece of paper. Ultimately, it's the same process. You've created a virtual piece of paper whose width is the length of the slider, and whose height is the range of values represented. When a new point is found, you scale down to another rectangle with the new width, but in the same proportions. It's necessary only to remember that your rectangle doesn't necessarily start at zero, so you have to add on the minimum value, as well.

You can also go the other way. If you have a particular value you want to represent, you can find its position on the slider by finding its distance from the minimum value, dividing by the intrinsic proportion, and adding the endpoint:

$$newPoint = \frac{thisPoint - minValue}{intrinsicProportion} + endPoint1$$

If you have trouble deciding whether to complete the calculation by dividing or multiplying, plug in one of the end values and see what happens. For example, try setting *thisPoint* to *minValue* in the first formula. You'll see that you get *newPoint* = 0 + *endPoint1*, which is the first endpoint. If you instead set *thisPoint* to *maxValue*, you get

$$newPoint = \frac{maxValue - minValue}{intrinsicProportion} + endPoint1$$

Recall that dividing by a fraction is the same as multiplying by its reciprocal, so this translates to

$$newPoint = (maxValue - minValue) \times \frac{endPoint2 - endPoint1}{maxValue - minValue} + endPoint1$$

You can see that in this fraction, the terms (*maxValue* - *minValue*) cancel out, which leaves:

$$newPoint = endPoint2 - endPoint1 + endPoint1 = endPoint2$$

So this is the second endpoint. Try doing the same checks in the second formula.

Scrollbars are a special kind of slider. They represent the position of a small image within a large one. The principle is the same as with the standard slider, but it is complicated by the fact that the small image may vary in size. For example, consider a scrolling piece of text. If you know the height of the text in pixels and the height of your scrolling window, then your scrollbar is a slider that represents the values between 0 (the topmost position) and $textHeight - windowHeight$ (the bottommost position). These values represent the pixel location of the text at the top of your window. When the text is at position $textHeight - windowHeight$, then the bottom of the window, $windowHeight$ pixels away, is at $textHeight$, the bottom of the text.

If you look at the scrollbars in standard windows, you see that they also resize the sliding marker within the bar, to represent how much of the image you can see. This proportion is given by $windowHeight / textHeight$ (see Exercise 2.1).

Calculating Percentages

Percentages are yet another kind of fraction. The number 20% is nothing more or less than the fraction $\frac{20}{100}$. (The word “percent” means “per cent,” or “per hundred.”) Calculating with percentages is therefore a fairly straightforward extension of calculating with fractions, with the help of your English-to-math translation skills.

1. 20% of 1000 is the same as $20/100$ multiplied by 1000, which is 200.
2. 20% of 1000 is $1000 - 20\%$ of 1000, which is $1000 - 200 = 800$.
3. 20% more than 1000 is $1000 + 20\%$ of 1000, which is $1000 + 200 = 1200$.

These last two calculations can be simplified slightly by adding or subtracting the percentage from 100 first:

1. 20% off 1000 is 80% of 1000.
2. 20% more than 1000 is 120% of 1000.

Going the other way is just as simple. If an item is knocked down from \$25 to \$20, you can calculate the percentage decrease. The item has dropped by \$5, which is $\frac{5}{25}$ of its original value. Since $\frac{5}{25} \times 100 = 20$, this is a decrease of 20%.

Compound Interest

Percentages only become complicated in the world of compound interest, such as when money is added to an account by a fixed percentage each year. If you have a bank account that pays 3% interest, and you put in \$1000, how much will you have at the end of 10 years? You might think that it's just a matter of adding 3% of \$1000, which is \$30, each year, but of course at the end of the second year you no longer have \$1000. Instead, you have \$1030. As a result, the percentage is wrong. To arrive at the right percentage, you must count the increase cumulatively.

Here is the output from the code in the mortgages.html file, which generates a 10-year schedule.

At the end of year 1 you have 103% of \$1000 = \$1030
At the end of year 2 you have 103% of \$1030.00 = \$1060.90
At the end of year 3 you have 103% of \$1060.90 = \$1092.73
At the end of year 4 you have 103% of \$1092.73 = \$1125.51
At the end of year 5 you have 103% of \$1125.51 = \$1159.27
At the end of year 6 you have 103% of \$1159.27 = \$1194.05
At the end of year 7 you have 103% of \$1194.05 = \$1229.87
At the end of year 8 you have 103% of \$1229.87 = \$1266.77
At the end of year 9 you have 103% of \$1266.77 = \$1304.77
At the end of year 10 you have 103% of \$1304.77 = \$1343.92

Notice how quickly compound interest rises compared to just adding \$30 each year. After 10 years, you have earned \$43.92 more than you would have by adding \$30 a year. This is a percentage increase of $\frac{43.92}{300} \times 100 = 14.64\%$. Your earnings are nearly 15% higher than they would have been under simple interest.

This then leads to the question of whether you have done that calculation. If not, look back over the beginning of this section until you understand it. The interest rises so much faster because it is an exponential growth (multiplying by the same amount each year) rather than a linear growth (adding the same amount each year). While this will receive more attention later on, for now note that the calculation can be summed up in this way: After n months you have $initialCash \times increase^n$, where the increase is the percentage

increase, given by $\frac{(100 + interest)}{100}$, or $1 + \frac{interest}{100}$.

Debts and Interest

Interest on debts is similar, although loans such as mortgages are complicated further by the fact that you are paying off the debt as you go along. You may have seen the “mortgage calculators” you can use to enter the amount of mortgage, interest rate, and years of payment. Since this is probably the most complicated type of percentage calculation, it’s worth extended consideration. (Incidentally, what is being described here is a particular kind of capital and repayment mortgage: the precise nature of mortgages varies from country to country and lender to lender.)

Consider first paying a monthly amount of \$1000 on a mortgage of \$100000, at an interest rate of 5%. How long would it take to pay off the mortgage?

To start with, you must convert the interest rate into a monthly form. To accomplish this, make it a fraction so you don’t have to worry about percentages. From this beginning, the annual interest rate is $\frac{5}{100}$, or 0.05. You divide this by 12 to get the monthly interest rate of 0.00417. Note that this already means that the true interest rate over the year is in fact more than 5%, because this is compound interest. The annual interest is $1.00417^{12} - 1 = 0.0511$, or 5.11%.

At the end of each month, the debt is increased by 0.00417. In other words, it is multiplied by 1.00417. After one month, this amounts to \$100417. Then your payment is taken off the debt, so the current value of the loan is now \$99417. Each month, you calculate $newLoan = oldLoan \times 1.00417 - \1000 .

Using this equation, here are the results:

At the end of year 1 the loan is \$92837.33
At the end of year 2 the loan is \$85308.21
At the end of year 3 the loan is \$77393.89
At the end of year 4 the loan is \$69074.65
At the end of year 5 the loan is \$60329.79
At the end of year 6 the loan is \$51137.52
At the end of year 7 the loan is \$41474.95
At the end of year 8 the loan is \$31318.03
At the end of year 9 the loan is \$20641.47
At the end of year 10 the loan is \$9418.67
At the end of month 130 the loan is \$0

This mortgage took just over 10 years to pay off—and you paid back significantly more than your original loan. Your total payment was \$129628.96, nearly 30% more.

You can write this as another formula, although the algebra is a lot more complicated:

$$\text{debtAfterNMonths} = \text{initialAmount} \times I^n - \text{monthlyPayment} \times (I^{n-1} + I^{n-2} + \dots + I + 1)$$

where I is the monthly increase, given by

$$1 + \frac{\text{annualInterest}}{1200}$$

The sequence $I^{n-1} + I^{n-2} + \dots + I + 1$ is a special case of what is called a *geometric progression*, and turns out to have a value of $\frac{I^n - 1}{I - 1}$, so the final formula is

$$\text{debtAfterNMonths} = \text{initialAmount} \times I^n - \text{monthlyPayment} \times \frac{I^n - 1}{I - 1}$$

As though using a normal mortgage calculator, suppose that you want to calculate what your monthly payment will need to be if you are to pay off the debt in a particular number of years, assuming a fixed interest rate. If you are setting the amount of debt to 0 in the previous formula, set n to the correct number of months, and invert the formula, the result is as follows:

$$\text{monthlyPayment} = \text{initialAmount} \times \frac{I - 1}{I^n - 1}$$

where $n = \text{numberOfYears} \times 12$.

Plug in the values of \$100,000 at 5% over 10 years, and you get a monthly payment of \$1060.66, which is just what the online calculators will tell you.

Exponentials

You have already used the power function several times, and speaking generally, this function represents “multiplying a number by itself n times.” While this description marks a good beginning, full understanding of the power function requires delving further into what can be done with the function.

Calculating with Powers

What does it mean to multiply a number by itself -0.3 times? It seems to make no sense, but if you try entering `power(2, -0.3)`, the computer will spit the value 0.81 back at you without hesitation. The idea of negative and fractional powers follows from the original definition. To see how this is so, consider some basic questions:

- What is n^{p+q} ?

This means multiply n by itself $(p + q)$ times. This is the same as multiplying it by itself p times and then multiplying the result by it q times. So $n^{p+q} = n^p \times n^q$.

- What is $n^{p \times q}$?

This means multiply n by itself $p \times q$ times. It is the same as multiplying it by itself p times, and doing that q times, which means $n^{p \times q} = (n^p)^q = (n^q)^p$.

- What is n^0 ?

By the addition formula, for any p , n^{0+p} must be equal to $n^0 \times n^p$. But $n^{0+p} = n^p$, so for any n , $n^0 = 1$.

- What is $(n \times m)^p$?

This means multiply $(n \times m)$ by itself p times. This is the same as multiplying n by itself p times, multiplying m by itself p times, and then multiplying the results. So $(n \times m)^p = n^p \times m^p$.

- What is $(n^m)^p$?

This appeared the other way round in the first example. Just to confirm, this means multiply n^m by itself p times, $n^m \times n^m \times \dots \times n^m$. By the first addition formula, this is equal to $n^{m \times p}$.

- What is n^{-p} ?

By the addition formula, you know that $n^{p-p} = n^p \times n^{-p}$. But $n^{p-p} = n^0 = 1$, so this means that $n^{-p} = \frac{1}{n^p}$.

- What is $n^{1/p}$?

Take a look at $(n^{1/p})^p$. This is equal to n^1 , which is equal to n^1 , which is n . Therefore $n^{1/p}$ is the number such that when taken to the p th power, it is equal to n , otherwise known as the p th root of n . As a result, the square root, or second root, of n is equal to $n^{1/2}$.

As the previous list shows, the special definitions for negative or fractional powers follow naturally from just trying to make them consistent with the definition for integer powers. In fact, you can define a power function for all kinds of things, including imaginary numbers, matrices, and even functions themselves.

One warning is appropriate, however. Within the realm of real numbers, fractional powers of negative numbers tend to be a problem. Consider, for example, that there is no square root of -1 . On the other hand, there is a cube root (which is also $-1!$). In general, computers and calculators tend to avoid difficulties by not letting you even try to take the square root of -1 . If you were to try to take $\text{power}(-1.1/3.0)$, a problem that does not have an answer, it won't let you. Given that the computer has no native way to represent the exact fraction $\frac{1}{3}$, this makes sense. If you need to find the cube root (or any other root) of a negative number, find the root of its absolute value and check it. Here is a pseudocode approach to accomplishing this:

```
function mthRoot(n, m)
    if n<0 then
        set p to power(-n, 1.0/m)
        if abs(power(-p, m)-n)<0.1 then return -p
        otherwise return "No such value"
    otherwise
        return power(n,1.0/m)
    end if
end function
```

To account for likely rounding errors, this function uses the trick of checking for closeness rather than equality of floats.

The Number e and the $\text{exp}()$ Function

Another word for power is *exponential*, a term already encountered. This word is used to describe the behavior of powers in general, rather than any particular power. For example, you speak of an “exponential function,” or a value “growing exponentially.” When working with exponentials, you’re likely to come across the number e , which is approximately 2.718. This number has a special property that will be dealt with at length in a later chapter, but for now just think of it as a number. It happens to be equal to the infinite sum $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$, as well as a lot of other rather pretty patterns. (The exclamation points indicate the factorial function $n! = 1 \times 2 \times \dots \times n$, with $0!$ defined to be 1.)

From a programmer’s point of view, the number e is most likely to be encountered in the `exp()` function, where `exp(x)` equals e^x . [To get the number e itself, just enter `exp(1)`.] It’s a kind of “standard” for exponentials.

Note

Don’t confuse the number e with the “exponential” notation you saw in Chapter 1, such as `1.53 e6`. There “ e ” is just a label that tells us how many decimal places to shift. Mathematically, it’s equivalent to multiplying the mantissa by a power of the current base—in this case, assuming you are in decimal, it is 1.53×10^6 .

Exponential Functions in Real Life and Physics

Mary has a Russian Vine in her back garden, which increases in length by 20% every week. This week it is 1 m long. If it keeps growing at the same rate, how long will it be in a year’s time?

You may recognize this problem as similar to the mortgage calculations reviewed earlier in the chapter. As became evident with those problems, any time that a value is multiplied by a constant increment over time, it rapidly changes. This is called *exponential growth*, and it’s commonly encountered in real-world problems concerning growth of populations, organisms, or economies.

After one week, Mary’s vine is 1.2 m long. After two weeks, it is $1.2 \times 1.2 = 1.44$ m long. After 52 weeks, it will be $1.252 = 13104.63$ m long—over 13 km! This explains the reason for the term “population explosion.” Exponential growth is very fast, and even the smallest growth rate can add up to a lot. If it only increased by 5% every week, it would still reach over 12 m in a year.

When something is not growing but shrinking by a constant factor, you have *exponential decay*. This is most familiar from the area of radioactivity, where a radioactive element has a so-called “half-life,” which is the length of time it takes for half of any particular sample to decay. As exponential growth is fast, however, exponential decay is slow. In fact, in a sense, it is infinitely slow, because it will never finish altogether. If you keep multiplying a number by $\frac{1}{2}$, it becomes smaller and smaller, but it will never reach zero. Still, it will get as close to it as you like, and it is said, mathematically, to tend to (or approach) a limit of 0.

People often become confused about radioactive decay. When they speak of a radioactive element having a half-life of several hundred (or thousand) years, they forget that this means it is not very radioactive. It is decaying slowly. And the element still will not be very radioactive a long time from now. Conversely, elements that are highly radioactive have a short half-life, so that while they are harmful in the short term, they soon decay and become relatively harmless. The most dangerous elements are those with intermediate half-lives, such as strontium-90, with a half-life around 30 years.

Logarithms

The flip-side of the exponential function is the logarithm, a useful tool for dealing with very large numbers.

Calculating with Logarithms

A *logarithm* is the inverse of an exponential. If $a = b^c$, then you say $c = \log_b(a)$. This is pronounced as, “the logarithm to base b of a ,” or “log to base b of a .” It is the power you have to raise b to in order to get the answer a .

Logarithms are generally available on the computer as one or more of the functions $\log_e()$ and are sometimes referred to simply as $\log()$. Usually $\log_e()$ is also expressed as $\ln()$, which is pronounced “natural logarithm.”

Most calculations with logarithms are direct counterparts of calculations with powers:

- **$\log(a) + \log(b) = \log(a \times b)$ [to any base]**

If $n^p = a$ and $n^q = b$, then $n^{p+q} = n^p \times n^q = a \times b$. So $\log(a \times b) = p + q$, and since $\log(a) = p$ and $\log(b) = q$, you have $\log(a) + \log(b) = \log(a \times b)$.

- **$k \times \log(a) = \log(a^k)$ [to any base]**

This follows fairly naturally from the previous result, but you can also derive it directly. For instance, suppose $n^p = a$. Then $a^k = (n^p)^k = n^{p \times k}$. So $\log(a^k) = p \times k$, and since $p = \log(a)$, this means that $k \times \log(a) = \log(a^k)$.

- **$\log_a(a) = 1$**

This is a consequence of the definition of a logarithm and the fact that $a^1 = a$.

- $\log_b(a) = \frac{1}{\log_a(b)}$

This result follows from the previous two.

Substituting $\log_b(a)$ for k in the previous formula, you arrive at $\log_b(a) \times \log_a(b) = \log_a(b^{\log_b(a)})$. By the definition of a logarithm, $b^{\log_b(a)}$ is simply a , so $\log_b(a) \times \log_a(b) = \log_a(a) = 1$.

- $\log_a(n) \times \log_b(a) = \log_b(n)$

This is a more general case of the previous result, and comes from the same logic: $\log_a(n) \times \log_b(a) = \log_b(a^{\log_a(n)}) = \log_b(n)$.

- $\log(1) = 0$ [to any base]

Any number raised to the power 0 is equal to 1 (apart from zero, which is undefined).

- $\log(0) = ?$

Actually, $\log(0)$ is undefined—you cannot raise a number (other than zero) to any power and get the answer 0. Your computer will probably spit out an error message if you try it.

Warnings similar to those concerning fractional powers of negative numbers also apply here. The log function is not well-defined over the negative numbers and should be avoided.

Using Logarithms to Simplify Calculations

Before the arrival of pocket calculators and computers, logarithms were one of the most important tools used by mathematicians, scientists, and engineers. As Figure 2.5 illustrates, the fundamental tool was the slide rule, which showed two sets of numbers drawn in a logarithmic scale.

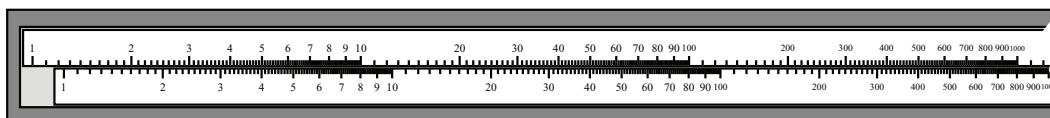


Figure 2.5

A slide rule. The lower scale can be slid back and forward against the upper one to perform calculations.

What exactly is a logarithmic scale? If you look at Figure 2.5, you can see that the numbers are not spread evenly. They are bunched up at one end and more spread apart at the other. It is called a logarithmic scale because, if you take the value at each point on the ruler and replace it with its own logarithm to some base, the resulting ruler is linear. Equivalently, a point n centimeters along the rule represents the value p^n , for some base p that depends on the scale.

If you measure the distance on the scale between, say, 1 and 10, it is the same as the distance between 10 and 100. In fact, this is the primary feature of a logarithmic scale. If $a = q \times b$ and $c = q \times d$ for some q , then the distance between a and b on the scale is the same as the distance between c and d . Compare this to the normal linear scale, which has the same property for addition instead of multiplication. If $a = q + b$ and $c = q + d$, then the distance between a and b is the same as the distance between c and d . Basically, a slide rule is to multiplication what a ruler is to addition.

This property is useful because it means that you can easily multiply two numbers together using the slide rule. For example, as shown in Figure 2.6, if you want to multiply 12.45 by 37.6, you find the point 12.45 on the top scale and slide the lower scale so that the inner 1 is matched to it.

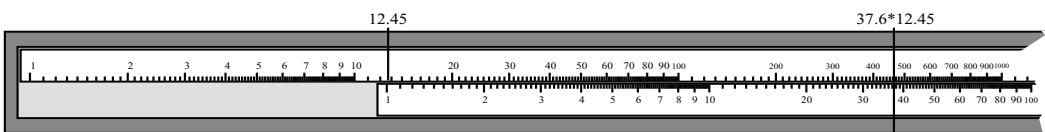


Figure 2.6
A slide rule in action.

Next, you locate the position of 37.6 on the lower scale and find the point that corresponds to it on the upper one. This point is marked in Figure 2.6. It is 468.12, the correct product. This works because the distance between 1 and 37.6 on one scale is equal to the distance between 1×12.45 and 37.6×12.45 on the other. Can you see how this follows from the definition of the logarithm? (See Exercise 2.3.)

Using Logarithms to Deal with Large Numbers

While slide rules are no longer standard issue in classrooms, the principle behind them remains important. This principle is that logarithms are a handy tool for dealing with large numbers. By applying a logarithm, you can bring enormously large numbers down to a more negotiable scale. As an illustration, consider that on a linear scale the value 5,000,000 is a thousand times greater than 5000. When working in the logarithmic base of 10, it is only three units greater.

This way of thinking is intuitively obvious. You have already used it when describing numbers in scientific notation. With the numbers just mentioned, 5,000,000 is represented in base 10 as 5×10^6 and 5000 as 5×10^3 . In fact, the word “base” for both the logarithmic scale and the base notation looked at in Chapter 1 is a marker for how similar the two concepts are. As with the slide rule, it is a natural way of thinking when dealing with a multiplicative world instead of an additive one.

Note

As will be seen in a later chapter, our ears do this, too. Both volume and pitch are interpreted by our brains using a logarithmic scale. The C below middle C has a frequency half that of middle C, and the C above that has twice the frequency again. Still, you hear them as being the same “distance” apart.

Logarithms are useful for dealing with problems involving powers. Looking back at Mary’s Russian Vine, if you recall, grows by 20% each week (each week its length is multiplied by 1.2), it’s reasonable to ask how many weeks it will be before the vine is long enough to encircle the world. The Earth has a circumference of approximately 40,000 km, which means that you want to know how many times you have to multiply 1 m by 1.2 to get an answer of 40,000,000 m.

Note

The circumference of the Earth has been known (approximately) since the time of the Greeks. It was first calculated accurately by Eratosthenes, creator of the Sieve glimpsed earlier in this chapter.

This is a simple logarithmic calculation. If you know that $1.2^p = 40000000$, then $p = \log_{12}(40000000)$. Assuming you have a computer that can calculate logarithms to base e —the $\ln()$ function—you must convert the base 1.2 calculation to the base of e . To accomplish this, use this approach:

$$\begin{aligned}\log_{1.2}(40000000) &= \log_{1.2}(e) \times \log_e(40000000) \\ &= \frac{\log_e(40000000)}{\log_e(1.2)}\end{aligned}$$

which is approximately 96 weeks, just under two years. To check the answer, find $\text{power}(1.2, 96)$. The answer is 40000000 (or thereabouts), as desired.

Exercises

EXERCISE 2.1

Make a set of scrollbar functions for scrolling a long piece of text within a window.

Imagine that you have a piece of text and a window in which to display it. Write functions that will use the length of the text and the height of the window to calculate the position of the scrollbar given a location in the text, and vice versa, as well as the size of the scrolling element.

EXERCISE 2.2

Write a compound interest function. The function should produce output similar to the examples in the text. You should be able to give it an amount of money and an interest rate, as well as an optional repayment value, and have it tell you the amount owed at the end of various periods of time.

EXERCISE 2.3

Write a slide rule function for calculating with large numbers. Follow the description in the text to make a function that uses the same principles as the slide rule to multiply any two numbers. If you are feeling frisky, you could even include a nice graphical slider, too.

Summary

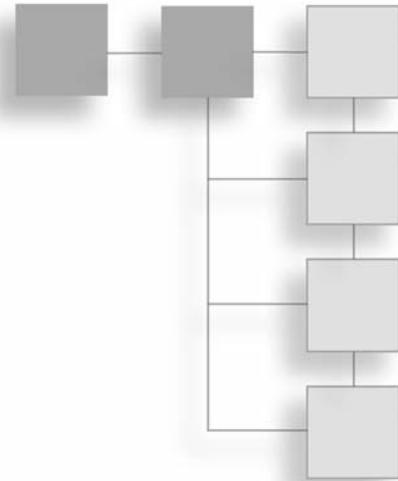
In this chapter, you have explored various more and less advanced areas of numerical mathematics to play with various techniques that are worth having in your mathematical toolkit. In the next chapter, you'll look at the second main area of basic mathematics, algebra.

You Should Now Know

- How to make calculations with *fractions*, *ratios* and *percentages*
- The concept of an *interest rate* and how it can be calculated
- The workings of the `mod()` function and how to use it to calculate with loops and integer divisions
- The concepts of *factor*, *prime number*, GCD and LCM, and how to use Euclid's Algorithm to calculate the GDC of two integers
- The meanings of *proportion* and *ratio*, and how to use them to map between values and sizes, as well as create a graphical representation of a range
- How to use *exponentials* and *logarithms* to deal with large numbers and multiplicative functions

CHAPTER 3

ALGEBRA



In This Chapter

- Overview
- Basic Algebra
- Working with Equations
- Factoring and Solving Quadratic Equations
- Functions and Graphs

Overview

This is a second chapter on fundamentals. In this chapter, the central concern is the principles of algebra, a branch of mathematics that deals with variables. Consideration is first given to the basic principles of algebra. After that, specific techniques are explored. The explorations include methods for using graphs to visualize equations and functions.

Basic Algebra

This section reviews a number of terms used in algebra. The terminology of algebra is extensive and essential for all areas of mathematics that are built on it. Given a beginning with terminology, the stage is then set for discussions of equations and other topics.

Variables, Parameters, and Constants

You represent two basic types of value in algebra. One is called a *constant*. The other is called a *variable*. Both of these values are usually represented as letters. Most people are familiar with them as the letters a , b , c , or x , y , z . A constant does not change. A variable does. Various names are applied to constants and variables depending on how they are used. The following list reviews the primary uses of variables and constants.

- **Constant.** A constant is a value that does not change. Examples are the constants e and φ . Likewise, values as represented by such symbols “1” and “2” are also constants. A constant can be an arbitrarily assigned value, as when you temporarily define the letter A to stand for the expression $\sqrt{2} + 1$.
- **Parameter.** A parameter is a value that defines a family of similar mathematical objects. For example, in the linear equation $y = m \times x + c$, the parameters m and c are used in all examples of this equation. The values substituted for m and c always convey the same type of information. You know what the value stands for.
- **Unknown.** An unknown is a token that represents a value that you do not know. For example, if you know that x is a particular number, and that $x + 3 = 4$, then you can calculate the (unknown) value of x .
- **Variable.** A variable represents something that can be of any value. This is like an argument passed to a function in the computer. For example, you can write a function that calculates the cube root of some number n , and n in this function is a variable.

You’ll notice that these terms are a little fuzzy. After all, in the equation $x + 3 = 4$, you can call x an unknown. On the other hand, it is also a variable, for you can substitute a value into it (1). If you consider the equation at length, you also see that the value that you substitute for x is always the same, so x is a constant.

The easiest way to think of these terms is as a hierarchy of variability. If you have an expression with a lot of letters, say $u + a \times t$, you can decide that one or more of these are “fixed,” and the others are “variable.” The fixed terms might be called parameters. For a particular set of parameters, you have a particular behavior for the variable terms. Then you might fix one of the parameters even more strongly and call it a constant. Each choice of a constant gives you a family of families. Finally, you can vary the constant to give you a family of families of families!

Variables and constants are used in similar ways by mathematicians and programmers. There is one difference that stands out, however. Programmers are encouraged to create mnemonically significant variable and constant names. For example, rather than naming a variable in a program x , a programmer is encouraged to use a name such as `num0fP1ayers` or `accountNumber`. Likewise, due to language conventions, programming languages might represent the Greek letter π with the characters `PI`. Constants, generally, are represented by words consisting of capital letters.

Expressions and Terms

A mathematical combination of variables or constants is called an *expression*. For example, $(a \times x) + (x^2 \times 4) + 3$ is an expression using the variables a and x , and the constants 3 and 4. Another constant also appears. This is the 2 used as an exponent, which is shorthand for the expression $x \times x$.

Note

As in manual evaluations, most compilers evaluate mathematical expressions in a standard order. First, expressions in parentheses are evaluated recursively. Next, other operators are evaluated sequentially: parentheses, division and multiplication, subtraction and addition. Thus the expression $1 + (2 \times 3 - 4) \times (-5 + 6)$ evaluates to $1 + 2 \times 1$, or 3. Expressions used as the numerators or denominators of fractions are considered to be in parentheses.

A *term* is any subexpression of the main expression that does not contain additions or subtractions. It is convenient to group expressions into terms. With $(a \times x) + (x^2 \times 4) + 3$, if you consider x to be a variable and a to be a constant, then there are three terms, $a \times x$, $x^2 \times 4$, and 3. A term can contain any number of variables multiplied by a constant, which is called the *coefficient*. In the term $x^2 \times 4$, 4 is the coefficient of x^2 .

Terms can be classified according to the exponents of the variables within them. The terms $3 \times x^2$ and $-2 \times x^2$ are considered *like terms* because they both have the same exponent of the variable x , namely 2. On the other hand, $3 \times x^2$ and $3 \times x$ are not like terms, because they contain different powers of x . Two terms are alike if they only differ by their coefficient.

To extend the discussion, consider the expressions $2x$ and $p(1 + q)$. In both expressions, two or more values are written next to one another. Such a construction indicates that a multiplication is to be performed. As a general rule, a coefficient is always written before the variables ($2x$, not $x2$), and terms outside parentheses are written first $p(1 + q)$, not $(1 + q)p$. A term is usually written as a single string without multiplication signs, so the expression $a \times x + x^2 \times 4 + 3$ is written as $ax + 4x^2 + 3$. This allows you to easily group terms according to the exponents of their variables. For example, you see the x term, the x^2 term, and then the constant.

Functions

A *function* is a map that takes values from one set, called the *domain* of the function, and transforms them into values from the same or another set, called the *range* of the function. Consider a situation in which rational numbers are transformed into integers. Functions in mathematics are usually represented as a single letter followed by open and close parentheses between which the parameter of the function is represented. For instance, the equation $y = f(x)$ indicates that the value x in the domain is mapped to the value y in the range. Functions are usually read as “ f of x .” The value of the range is a *function of* the value of the domain. If you have a function $f(x)$, then you refer to the value $f(3)$ as the *result* of substituting 3 for x in the function. The result of substituting 3 for x in the function $x \rightarrow x(x + 2)$ is the value $3 \times (3 + 2) = 3 \times 5 = 15$. (The arrow is a logical symbol for “ x maps to.”)

What applies to math also applies to programming. Consider the `floor()` function, introduced in Chapter 1. When you use this function, you feed it a number that is not an integer (3.45, for example). It then returns an integer (3). You can also have Boolean functions, which map their input to just two values, 0 and 1. As an example, one common Boolean function is named `isPrime()`. This function returns 1 if the input is a prime number and 0 otherwise. In *typed* computer languages, you generally need to specify in advance both the type of the input value of a function and the type of the output value. For example, you designate that the function requires an integer argument as input and returns a *double* (a floating-point value). This requirement formalizes the fundamental nature of functions, which requires that they draw from a specific domain and translate to a specific range.

Representations

Many mathematical functions have their own symbols, and these symbols are represented in program functions. The $\sqrt{}$ symbol represents the `sqrt()` function. Some mathematical functions directly correspond to program functions. Among these are `sin()`, `cos()`, and `tan()`. In other instances, the correspondence is not so clear. Consider the function $x^2 + 2$. This function might be formally expressed like this:

$$\text{squarePlusTwo}(x: \mathbb{R} \rightarrow \mathbb{R}) = x^2 + 2$$

The $\mathbb{R} \rightarrow \mathbb{R}$ part of this formulation is required in a formal definition of a function. It is not necessary to include it in this context, however. In this case, all it means is that both the input and output of the function are real numbers.

Note

The second \mathbb{R} could be replaced with a $>\mathbb{R}^+$, signifying the set of positive real numbers, since you know that the output of the function is always positive. Even more accurately, it could be said that the range of the function is the set $[2, \infty]$. This is a common notation for the set of real numbers from 2 to infinity. In this case, the function is shown to map to the whole of the range.

One-to-One, Inverse, and Multivalued

If you have a function $f(x)$ and there are no two distinct input values a and b in its domain such that $f(a) = f(b)$, then you call the function *one-to-one*. For such functions, there is also an *inverse function*, sometimes expressed as f' . With an inverse function, for every a in the function's complete domain, $f'(f(a)) = a$. For example, the cube function, $x \rightarrow x^3$, is one-to-one. On the other hand, the square function, $x \rightarrow x^2$, is only one-to-one when considered over the domain of positive real numbers (a partial or limited domain). A function for which values in the domain map to the same value in the range is called *many-to-one*.

Functions are *multivalued* when there is at least one value in the range that can map to more than one value in the domain. Among these is the mathematical square root function. With this function, both 1 and -1 are square roots of 1. Multivalued functions are usually the inverses of many-to-one functions, as in the case of the square root function. Strictly speaking, these are not functions at all, but it is useful to refer to them as functions. As a further point, it is impossible for self-contained functions programmed using standard programming languages to be multivalued. For example, the `sqrt()` function is a regular one-to-one function from \mathbb{R}^- to \mathbb{R}^+ , and it always returns a positive square root.

Polynomials

A *polynomial* is a particular kind of function of the form $x \rightarrow a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_nx^n$, where the values a_0, a_1, a_2, \dots are all real numbers. Polynomials are distinguished according to *degree*. The function $x \rightarrow 2x + 1$ is a polynomial. It is called a *linear* or *first-degree* polynomial because the highest power of x is 1. The function $x \rightarrow 2 - x + 3x^2$ is a *quadratic* or *second-degree* polynomial because the highest power of x is 2. It is also possible to have polynomials in more than one variable, such as $x \rightarrow x^2 + 2xy + y^2$.

Note

Discussion of functions has been formalized so far in this section, but a degree of informality makes things easier to communicate. From now on, such expressions as $x^2 + 1$ will be referred to as functions. This removes the need for expressing functions using the mapping notation of mathematics: $x \rightarrow x^2 + 1$.

Equations, Formulas, and Inequalities

An *equation* is like a mathematical sentence. It states that one expression is equal to another. Consider $1 = 1$. This is a simple and rather obvious equation, as is $2 + 2 = 4$. In most contexts, the term *equation* refers to a sentence that includes a variable, such as $x + 2 = 5$. This is a sentence that tells you a fact about the unknown x . In this case, it tells you enough about it that if you know that the equation is true, you can deduce what the value of x must be, namely, $x = 3$. The process of deduction is the main focus of elementary algebra.

It is important to distinguish between a function, an expression, and an equation. A function is like a phrase with blanks in it. It might read, “The ___ with tall ___’s.” An expression goes a step farther. It fills in the blanks with dummy variables: “The *nnn* with tall *mmm*’s.” An equation then tells you something about this expression by relating it to another. “The *nnn* with tall *mmm*’s is a *qqq*.”

Of the three activities, only an equation can be true or false. In this case, you don’t know whether it is true or not because you don’t know what an *nnn*, an *mmm*, or a *qqq* are. But if the sentence were “an *nnn* is an *nnn*,” then you could know it is true, whatever an *nnn* might turn out to be. Such an equation is called a *tautology*. For example, $x + 2 = x + 3 - 1$ is a tautology because it is true no matter what the value of x is. The equation $x + 1 = 2$ is not a tautology, however. If $x = 1$, then the equation is true. If x is any other value, then it is false. The same happens in a program. If you are working with a variable

x , and you have the line, `if x + 2 = 3, then` the program executes the next line if and only if $x = 1$ —that is, if the equation is true. Reversing the reasoning, if you know the equation is true, then you know the value of x must be 1.

A *formula* is an equation with more than one variable, and it defines one variable in terms of others. For example, $v = u + at$ is a formula that defines the value v in terms of the variables u , a and t . However, since formulas and equations can be treated identically, this is just a terminological distinction. Formulas are most often used to express relationships between physical values, such as $distance = speed \times time$.

An *inequality* is like an equation, but it tells you something other than that the two equations are equal. The types of inequalities vary fairly extensively. An inequality might say that one expression is less than the other. Consider, for example, $x + 1 < x + 2$. This happens to be a tautological inequality, for it always true, no matter what the value of x . An inequality might also say that one expression is greater than another. Consider, for example $x + 1 > y$. This is an inequality that might or might not be true, depending on the values of x and y . On the other hand, an inequality might simply assert that two expressions are not the same. In a computer language, you might write $x \neq y$ or $x \leftrightarrow y$. Mathematically, you usually write $x \neq y$.

Other symbols for inequalities are compounded or more abstract. Consider \leq and \geq , representing “less than or equal to” and “greater than or equal to.” Still another symbol, not asserting inequality but instead a limited form of equality, is \approx , meaning “approximately equal to.” There is also the congruency symbol \equiv . The latter two are not generally referred to as inequalities, but neutrally as statements.

Working with Equations

Algebra allows you to make deductions about unknown quantities by manipulating them as symbols rather than calculating with them directly. This section reviews the primary techniques involved in algebraic manipulations.

Balancing Equations

An equation is said to be like a set of scales. If you know that a pair of scales balances, then you know that if you add or take away the same amount from each side of the scales, the scales will still balance. The same is true of an equation. The two expressions on each side of the equals sign are equal by definition. If you add the same value to both sides, they will still be equal. In fact, if you apply any non-multivalued function to both sides of an equation, the equation remains true.

Why specify that the function must not be multivalued? Imagine that you have the equation $a^2 = b^2$. It is tempting to take the square root of both sides of the equation, yielding $a = b$, but this is not necessarily true. It is not true because if $a = 2$ and $b = -2$, then the first equation is true but the second is not. This is because the function $x \rightarrow \sqrt{x}$ is multivalued. Two distinct values in its domain can map to the same value in the range. All you can say with safety is that $a = \pm b$, where the \pm symbol means “plus or minus.”

The process of solving an equation in an unknown involves performing operations on both sides of the equation to find the value of the unknown. The most general equation is something like $f(x) = g(x)$ for some functions f and g . If you subtract $g(x)$ from both sides of the equation, you have $f(x) - g(x) = 0$. This can be as a new function $q(x)$, where $q(x) = f(x) - g(x)$. Generally, then, in its simplest terms, solving an equation is trying to find the inverse of the function q at 0.

Consider a less theoretical example. Suppose the equation is $2x + 3 = 7$. Here, the left-hand side of the equation has two terms. One term contains x . The other provides a constant. To solve the equation, you can begin by subtracting the constant from both sides:

$$\begin{aligned}2x + 3 &= 7 \\2x + 3 - 3 &= 7 - 3 \\2x &= 4\end{aligned}$$

Now you have an equation with one term on each side. On the left-hand side is a multiple of x . On the right-hand side is a constant, 4. Now, to isolate the value of x , divide both sides by 2:

$$\begin{aligned}2x &= 4 \\ \frac{2x}{2} &= \frac{4}{2} \\x &= 2\end{aligned}$$

Using this process, you find the value of x , which is 2. If you check the work, you find that $2 \times 2 + 3$ is indeed equal to 7.

As was emphasized previously, you can think of $2 \times 2 + 3$ differently, in terms of functions. If you do this, you have a function $x \rightarrow 2x + 3$, which presents a good starting place for exploring inversion. Since the function multiplies by 2 and then adds 3, the inverse is a process of subtracting 3 and then dividing by 2. This amounts to the reverse steps in the reverse order. The inverse function is $x \rightarrow (x - 3)/2$. If you apply this function to both

sides of the equation, the left-hand side maps to x (by the construction of the inverse function), and the right-hand side maps to $(7 - 3)/2 = 4/2 = 2$. This activity is the same as what was done previously, but the reasoning behind it is a little different.

Simplification

It isn't always so easy to solve an equation through inversion. Consider the equation $\frac{x+1}{2(x-1)-3(2-x)-x} = 3x - 8$. You have a bit of work to do before you can isolate the value of x . To do so, you must resort to *simplification*. To simplify an equation (or function), you reorganize it so that it is in successively simpler forms. In the end, you reach the simplest form. What constitutes the simplest form depends on the circumstances, but there are a few processes that you can apply regardless of circumstances. Here is a short list:

- **Group like terms.** Like terms are terms that contain the same combination of variables, differing only by the coefficient. With $2x + 3x = 5x$, two terms $2x$ and $3x$ are like terms and so can be combined into one.
- **Cross-multiply.** Drawing from the discussion in Chapter 2, the expression

$$\frac{x}{x+1} + \frac{x+2}{x+3}$$

can be simplified to

$$\frac{x(x+3)+(x+2)(x+1)}{(x+1)(x+3)}$$

This might seem involved, but if you substitute 4 for x , it is the same process as in

$$\frac{4}{5} + \frac{6}{7} = \frac{4 \times 7 + 6 \times 5}{5 \times 7}$$

- **Remove fractions.** If you have a fraction on one side of the equation, to remove the fraction, multiply both sides of the equation by the denominator. Using this approach, $\frac{x}{x+1} = 2$ becomes $x = 2(x+1)$.

- **Multiply out parentheses.** If you have a product of one element with an expression in parentheses, multiply the outside element by each element inside the expression to create an expression with no parentheses. Here are a few examples of this:

a. $2(x + 3) = 2 \times x + 2 \times 3$

$$= 2x + 6$$

b. $3x(2 - x + 4x^2) = 3x \times 2 - 3x \times x + 3x \times 4x^2$

$$= 6x - 3x^2 + 12x^3$$

c. $5 - 2(2x + 1) = 5 - 2 \times 2x - 2 \times 1$

$$= 5 - 4x - 2$$

$$= 3 - 4x$$

Negatives and Substitution

The third example (c) in the previous list proves troublesome due to the minus sign in the first expression of the equation. Items within the parentheses are multiplied by -2 . When multiplying values within parentheses by a negative number, the negative number applies to all values inside the parentheses. To explore a different view of this topic, suppose that the original expression had been written as $5 + (-2) \times (2x + 1)$. Comparing this version of the equation with the original allows you to see the effect of the negative sign.

Beyond negatives, you can also multiply together the values contained by parentheses. One approach to such a problem is to split it into two steps. Here is an example of how to proceed:

$$\begin{aligned}(x - 2)(2x + 3) &= x \times 2x - 2 \times 2x + x \times 3 - 2 \times 3 \\ &= 2x^2 - 4x + 3x - 6 \\ &= 2x^2 - x - 6\end{aligned}$$

To extend the discussion, if you encounter an expression that is common, you might be able to transform the equation into another form by defining a new variable. While this is an obscure technique, it is useful in complicated circumstances. For example, suppose you want to solve the equation $4x^4 - 9 = 0$. Notice that $4x^4 = (2x^2)^2$. To simplify the expression, you can substitute a variable, p , defining it as $p = 2x$. Since $4x^4 = p^2$, you have $p^2 - 9 = 0$, or $p^2 = 9$. Since the square root of 9 is 3, $p^2 = \pm 9$.

Now to find x , you simply substitute back into the definition. Accordingly, since $p = 2x^2$, $2x^2 = \pm 3$. Likewise, since $2x^2$ must be positive, you can ignore the negative square root of p . As a result, $2x^2 = 3$, so $x = \pm\sqrt{\frac{3}{2}}$. As already mentioned, this is an exceptional and difficult technique you encounter only on occasion. It is used, for example, later in this chapter with equations involving cubed values.

Solving the Original Problem

Look again at the equation with which this section began. It provides an occasion for applying some of the notions developed in the previous sections.

$$\frac{x+1}{2(x-1)-3(2-x)-x} = 3x-8$$

To solve this equation, begin by simplifying the denominator. Toward this end, multiply out the parentheses:

$$\frac{x+1}{2x-2-6+3x-x} = 3x-8$$

Now combine like terms in the denominator:

$$\frac{x+1}{2x+3x-x-2-6} = 3x-8$$

$$\frac{x+1}{4x-8} = 3x-8$$

To clean up the fraction, multiply both sides of the equation by the denominator. This gives you:

$$x+1 = (3x-8)(4x-8)$$

Now multiply out the parentheses on the right-hand side to get:

$$\begin{aligned} x+1 &= (3x-8)(4x-8) \\ &= 12x^2 - 32x - 24x + 64 \\ &= 12x^2 - 56x + 64 \end{aligned}$$

Finally, bring the terms from the left-hand side to the right-hand side and simplify again:

$$\begin{aligned} x+1 &= 12x^2 - 56x + 64 \\ 0 &= 12x^2 - 56x + 64 - x - 1 \\ 12x^2 - 57x + 63 &= 0 \end{aligned}$$

In the last line, the expressions have been switched (putting the 0 on the right). This is a traditional form of presenting an equation. When you have an equation with a constant on one side, you put the constant on the right. The practice corresponds to the convention in English of putting a proper noun before the verb in a sentence that contains a copulative verb. For example, speakers of English say, “Patch is a dog,” rather than “a dog is Patch.”

Factoring and Solving Quadratic Equations

Another simplification technique goes in almost the opposite direction of multiplying out the values in parentheses. Rather than multiplying out the values in parentheses, it simplifies the expression by finding *common factors*. The approach is the same as used in Chapter 2 when the `gcd()` function was applied to denominators to simplify addition of fractions. When working with plain numbers, this is called *factorization*, or more commonly, *factoring*.

In general, it’s best to leave factoring until you have already gone through the simplification process discussed in the previous section, joining all the like terms together and eliminating fractions with variables in the denominator. When working with an equation, it also a good idea to move all the terms over to one side, giving you an equation of the form $f(x) = 0$. Factoring, strictly speaking, is an operation performed on functions rather than equations.

Having expressed the equation as a function, you try to find any common factors between the terms. For example, in the expression $6x^2 - 15x + 9$, all the terms have a factor of 3. You can take out the factor of 3, making it a separate term, to get $3(2x^2 - 5x + 3)$.

Another example is $x^2 - 4x$. Here, both terms have a factor of x . As before, you factor the expression on the left-hand side of the equation by turning it into a product of the factor x . To express this change, you use parentheses. First you write x , then, in parentheses, you write the value of $x - 4$. This gives you $x(x - 4)$.

Note

If this isn’t clear, try multiplying together the terms inside and outside the parentheses to see what you end up with. It is the same process as in the previous case, but with a factor of x instead of the number 3.

You can use the factored version of a function to make it easier to solve equations. If you have $3(2x^2 - 5x + 3) = 0$, for example, you can divide both sides by 3 to get a simpler equation, $2x^2 - 5x + 3 = 0$. And if you have $x(x - 4) = 0$, then you have two unknown numbers (x and $x - 4$) whose product is zero. If the product of two numbers is zero, you know that either one or the other number must be zero. This means that you end up with either $x = 0$ or $x - 4 = 0$. The value of x , then, is either 0 or 4.

Examples of Factoring

Given the preliminary discussion, here are some specific examples of factoring:

- $12x^3 - 8x^2$ has a common factor of $4x^2$, because 4 is a factor of both 12 and 8. x^2 is a factor of both itself and x^3 . This makes it possible to isolate this factor, resulting in the expression $4x^2(3x + 2)$.
- $-2x - 3x^2$ has a common factor of $-x$. Since a negative number divided by a negative number is positive, factoring this expression you arrive at $-x(2 + 3x)$.
- $2xy^2z + 4x^2yz$ has a common factor of $2xyz$. Factoring, the outcome is $2xyz(y + 2x)$.
- $\frac{x}{2} - \frac{x^2}{4}$ has a common factor of $\frac{x}{4}$. Think of this as a two-stage process. The first involves simplification: $\frac{x}{2} - \frac{x^2}{4} = \frac{2x - x^2}{4}$. After further factoring, you end up with $\frac{2x - x^2}{4} = \frac{x}{4}(2 - x)$.

In addition to factors that are simple terms, whole expressions can be taken as factors. For example, in the expression $x(x + 1) + 3(x + 1)$, the expression $(x + 1)$ is a common factor, so the expression can be factored to $(x + 1)(x + 3)$.

Factors and Quadratics

Consider what happens if you multiply out the terms in parentheses of $(x + 1)(x + 3)$ and then join the like terms. This results in $x^2 + 4x + 3$. To factor this expression, given that you just performed the multiplication that created it, you can reverse the process and fairly easily reach its factored form. It is clear, however, that expressions such as $x^2 + 4x + 3$ are not as easily factored as expressions like $x(x + 1) + 3(x + 1)$.

The expression $x^2 + 4x + 3$ is a *quadratic* expression. A quadratic is any expression of the form $ax^2 + bx + c$. If a quadratic expression can be factored, it takes the form $(px + n)(qx + m)$.

Note

As a matter of review, in such generalized terms as $ax^2 + bx + c$ and $(px + n)(qx + m)$, a , b , p and q are place holders for coefficients. Unless its value is greater than 1, the coefficient is not usually represented by a literal expression. For example, you usually do not write $1x^2 + 1x + 2$ (where c , a constant, is equal to 2).

To develop a general pattern for simplifying quadratic equations, you can equate the two expressions using the equation $ax^2 + bx + c = (px + n)(qx + m)$.

If you multiply out the expression on the right-hand side and then combine terms, the equation assumes the following form:

$$\begin{aligned} ax^2 + bx + c &= px^2 + pmx + nx + nm \\ &= px^2 + (pm + n)x + nm \end{aligned}$$

This is an equation in which x is a variable, while a , b and c are parameters. For any possible value of x , given particular values of a , b and c , the equation is supposed to be true for any possible value of x . If this is to be so, each of the terms in x must match. As a result, if you assume that $p = 1$, you can reason as follows:

If $ax^2 = px^2$, then $a = p$.

Likewise, if $(pm + n)x = bx$, then $b = am + n$ (since $a = p$).

Therefore, $nm = c$.

Consider a case in which $a = 1$. The general form of the quadratic becomes $x^2 + bx + c$. As a result, moving to the corresponding expression, $n + m = b$, and $nm = c$. Further, n and m are two numbers whose product is c and whose sum is b .

How can you find these numbers? One approach is to use inspection. Here are a few examples:

- **$x^2 + 4x + 3$.** Since $b = 4$ and $c = 3$, you need numbers whose product is 3 and whose sum is 4. These numbers are 1 and 3. So the expression factors to $(x + 1)(x + 3)$. (This is the same as the result shown previously.)
- **$x^2 + 3x - 4$.** Since $b = 3$ and $c = -4$, you need numbers whose product is -4 and whose sum is 3. The numbers 4 and -1 meet this description. So the expression factors to $(x + 4)(x - 1)$.

- **$x^2 - 5x + 6$.** Since $b = -5$ and $c = 6$, you require numbers whose product is 6 and whose sum is -5 . The -2 and -3 meet this description. The expression factors to $(x - 2)(x - 3)$.
- **$x^2 - 5x - 6$.** Since $b = -5$ and $c = -6$, you need numbers whose product is -6 and whose sum is -5 . These numbers are -6 and 1 . The expression factors to $(x - 6)(x + 1)$.

Notice that the numbers you are looking for vary according to the sign of the numbers b and c . In the last two examples, the absolute values of b and c are the same, but because the sign of c is different, the answer is changed significantly.

Can you always find two numbers, n and m , that satisfy this requirement? The answer is no. For example, if $b = 1$ and $c = 1$, then no pair of numbers whose sum and product are both 1 exists. As a general rule, you can only factor a quadratic expression as long as $b^2 \geq 4ac$. This expression is called the *discriminant*. Why this is so is discussed shortly.

Given the quadratic form $ax^2 + bx + c$, what if a is not equal to 1? When this is so, you can reason things as follows:

$$b = am + n$$

$$nm = c, \text{ so } amn = ac$$

In this case, you are looking for two values, am and n , whose sum is b and whose product is ac . While the technique used to factor remains much the same, after you have found am , you need to divide it by a to find m .

As an example, consider an equation examined earlier: $12x^2 - 57x + 63 = 0$. To factor this equation, start by dividing the whole equation by a common factor, 3. This division results in $4x^2 - 19x + 21$. For this equation, $b = -19$, and $ac = 4 \times 21 = 84$. Consequently, you are looking for two numbers whose product is 84 and whose sum is -19 . These numbers are -7 and -12 . You can choose either one of these to be n and am . Since $a = 4$, however, and 4 is a factor of 12, it makes sense to choose $n = -7$, and $m = -3$. This way, it is not necessary to use fractions.

Given these preliminaries, you arrive at $4x^2 - 19x + 21 = (4x - 7)(x - 3)$. Working from the notion that the product must be zero, you also know that either $x - 3 = 0$ or $4x - 7 = 0$. Working these out, you find that either $x = 3$ or $x = \frac{7}{4}$. To test, substitute either of these values into the original equation and see if it evaluates correctly to 0.

The Quadratic Formula and the Difference of Two Squares

The most prevalent way that most people learn to solve quadratic equations involves using the *quadratic formula*, which reads, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. To work with this formula, notice that if $b^2 < 4ac$, then the value under the square root sign is negative. There is no real number solution for x in these circumstances. If $b^2 > 4ac$, you arrive at two values for x , one using the positive square root, the other using the negative. If $b^2 = 4ac$, then the quadratic equation is an *exact square*, and there is only one solution for x . This occurs in an equation such as $x^2 - 6x + 9 = 0$, which factors to $(x - 3)(x - 3)$, or $(x - 3)^2$.

In addition to those that factor to exact squares, another kind of quadratic expression involves the *difference of two squares*. Consider the expression $x^2 - 25$. This expression factors neatly as $(x - 5)(x + 5)$. How it factors provides a reliable model for approaching all such factorizations. If you want to multiply two numbers that differ by a small amount, you can do it by squaring the average of the two numbers and subtracting the square of half the difference. Here is an example:

$$\begin{aligned} 202 * 198 &= (200 + 2) * (200 - 2) \\ &= 200^2 - 2^2 \\ &= 40000 - 4 \\ &= 39996 \end{aligned}$$

Solving Cubic Equations

It's possible to solve higher-degree polynomials, too, although it involves a little more work. Consider, for example, *cubic* functions. A cubic function (or equation) always has at least one *root* (a value x such that $f(x) = 0$). It can have up to three. The standard form of a cubic function is $Ax^3 + Bx^2 + Cx + D = 0$. The term D is referred to as the *discriminant*.

As with quadratic equations, you can find a solution to the cubic equation by finding a simpler equation. For starters, you find a quadratic equation that you can solve using the methods previously mentioned. For example, start by dividing the equation by A to obtain a simpler equation $x^3 + ax^2 + bx + c = 0$. Then make a substitution, replacing the variable x with a new variable t such that $t = x + \frac{a}{3}$. This results in a new cubic equation with no quadratic term:

$$t^3 + 3pt + 2q = 0, \text{ where } p = \frac{b}{3} - \frac{a^2}{9} \text{ and } q = \frac{a^3}{27} - \frac{ab}{6} + \frac{c}{2}.$$

As before, the key to solving this cubic is the discriminant, D , where $D = p^3 + q^2$. Consider these notions:

- If $D > 0$, then the equation has exactly one real root, the value $r + s$, where $r = \sqrt[3]{-q + \sqrt{D}}$ and $s = \sqrt[3]{-q - \sqrt{D}}$.
- If $D = 0$, then the equation has two roots, $2 \times \sqrt[3]{-q}$ and $-\sqrt[3]{-q}$ (a double root).
- If $D < 0$, then you can find the three roots by using a trigonometric function. This is not covered until the next chapter, but for reference, the roots are given by

$$t = 2\sqrt{-p} \cos\theta, \text{ where } \frac{1}{3}(\cos^{-1})\left(\frac{-q}{\sqrt{-p^3}}\right)(-2\pi k) \text{ for } k = 0, 1 \text{ or } -1$$

Having found values for t , you can then transform these into values for x by taking $x = t - \frac{a}{3}$ again.

As you can see, this process is complicated. On the other hand, it is predictable enough to serve as the basis of an algorithm. From the algorithm, a reliable programming function can be generated. Here is one approach to such a function:

```
function solveCubic(a,b,c,d)
    // d is the coefficient of the cubic term, the default being 1
    if d is defined then divide a, b and c by d
    set p to b/3 - a*a/9
    set q to a*a*a/27-a*b/6 + c/2
    set disc to p*p*p + q*q
    if disc>=0 then
        set r to cubeRoot(-q+sqrt(disc))
        if disc=0 then set ret to [2*r, -r]
        otherwise
            set s to cubeRoot(-q-sqrt(disc))
            set ret to [r+s]
    end if
    otherwise
        set ang to acos(-q/sqrt(-p*p*p))
        set r to 2*sqrt(-p)
        set ret to an empty array
        repeat for k=-1 to 1
            set theta to (ang-2*pi*k)/3
            append r*cos(theta) to ret
```

```
    end repeat
end if
subtract a/3 from each element of ret
return ret
end function
```

Note

This algorithm is adapted from the method described by Eric Lengyel in *Mathematics for 3D Game Programming*. This is a particularly elegant formulation. Other methods that may seem to involve fewer miracles can be found on the Web, but they tend to require a knowledge of complex numbers.

Solving Simultaneous Equations

There are two approaches generally used to solve equations. One is by substitution. The other is by elimination. Both approaches are usually interchangeable, but there are many instances in which you are likely to prefer one over the other.

Simultaneous Solutions by Substitution

If you have more than one unknown in an equation, it is not generally possible to determine them both. For example, if $x + y = 5$, then x could be 1 and y could be 4, or x could be 10 and y could be -5 . An infinite number of other combinations are possible. However, if you have more information, such as another equation, it may be possible to find both unknowns. In general, if you have n unknowns, then you need n independent equations to find them. (An independent equation is one that cannot be deduced from the starting equation.) Fewer than n equations will not give you enough information, and more than n equations might give you too much information. If you have too little or too much information, you cannot solve the equations consistently.

A set of equations with the same unknowns is called a *set of simultaneous equations*. To explore this, first consider how you might solve a pair of simultaneous equations in two unknowns.

As a set of equations, suppose you start with the following:

1. $x + 3y = 10$
2. $5x - 2y = -1$

There are several ways to solve this set of equations. The first is by substitution. With substitution, you use one equation to find the value of one unknown as a function of the other. Then you substitute this value into the second. In this case, if you rearrange equation 1, you get $x = 10 - 3y$. If you then substitute this value into equation 2, you get

$$\begin{aligned} 5(10 - 3y) - 2y &= -1 \\ 50 - 15y - 2y &= -1 \\ -17y &= -51 \\ y &= 3 \end{aligned}$$

This process is called *eliminating x*. Generally, you use one equation in order to make a new equation for y that doesn't involve x . Having found the value of y , you use the function for x : $x = 10 - 3y = 10 - 9 = 1$. If you substitute these values for x and y back into equation 2, you get $5x - 2y = 5 - 6 = -1$.

The substitution method is most useful when dealing with non-linear equations. Such equations involve such terms as x^2 , y^2 or xy . To explore substitution further, consider this set of equations:

1. $3x + 2xy = 7$
2. $2x + 5y - y^2 = 8$

Examining the first equation, you can see that you can factor the left-hand side to get $x(3 + 2y) = 7$, which means that $x = \frac{7}{3+2y}$. You can now substitute this value into the second equation, which allows you to arrive at a cubic equation as follows:

$$\begin{aligned} 2\left(\frac{7}{3+2y}\right) + 5y - y^2 &= 8 \\ \frac{14 + 5y(3+2y) - y^2(3+2y)}{3+2y} &= 8 \\ 14 + 5y(3+2y) - y^2(3+2y) &= 8(3+2y) \\ 14 + 15y + 10y^2 - 3y^2 - 2y^3 &= 24 + 16y \\ 0 &= 2y^3 - 7y^2 + y + 10 \end{aligned}$$

Given that you have reached a cubic equation, you have a clear path to a solution. In this case, the solution is made simpler because you can see it by inspection. If you substitute the value $y = -1$ into the expression on the right, you get the answer, 0. This means that the expression $(y + 1)$ must be a factor. [This is an example of a general theorem, which says that if a is a root of a function $f(x)$, then $(x - 1)$ is a factor of the function. Recall that a root is expressed as $f(a) = 0$.]

To factor a polynomial expression when you know one of the factors is a simple task. You just take it one step at a time. Consider the equation arrived at so far, $2y^3 - 7y^2 + y + 10$. If you know one of the factors, then you can express the problem as $2y^3 - 7y^2 + y + 10 = (y + 1)(\dots)$. You then proceed to find the expression inside the parentheses. Toward this goal, the first term in the parentheses must be $2y^2$, and the first term when expanded will be $2y^3$. This means that the factors can be expressed as $(y + 1)(2y^2 + \dots)$.

If you try expanding this tentative expression, you get $2y^3 + 2y^2 + \dots$, but the term in y^2 should actually be $-7y^2$. You need another term, $-9y^2$, to make up the difference. This means that you now have the tentative factorization $(y + 1)(2y^2 - 9y + \dots)$. Again, if you expand this expression, you arrive at $2y^3 - 7y^2 + 9y + \dots$. Now you need to match the coefficient of y , which should be 1. To get this, you must add another $10y$ to the answer, which means your final factorization is $(y + 1)(2y^2 - 9y + 10)$. When you multiply out the terms in the parentheses, you get the answer.

Now you have a quadratic function, and you can factor it as before. To do this, you require two numbers whose product is 20 and whose sum is -9 . These are -5 and -4 . This gives you these factors: $(y + 1)(y - 2)(2y - 5)$.

You are almost there! From the previous factorization of the cubic equation, you now know that there are three possible values for y : $y = -1$, $y = 2$, or $y = \frac{5}{2}$. For each of these values, you can substitute back into equation 1, to get three possible pairs of values x and y : $(7, -1)$, $(1, 2)$, or $(\frac{7}{8}, \frac{5}{2})$. Substituting any of these pairs into equation 2, you arrive at the answer, 8.

If the process seems exhausting, one explanation is that the problem is more difficult than many others. Working with a difficult problem is worthwhile, however, because such a problem reveals how helpful the substitution method can be.

Simultaneous Solutions by Elimination

In addition to substitution, simultaneous equations can be solved using elimination. To see how this is so, consider a new pair of linear simultaneous equations in two variables:

1. $3x + 2y = 2$
2. $2x + 5y = 16$

While this system of equations could be solved using substitution, you can also add multiples of the equations together. Given that you know that $a = b$, and that $c = d$, you can add together linear sums of the equations and get a new equation, such as $2a + 3c = 2b + 3b$. This process can be used to eliminate a particular variable.

Accordingly, if you multiply the first equation by 2, you get

$$3. \quad 6x + 4y = 4$$

and if you multiply the second one by 3, you get

$$4. \quad 6x + 15y = 48$$

If you then subtract the first of these new equations from the second, the result is an equation in y , which is

$$\begin{aligned} 6x + 15y - 6x - 4y &= 48 - 4 \\ 11y &= 44 \\ y &= 4 \end{aligned}$$

You can then substitute this value back into equation 1, getting

$$\begin{aligned} 3x + 2 \times 4 &= 2 \\ 3x &= -6 \\ x &= -2 \end{aligned}$$

How do you determine the values used to multiply the two equations? It's the same process involved in finding a common denominator. If you want to eliminate the variable x , your goal is to put the two equations over the common denominator of the coefficients of x . In this case, the common denominator of 3 and 2 is 6. As with fractions, you then multiply each equation by the common denominator divided by the coefficient of x in the equation.

To explore this further, here is another example:

1. $3x + 10y = 2$
2. $5x + 6y = 14$

This time, you first eliminate y . The coefficients of y in the equations are 10 and 6. Given that the lowest common multiple of 10 and 6 is 30, you multiply Equation 1 by $\frac{30}{10} = 3$ and Equation 2 by $\frac{30}{6} = 5$. The results is two new equations:

$$3. \ 9x + 30y = 6$$

$$4. \ 25x + 30y = 70$$

Subtracting equation 3 from equation 4, you arrive at

$$25x - 9x = 70 - 6$$

$$16x = 64$$

$$x = 4$$

Substituting back in equation 1, you get

$$3 \times 4 + 10y = 2$$

$$12 + 10y = 2$$

$$10y = -10$$

$$y = -1$$

Check by substituting both values back into equation 2:

$$5 \times 4 + 6 \times (-1) = 20 - 6 = 14$$

A Function for Solving Systems of Equations

Elimination is used generally to solve any system of linear simultaneous equations. As a demonstration of how this is so, one approach is to develop an algorithm and write code to implement it.

Suppose you have n linear equations in n variables. You can write each equation as an $n + 1$ element array. For example, $2x + 3y = 3$ becomes $[2,3,3]$. The complete system of equations is then an array of n such arrays, which can be represented as a function parameter `simul`. To solve the system of equations, the following function can then be developed:

```
function solveSimultaneous(simul)
    set redux to an empty array
    set n to the number of elements of simul
    repeat for i=n down to 1
        repeat for j=i down to 1
            if simul[j][i] is not 0 then
                set row=simul[j] and quit this loop
            end if
        end repeat
        if no row found then return "no unique solution"
        divide row by row[i]
        add row to redux
        delete row from simul
        repeat for j=i-1 down to 1
            if simul[j][i] is not 0 then
                subtract row*simul[j][i] from simul[j]
            end if
        end repeat
    end repeat
    set output to an array with n elements
    repeat for i=n down to 1
        set sum to 0
        repeat for j=i+1 to n
            add redux[i][j]*output[j] to sum
        end repeat
        set output[i] to redux[i][n+1]-sum
    end repeat
    return output
end
```

The `solveSimultaneous()` function uses an algorithm that has two parts. First, it goes through the equations one by one, at each step picking an equation with a target coefficient. Say that you are working in three variables, x , y , z , and the current variable is x . Such an equation might be $2x + 4y + z = 8$. You divide this equation by the coefficient of x to give it a coefficient of 1: $x + 2y + \frac{z}{2} = 4$. The resulting equation is added to a list of reduced equations. (Such a list is called the *redux*.) Then you subtract multiples of the equation from all the remaining equations to eliminate the variable x from each.

Suppose one of the equations is $3x - z = 5$. You subtract 3 times the equation from it in order to eliminate x , getting $-6y - \frac{5z}{2} = -7$. While this specific step eliminates x from the equation, it has also brought y into it once again. This is not a problem. The goal is to eliminate x . At the end of the process, you have an equation in redux with a coefficient of 1 in x , and all the remaining equations in `simul` have a zero coefficient in x .

If at any stage you can't find an equation with a non-zero coefficient in the current variable, then you are stuck. The n equations are not independent. In this case, there are two possibilities: either there is no solution to the equations, or there are an infinite number of solutions.

When you repeat this process for y and z , you end up with a set of equations in redux with the following properties:

- The i th equation has zero coefficients for the first $(i - 1)$ variable.
- The i th equation has a 1 coefficient for the i th variable.

The second stage of the process uses this information to solve the equations. The solution arises from working backward. Notice that the final equation is simple. Since it tells you the value of the last variable, which is something like $z = 2$, you can write this number into the output. Now you look at the second-to-last equation, which is something like $y - 2z = 3$. Since you know the value of z , you can substitute it into the equation and quickly find y . In this case, y is 7. You keep working backward through the equations, and at each stage you find the current unknown by substituting for all the later variables.

Simultaneous equations come up a lot in physics, particularly the collision detection calculations. While they might serve as the topic of a much more extended discussion, however, at this point it is important to move on to another central concern of algebra, one that involves visualizing the behavior of functions.

Functions and Graphs

This section explores functions and how they can be used. One of the most prevalent uses of functions is to generate visualizations, and graphs are the primary form of visualization.

What Is a Graph?

A graph is a way of representing data visually. The standard form of graph is the two-dimensional *Cartesian* graph, which displays all possible ordered pairs of two numbers in the form of points drawn on a flat sheet or Cartesian plane. To make a Cartesian plane, you need three things: an *origin*, which is a single point that you define as representing the point $(0, 0)$, and two *axes*, (pronounced *ax-ease*, the plural of the word *axis*). An axis is a direction on the plane, and it is drawn as a line through the origin, with arrows to indicate the directions. The values plotted on the axes can be negative or positive. Those above or to the right of the origin are positive, and those to the left or below the origin are negative. The arrows indicate that the values in all directions continue to infinity. Figure 3.1 shows an empty Cartesian plane.

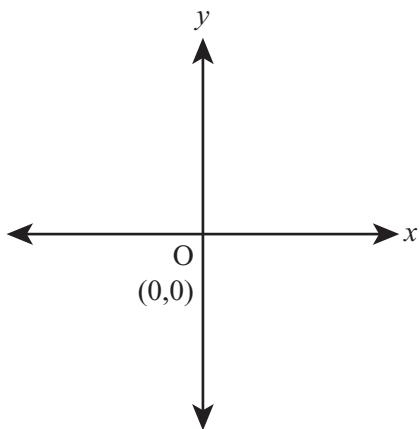
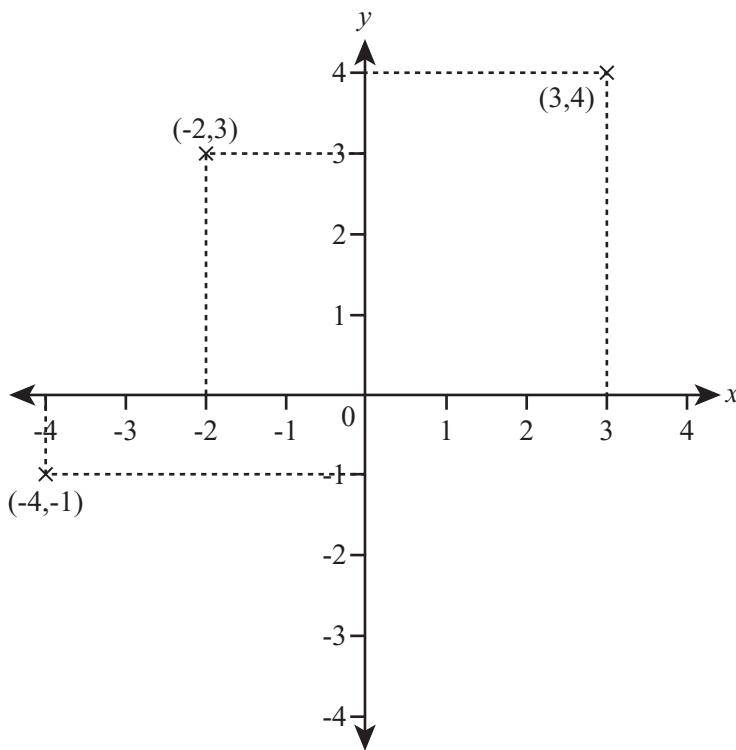


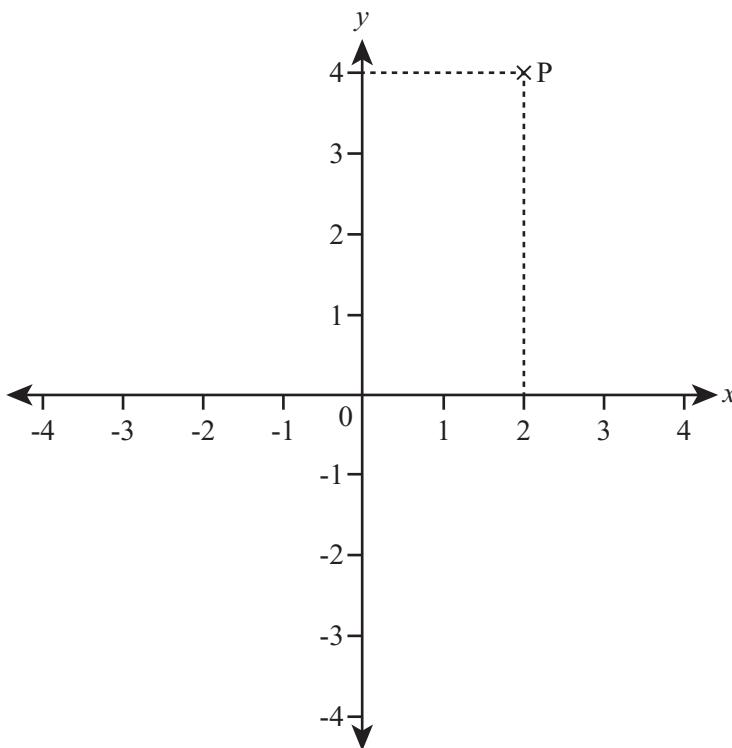
Figure 3.1
The Cartesian plane.

For the Cartesian plane to represent pairs of numbers, each axis must correspond to one number in the pair. While the horizontal axis represents the first number, the vertical axis represents the second. The two axes are usually labeled as x and y and drawn at right angles to one another. As illustrated by Figure 3.2, if you assign each axis a scale, then any pair of numbers (a, b) can be represented by measuring a distance a in the direction of the first axis, and a distance b in the direction of the second axis. This process is called *plotting* the point (a, b) , and the values a and b are called *coordinates*. If the axes are x and y , then a is the *x-coordinate*, and b is the *y-coordinate*.

**Figure 3.2**

The points $(3, 4)$, $(-2, 3)$ and $(-4, -1)$ plotted on the Cartesian plane.

You can also follow the opposite process, reading the value of a point P on the plane. If you draw a line through P , perpendicular to the x -axis, it has to meet the y -axis at some point Q . As illustrated by Figure 3.3, if you measure the distance from Q to P in the scale of the x -axis, this will give you the x -coordinate of P , and if you measure the distance from the origin (often abbreviated as O) to Q in the scale of the y -axis, this will give you the y -coordinate. Note that the point P has been marked, and P is the point $(2, 4)$.

**Figure 3.3**

Reading the point $(2,4)$ from a graph.

Plotting and Examining Functions on a Graph

Graphs are most useful for representing functions. If you have a function $f(x)$, then you can represent it on a graph by taking every possible value of x , finding the value of $f(x)$, and plotting the point $(x, f(x))$. Generally, you plot the variable along the horizontal axis and the output of the function on the vertical axis. A common expression of this is *plotting $f(x)$ against x* . Alternatively, you can label the vertical axis with y and say that you are plotting the graph of $y = f(x)$. If $f(x) = 2x + 1$, then you are plotting the graph of $y = 2x + 1$.

Note

In this chapter, functions in one variable are discussed. However, it is perfectly possible to make a graph of a function in more than one variable. Such graphs are difficult to create on a piece of paper. To draw a function in two variables, a three-dimensional graph (or surface) is required. Tools such as MATLAB make working with 3-dimensional graphs fairly easy. To draw a function with four or more variables, you need a graph with four or more dimensions. With these, also, mathematical software is extremely helpful.

It's fairly straightforward to make the computer draw a graph. The precise details depend on the classes or functions that a given programming language offers. The `drawGraph()` function draws a graph on the basis of another function passed to it as a parameter along with a range of values for x . Representing the number of evenly spaced values of x to be plotted, the `resolution` parameter is used to determine how accurately the graph is drawn. It will also automatically scale the y -axis to fit the entire function into the graph. The dimensions of the graph are passed in as the last two parameters, and the graph is always drawn to include both axes.

Note

The graphics functions of most programming languages represent point $(0,0)$ at the upper left of the screen and draw to the right and downward from it. This approach differs from how people are usually taught to draw graphs. In this `drawGraph()` function, these details are ignored; it simply plots a point or draws a line. To make it plot a line or figure relative to the origin of the axes, you must translate the value to the coordinate plane.

```
function drawGraph(functionToDraw, minX, maxX, resolution, width, height)
  // calculate values of the function
  set xValues to an empty array
  set yValues to an empty array
  set spacing to (maxX-minX) / (resolution - 1)
  // spacing is the distance between consecutive x values
  repeat for i = 0 to (resolution-1)
    set x to minX + i * spacing
    set y to calculateValue(functionToDraw(x))
    // how this is done depends on how you want to represent
    // the function. In the version on the CD-ROM, you can
    // pass either a string such as "x*x + 2*x + 3" or the
    // name of any function defined somewhere. The latter is
    // more flexible as it allows you to deal with special
```

```
// cases such as "undefined" (see later in the function)
append x to xValues
append y to yValues
end repeat

// calculate the scale of the graph
set leftX to min(minX, 0)
set rightX to max(maxX, 0)
// leftX and rightX are the x-values at each end of the
// x-axis to be drawn
set xScale to width / (rightX - leftX)
set topY to max(largest(yValues), 0)
set bottomY to min(smallest(yValues), 0)
// largest() and smallest() should return the largest and
// smallest values in the array respectively
set yScale to height / (topY-bottomY)

// draw axes
set x0 to xScale * (-leftX)
set y0 to yScale * (-bottomY)
// x0 and y0 are the positions within the graph of the axes
draw a line from the point (x0, 0) to the point (x0, height)
// this is the y-axis--you should also add arrows,
// a scale and labels here
draw a line from the point (0, y0) to the point (width, y0)
// this is the x-axis

// draw the function
set currentPoint to 0
repeat for i = 1 to the number of elements in xValues
  set x to xValues[i]
  set y to yValues[i]
  if y = "undefined" then
    set currentPoint to 0
  otherwise
    set thisPoint to ((x-leftX)* xScale, (y-bottomY) * yScale)
    if currentPoint = 0 then
      plot the point thisPoint
    otherwise
      draw a line from currentPoint to thisPoint
```

```

    end if
    set currentPoint to thisPoint
  end if
end repeat
end

```

Figure 3.4 provides samples of the output that `drawGraph()` function might generate. The version of the function used to generated Figure 3.4 provides labeling of the plots. You can access this function in the code for this chapter provided on the book's companion website.

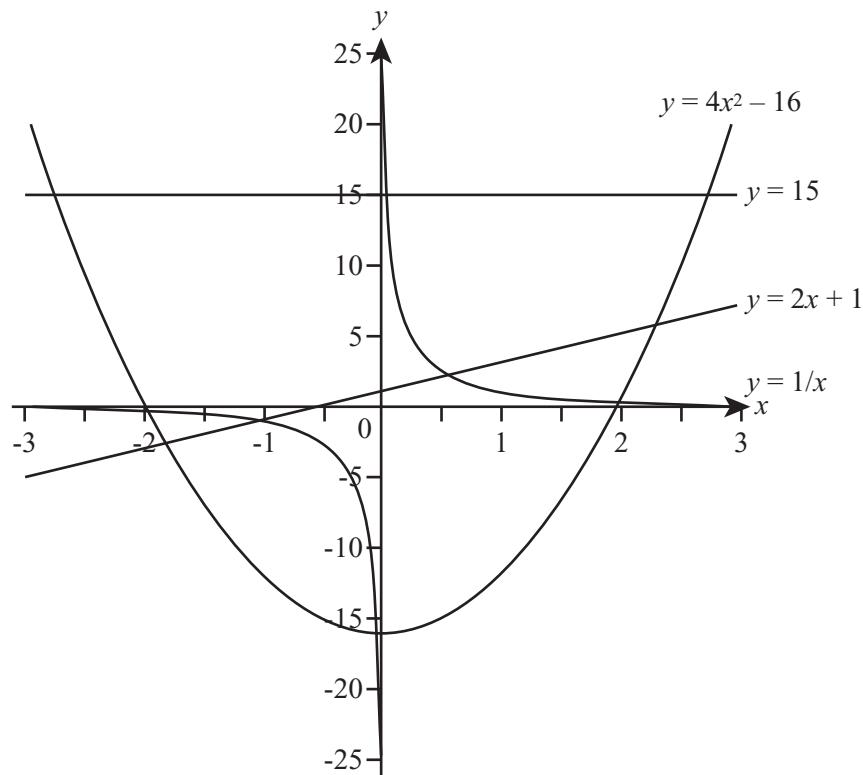


Figure 3.4

Graphs of the functions $y = 15$, $y = 2x + 1$, $y = 4x^2 - 16$ and $y = \frac{1}{x}$.

As rudimentary as the output of the `drawGraph()` function is, it is still valuable as a template or creating functions that generate graphs. The chief characteristics are as follows:

1. The horizontal line represents the equation $y = 15$. This equation is based on the constant function $f(x) = 15$, which returns 15 regardless of the value of x . You can also draw a vertical line with the equation $x = c$. As an example, the y -axis is drawn along the line $x = 0$.
2. The diagonal straight line represents the equation $y = 2x + 1$, illustrating why a function with only a constant term and a term in x is called *linear*. All such functions appear as a straight line graph.
3. The curve, which looks like a big U-shape, is called a *parabola*. It represents the equation $y = 4x^2 - 16$. All quadratic functions produce a similar shape. If the term in x^2 is negative, the curve is inverted.
4. The two curves that hug the axes of the graph, sloping upward and to the right or downward and to the left and disappearing off to infinity near the axes, is the graph of $y = \frac{1}{x}$. As x gets closer and closer to 0, $\frac{1}{x}$ gets very large. In fact, it gets arbitrarily large, and the general expression for this is that it *tends to infinity*. Similarly, as x increases to infinity, y gets smaller and smaller, without ever quite reaching 0. When you see this behavior, you say that the lines $x = 0$ and $y = 0$ are *asymptotes* of the function.

What Graphs Tell You

While plots of graphs don't tell you anything new about functions, they are of inestimable value in almost any mathematical, scientific, and technical context. Among other things, they allow certain important pieces of data to be seen quickly. Consider, for example, the questions of whether or where the line passes through the axes. Does it reach a maximum value or minimum value? Does it go off to infinity? Does it have asymptotes? As importantly, you can use this information to deduce other facts about the function. To explore such questions, consider what the graphs plotted in Figure 3.4 allow you to determine about the functions.

About Horizontal Functions

The horizontal line has two main features. First, it is horizontal, which means that it is a constant function, independent of x . Second, it crosses the x -axis at $y = 3$. These two facts tell you everything that can be said about the function. It is a *constant* function, and all such functions behave in a similar way.

About Diagonal Functions

The diagonal line can be described in various ways, but the two most important are the *gradient* and the *intercept*. The gradient of a straight line is defined the same way as the gradient of a hill. If you travel a certain horizontal distance, you also travel a certain distance upward. The ratio of these two values, vertical to horizontal, is the gradient. You can measure the gradient by taking any two points on the line and dividing the vertical distance between them by the horizontal distance. In this case, the line passes through the points $(2, 5)$ and $(-0.5, 0)$, which are separated by 5 units vertically and 2.5 units horizontally. If you divide 5 by 2.5, you get a gradient of 2.

The intercept is the point at which the line crosses the y -axis, which in this case is $y = 1$. Another thing to notice about these values is that when you look at the equation of the line, $y = 2x + 1$, you see that 2 is the gradient and 1 is the intercept. This is a general fact about straight-line graphs. The gradient is the coefficient of the x term, and the intercept is a constant value. The gradient is often represented by the parameter m and the intercept by c . Given this characterization, the family of linear equations is represented by the equation $y = mx + c$.

About Parabolic Functions

Parabolas offer more information than horizontal and diagonal lines. First, as mentioned previously, they can either curve downward, like a bowl, or upward, like a mountain. This gives you the sign of the term in x^2 . A bowl has a positive coefficient of x^2 , and a mountain has a negative coefficient.

Second, by observing the points at which the parabola intersects the x -axis, you can determine the roots of the function. [Recall that a root of a function $f(x)$ is a value a such that $f(a)$ is zero.] In this case, the curve crosses the x -axis at $+2$ and -2 , showing that the roots of the function are $+2$ and -2 . This means that $(x + 2)$ and $(x - 2)$ are the factors of the function. If the parabola only just touches the x -axis, then you have a function with just one root. This means that the function is a *square quadratic*. If it doesn't cross the x -axis at all, then the function has no real roots and cannot be factored.

A third point is that, as with the straight line, you can determine the constant term of the function from the point at which it crosses the y -axis. A final point is that you can find the maximum or minimum value of the function from the curve, by locating the point at which the parabola turns back on itself and reading the y -value at that point.

About Asymptotic Functions

The function $y = \frac{1}{x}$ generates asymptotes. Such functions are harder to characterize or read information from than are the others, but you can read off the values of the asymptotes directly. Caution is necessary, however. Many functions that seem to have asymptotes are actually just changing very slowly. Consider, for example, the graph in Figure 3.5 of $y = \log_e(x)$. The logarithm function does not have a maximum value, although from the graph it may seem that it does.

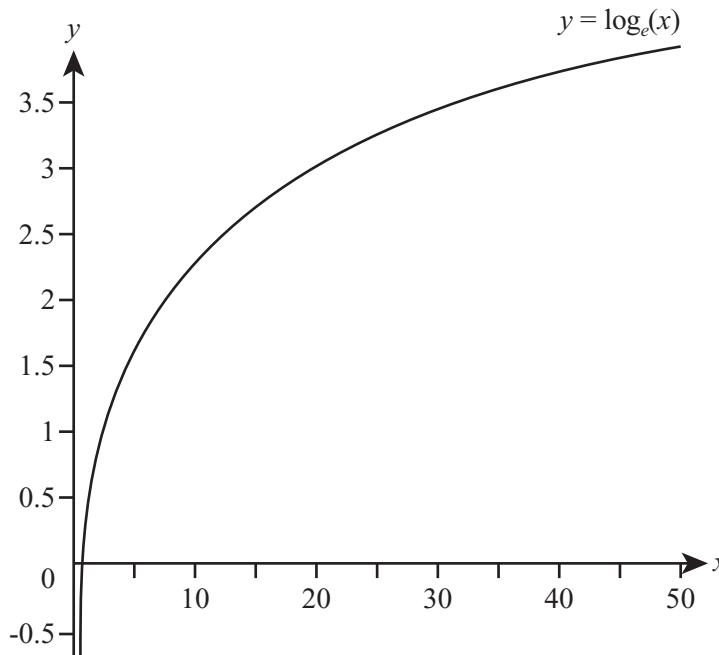


Figure 3.5

The graph of $y = \log_e(x)$.

Parametric Curves and Functions

Although simple functions are the most common things to draw on a graph, not all curves can be described in terms of a standard function. The graphs you have seen so far have been of single-valued functions. It is not possible to draw a circle using this technique because a circle does not have a distinct value of y for each x -coordinate. In terms of functions, a circle is multivalued.

One way to avoid the problem created by multivalued functions is to use a *parameterization*. When you do this, instead of using a single function $f(x)$ and plotting $y = f(x)$, you use two functions $x(t)$ and $y(t)$ and plot the points $(x(t), y(t))$ for each value of t . The token t serves as a dummy variable here. However, because parametric functions are often used to represent motion and motion involves time, t frequently represents time.

To illustrate how parametric equations work, suppose that two functions are given by the formulas $x = at^2$ and $y = 2at$. If you plot a graph of the points (x, y) given by allowing t to vary across the real numbers, you get the graph shown in Figure 3.6, which is a parabola lying on its side. In fact, it's the parabola $y^2 = 4ax$, as you can find by substituting for t in the parametric formula.

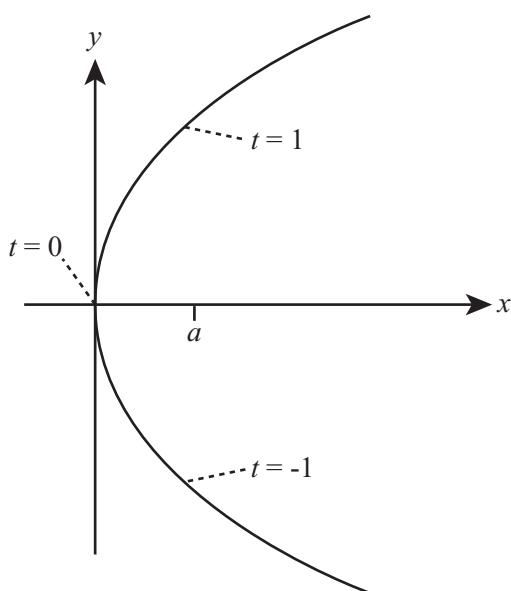


Figure 3.6

The parabola formed from the parametric formula $x = at^2$, $y = 2at$.

This curve is one that you couldn't draw using the `drawGraph()` function, because it's multivalued in the y -coordinate. As you'll see in later chapters, parametric formulas allow you to draw far more complex curves than simple functions, including the Bezier curves and splines seen in vector drawing and 3D modeling packages.

Exercises

EXERCISE 3.1

Write a function named `substitute(functionString, x)` that substitutes a value for x into a function given in standard notation.

The function should take two arguments, a string such as “ $5x^2 + 3(4-2x)$ ” and a value such as 5, and should return the result of substituting the second argument for the variable x in the string. Use the `^` character as shown previously to represent powers, and the `/` and `*` characters to represent division and multiplication. Try to make the function as general as possible, particularly dealing with parentheses. Here are a few test functions you could try it out on: $5x + 3$, $4 - 2(x-5)$, $(x^3-4)/2$, $(x - 4)(2 - 3x)$, $2^{(x - 4)/(x - 5)}$.

EXERCISE 3.2

Write a function named `simplify(functionString)`, which simplifies a given function as far as it can.

Simplification is a task requiring intelligence, and you won't be able to make the function work as well as a human would, but you should be able to make some headway. Your program should be able to take a function, group like terms together, put fractions over a common denominator, and if you are very ambitious, factorize the result. You could work just with the variable x , or allow multiple variables.

EXERCISE 3.3

Write a function named `solve(equationString)` which solves a given equation.

As before, this will not be infallible, but it should be able to deal with linear or quadratic equations. Use your previous function `simplify` to help you.

Summary

In the course of this chapter, you have covered several years' worth of basic algebra, which necessarily means you have missed out on many details and lots of practice. On the other hand, applying programming techniques to the concepts will have given you a good head start in understanding them.

The methods and concepts involved have been presented in an advanced fashion. The key focus of the discussion has been keeping the idea of the function at the forefront. As long as you understand what a function is, the rest of algebra falls neatly into place.

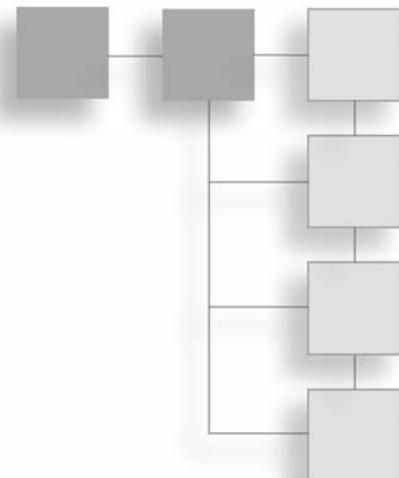
In the next chapter, you'll slow down a little and begin looking at geometry, another topic directly relevant to programming.

You Should Now Know

- The meanings of the terms *variable*, *parameter*, *constant*, and *unknown* and how they are related to and different from each other
- The meaning of the word *function*, the idea of a function as a map between different sets, and the concepts of *one-to-one*, *many-to-one*, and *multivalued* functions
- What an equation is and how to *solve* one for a particular unknown in simple cases, including quadratics, cubics, and simultaneous equations in two or more unknowns
- How to *simplify* and *factor* functions, and use these skills to help with solving equations
- How to draw a *graph* of a function, and how to use it to read off information about the function
- How to draw a *parametric curve*

CHAPTER 4

GEOMETRY AND TRIGONOMETRY



In This Chapter

- Overview
- Angles
- Triangles
- Calculations with Triangles
- Rotations and Reflections

Overview

Geometry is the study of shapes and space, and particularly of symmetry. In this chapter, you'll look mostly at the practical side of the subject, however, beginning with the measurement of angles and triangles. This area of mathematics is trigonometry. When programming anything involving movement, such as games, you constantly use trigonometry. In fact, it is so important that you must have a thorough understanding of how it works.

Angles

An *angle* is a way to measuring a direction. If two people set off from the same point, after they have both traveled 10 meters, they can be anything from zero meters apart (if they set off in the same direction) to 20 meters apart (if they set off in opposite directions). An angle is a measurement of how different the two directions are.

Angles and Degrees

The most common way to measure an angle is to consider the two possible directions as radii of a circle. You can then consider the angle between any two radii as a fraction of the circle. For example, in Figure 4.1, the clockwise angle between the lines A and B is a quarter of the circle, and the clockwise angle between the lines B and C is a third of the circle.

Note

A *radius* of a circle is a line from its center to its *perimeter*. Another term for perimeter is *circumference*. *Radii* is the plural of *radius*. If a straight line is drawn between two points on the circumference and passes through the center of the circle, it is called a *diameter* of the circle.

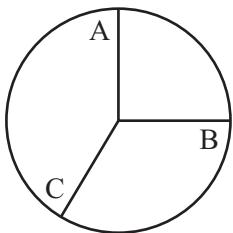


Figure 4.1

A circle with various radii marked.

Since you can measure the angle in either direction, the phrase “the angle between two lines” is potentially ambiguous. In Figure 4.1, the counterclockwise angle between A and B is three quarters of the circle. Generally, when you talk of the angle between two lines, you mean the smallest angle. This will be the approach used in this chapter unless otherwise indicated. However, in later chapters, you’ll be more interested in measuring the angle in a particular direction.

Angles can be measured in different units, the most common of which is the *degree*. When measuring in degrees, you divide a full circle into 360 equal parts. Each part is a single degree, written as 1° . There is nothing special about 360, except that, as noted in Chapter 1, it has a large number of divisors, meaning that many common fractions of a circle have an integer number of degrees. Consider, for example, that a quarter turn is 90° (called a right angle), a half turn is 180° (a straight line), one third is 120° , a sixth is 60° , a fifth is 72° , and so on. Figure 4.2 illustrates some of these angles.

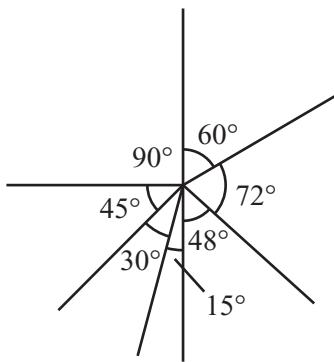


Figure 4.2

Some common angles measured in degrees. Measured angles are denoted by a small arc between the lines, and right angles are denoted by squares.

Of the angles shown in Figure 4.2, the most important is the right angle. A square has four equal sides, and the angle between each pair of adjacent sides is a right angle. If you draw the diagonals of the square, they also meet at right angles. When two straight lines meet at right angles, they are called *perpendicular*. Four right angles divide the circle into four equal quadrants. As illustrated by Figure 4.3, if a circle is drawn on a graph, the quadrants can be seen to be complimentary. For every point (x, y) inside the circle, there are corresponding points $(x, -y)$, $(-x, y)$ and $(-x, -y)$. These are also inside the circle, each in a different quadrant—except when x or y is zero.

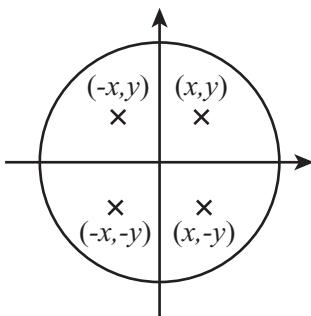


Figure 4.3

The quadrants of a graph.

Certain terms facilitate discussion of the relative magnitudes of angles. An angle smaller than a right angle is called *acute*. An angle larger than a right angle (but smaller than two right angles) is called *obtuse*. An angle greater than half a circle is called a *reflex angle*.

Area and the Number π

If you draw a square around a circle and pick a point at random within it, what is the likelihood that this point lies inside the circle? This question is one way of explaining the concept of the *area* of a shape. Imagine that you have two squares, one that has sides twice as long as the sides of the other. You place the two squares side by side on a table. You then repeatedly make evenly spaced dots on the table with a pen. How many dots are within the respective squares? The answer is that four times as many dots are in the large square as in the smaller. The reason is that you can fit four of the small squares in the larger one, for four times as many dots appear in the larger square than in the smaller one. The notion of a *unit squared* extends this relationship. If a square has sides x units long, then the area of the square is x^2 *square units* or *units*² (hence the term *squared*). When $x = 1$, the shape is called a *unit square*.

You can easily calculate the area of a rectangle. Like a square, a rectangle is a shape that has four sides, and each pair of adjacent sides lies at right angles to each other. Unlike the four sides of a square, the four sides of a rectangle do not need to be equal. Only the sides that are opposite each other need to be equal. The implication is that a square is a special kind of rectangle. The area of a rectangle is the product of its length and width. The area of the rectangle in Figure 4.4 is 12 square units. In other words, 12 unit squares fit inside it.

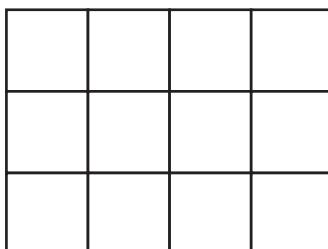


Figure 4.4

3×4 rectangle.

You'll return to areas when you look at triangles, but for now consider again the question that began this section. This question can be rephrased as, "What is the area of a circle?" To answer this question, it is necessary to first say specifically what is meant by a circle. A circle is the shape drawn when you hold the end of a piece of string at a fixed point, attach a pen to the other end, and draw a curve with the string held taut. The length of the string is the radius of the circle.

Note

The term *radius* refers both to a length and to a line. The radius of a circle is a line drawn from the center of a circle to its circumference, as mentioned earlier, but it is also the length of the line. Uses of *radius* are similar to those of *side*. For example, you might use the phrase "the side of a square" to mean "the length of each side of the square." Likewise, you might use "the sides of a rectangle" to mean "the lengths of the two pairs of opposite sides of the rectangle." *Radius* here means "the length of any radius." Similarly, *diameter* can mean "the length of any diameter."

Once you know the radius of a circle, you know almost everything there is to know about it, although you might also need to know the position of the center point. The area of a circle is always in a particular proportion to the square of its radius. If a circle has radius r , then it has an area of $3.1415927\dots \times r^2$. The precise value of this constant $3.1415927\dots$ is given by the symbol π , which is the Greek equivalent of the letter "p," spelled in English "pi" and pronounced "pie." In many computer languages, this value can be accessed as a math property and is usually available either as a predefined constant PI or (less often) as function named pi().

π is probably the most ubiquitous irrational number in mathematics, cropping up in all kinds of unexpected places—even more than e , which runs a close second. Mathematicians have proved any number of surprising facts about this number over the centuries, with probably the most elegant provided by the mathematician Leibniz:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

Involving a series similar to Leibniz's, one of the earliest uses of computers on a vast scale was in calculating the digits of π to an enormous precision. Many terms of a series are required to get anywhere near to a precise value. Today the digits of π are known to over a trillion decimal places. This is testimony both to the importance of the number and to the obsessive nature of computer programmers.

If you know the formula for the area of the circle, you can answer the question concerning how to define a circle. A square drawn around a circle has a side equal to the diameter of the circle. If the circle has a radius of 1 unit, then the square has a side of 2 units and so an area of 4 units^2 . By the formula for the area of a circle, the circle has an area of $\pi \text{ units}^2$. So the circle has $\pi/4$ times the area of the square, which is a value of approximately 0.7854.

π also turns up in another important element of the circle, the length of its circumference. The circumference is equal to π times the length of the diameter, or π times twice the length of the radius.

Radians

As was emphasized previously, there is no particular reason for choosing the degree as a unit of angles. It is simply a unit that is convenient for some calculations. Although it may not seem very natural at first, another common unit of measurement is the *radian*. Instead of dividing a circle into 360 equal degrees, you divide it into 2π radians—which is to say, “six and a bit” of them. The idea of dividing the circle into non-integer units is what seems strange at first, but there is nothing wrong with it. After a while, it seems no odder than accepting the idea that there is not an exact number of centimeters in an inch.

It is often necessary to convert between degrees and radians. One radian is $\frac{1}{2\pi}$ of a circle, so it is $\frac{360}{2\pi}$ degrees. Similarly, one degree is $\frac{1}{360}$ of a circle, so it is $\frac{2\pi}{360}$ radians. Thus an angle of 12° is equal to $\frac{12 \times 2\pi}{360} = \frac{\pi}{15} = 0.209$ radians. An angle of $\frac{\pi}{2}$ radians is equal to 90° .

If you must often convert between the values, it is generally simplest to store the conversion factor as a constant from the beginning.

There is a third unit of angles, which is called the *gradian*, but this is not something you are likely to encounter or need, although you can still find it on calculators. It is based on a division of the circle into 400 parts.

Triangles

As was mentioned previously, trigonometry is the area of mathematics that studies triangles. At the same time, the triangle is also studied extensively as a subject of geometry. A *triangle* is a figure made up of three points not lying on a straight line that are joined by three straight-line segments. When points are positioned in this manner, they are known as

non-collinear. Each point of the triangle is referred to as a *vertex*, and together they are the *vertices* of the triangle. When you apply algebra to trigonometry and geometry, the result is analytical geometry—a field of mathematics pioneered by Rene Descartes.

The Types of Triangle

Triangles can be classified into four main types according to the three angles inside them. As illustrated in Figure 4.5, you normally identify the features of a triangle by using capital letters for vertices, lower-case letters for side lengths, and angles labeled either by Greek letters as in the figure or with an angle designation (such as $\angle ABC$), listing three points that define the angle. Likewise, each side is labeled to correspond to the opposing vertex. You may also see the angles denoted simply by the letter of the vertex: angles A, B, or C. Small arcs are used to emphasize angles of the lines joined at the vertices.

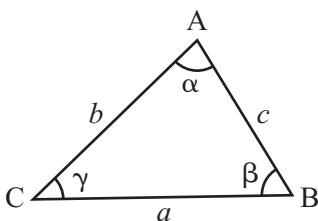


Figure 4.5

A triangle with its vertices, sides, and angles labeled.

Note

If you are unfamiliar with the Greek alphabet, see Appendix C. For a quick review, however, note that α is alpha, β is beta, γ is gamma, δ is delta, ε is epsilon, and θ is theta. You have already seen π (pi).

The angles of a triangle always add up to 180° . One demonstration of this is that if you draw the same triangle three times, as shown in Figure 4.6, you can see that the angles α , β , and γ lie on a straight line, which means that the sum of these angles must be half a circle, or 180° . That this is true for any triangle was demonstrated by Euclid (324 – 264 B.C.E.), the most famous of geometers, but has been known since the earliest days of geometry.

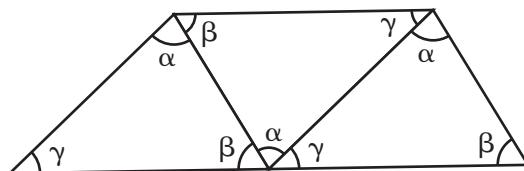


Figure 4.6

The angles in a triangle sum to 180° .

General Types of Triangles

While a general set of propositions apply to all triangles, regardless of their shape, the shapes of triangles are classified according to the magnitudes of their angles. Generally, there are three classifications of triangles, as illustrated by Figure 4.7. The following list summarizes the three classifications.

- **Equilateral.** The simplest triangle is the equilateral triangle, which has three equal sides and three equal angles. Given that angles of a triangle always equal 180° , each angle is equal to 60° .
- **Isosceles.** An isosceles triangle has two equal sides. If sides a and b are equal, then angles α and β are also equal.
- **Scalene.** A scalene triangle is any other triangle. Such triangles have no equal sides and no equal angles.

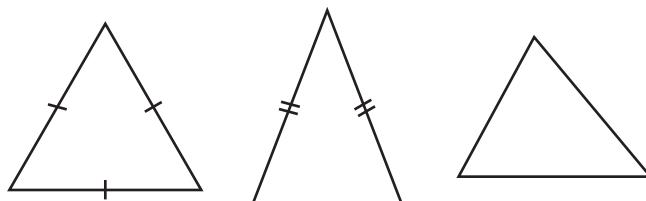


Figure 4.7

Equilateral, isosceles, and scalene triangles.

In addition to the three primary classifications of triangles, geometers refer to triangles as *acute* and *obtuse*. An obtuse triangle has one angle greater than 90° . An acute triangle has no angles greater than 90° . In this sense, then, equilateral and isosceles triangles are acute, and a scalene triangle is a candidate for being obtuse.

Right Triangles

One particularly important kind of triangle is the *right-angled triangle*. It is often referred to as a *right triangle*. As its name implies, it is a triangle that has a right angle within it. A right angle is an angle of 90° . There are special names for the sides of a right triangle. Two of the sides are called *legs*, and one is called the *hypotenuse*. As illustrated in Figure 4.8, two of the legs (a and b) are joined by a right angle (which is indicated by a small square or rectangle). The ends of the two legs are joined by the hypotenuse. Given that the right angle equals 90° and that the sum of the three angles must be 180° , the two smaller angles must necessarily sum to 90° .

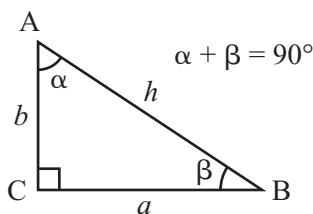


Figure 4.8

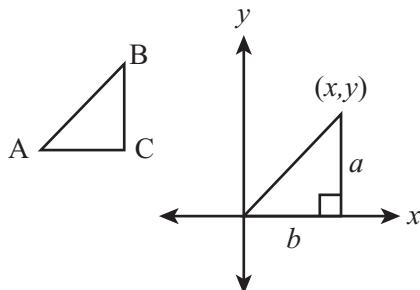
A right-angled triangle.

The Pythagorean Theorem

The Greek mathematician and philosopher Pythagoras (569 – 494 B.C.E.) is famous for having developed a theorem that relates the sides of the right triangle. This is usually referred to as the Pythagorean Theorem. It was left for René Descartes (1596 – 1650) and other mathematicians to extend and apply the Pythagorean Theorem to the coordinate system and algebra.

How the Pythagorean Theorem works begins with right-angled triangles. There are several reasons why such triangles are important. The first is that they are easy to recognize and crop up in many natural circumstances. For example, when working on a graph, it is often useful to draw the lines joining points and create a right-angled triangle with sides parallel to the x - and y -axes, as is shown in Figure 4.9. As was discussed in a previous chapter, the gradient, or slope, of a line is given by the ratio of the one side of the triangle (a) to another (b) or, on the x - and y -axes, the ratio of y to x (the run to the rise).

Triangle: Gradient (or slope) = a/b



Coordinate System: Gradient (or slope) = y/x

Figure 4.9

A right-angled triangle drawn on a graph.

A second reason for the importance of right-angled triangles is that they have a number of useful properties. These will be dealt with in subsequent sections. A third reason, as Figure 4.10 illustrates, is that any triangle can be easily split into two right-angled triangles by dropping a perpendicular from one vertex to the opposing side.

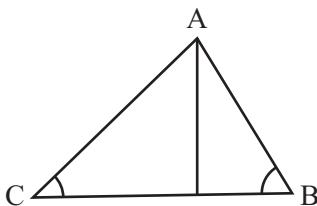


Figure 4.10

A triangle split into two right-angled triangles by dropping a perpendicular from vertex A.

As mentioned already, right-angled triangles were studied extensively by Pythagoras. The Pythagorean Theorem states that for a right-angled triangle ABC with the right-angle at vertex C, $a^2 + b^2 = c^2$. There are many ways to prove this theorem. One approach is shown in Figure 4.11.

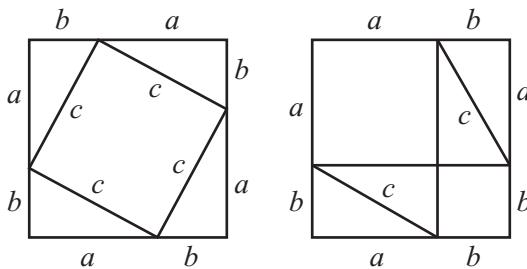


Figure 4.11

A geometrical proof of Pythagoras' Theorem.

In Figure 4.11, in the diagram on the left, you find a square with side $a + b$. Around the outside are arranged four equal right-angled triangles with legs a and b . The hypotenuses of the triangles, with length c , form a smaller square inside the larger, whose area is c^2 .

In the diagram on the right, you have an identical square, again with side $a + b$. This time, the four right-angled triangles have been rearranged to fit inside the square as two rectangles meeting at a corner. Each rectangle has sides a and b . The remainder of the square is subdivided into two smaller squares, one of which has side a , the other side b .

The total area of these two squares is $a^2 + b^2$. However, since these the two large squares and the eight right-angled triangles within them are all the same, the area of the two small squares in the diagram on the right must be equal to the area of the square in the diagram on the left. Given this observation, you can conclude, that $a^2 + b^2 = c^2$.

Triples

There are an infinite number of right-angled triangles whose sides are of integer lengths. The smallest is the triangle with legs 3 and 4, which has hypotenuse 5. There are a number of ways of generating such triangles, and the sets of three numbers making the sides are called *Pythagorean triples*. There are no sets of integers for which the same is true in any higher power. In other words, there are no sets of positive integers a, b, c, n , with $n > 2$, such that $a^n + b^n = c^n$. This fact is called *Fermat's Last Theorem* and was a famous unsolved problem until very recently.

Extending the Pythagorean Theorem

Using the Pythagorean Theorem, if you are given the lengths of the two sides of a right triangle, you can quickly determine the length of the third side. For example, if you know that the hypotenuse of a triangle is 13 cm, and one side is 5 cm, then by Pythagoras, the third side's length must be equal to $\sqrt{13^2 - 5^2} = \sqrt{144} = 12$ cm.

As illustrated by Figure 4.12, a right-angled isosceles triangle with sides that are half a square (usually referred to as *unit length*) has a hypotenuse $\sqrt{2} = 1.414 \dots$ units long. Similarly, if you take an equilateral triangle with sides of length 2 and cut it in half, the length of the perpendicular is $\sqrt{3}$ units. When this is done, notice that the isosceles triangle has interior angles of 45° , and the triangle with a perpendicular of $\sqrt{3}$ units has angles of 60° and 30° .

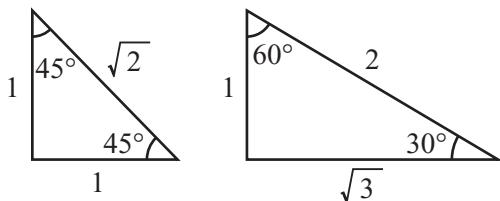


Figure 4.12

Two important right-angled triangles.

The Trigonometric Functions

Refer back to Figure 4.9 and note that the gradient (or slope) of a line can be represented by drawing a right-angled triangle on a graph. For any right-angled triangle with leg a parallel to the y -axis and leg b parallel to the x -axis, the gradient of the hypotenuse is equal to $\frac{a}{b}$. It is useful to be able to relate this value to the size of the angles at A and B, and you do this by using the $\tan()$ function. If the angle at A is x , then $\tan(x)$ gives the gradient of the line AB. Two other functions, $\sin(x)$ and $\cos(x)$ are equal to the ratios $\frac{a}{c}$ and $\frac{b}{c}$, respectively. These three functions are known as the trigonometric functions. Notice that $\tan(x) = \sin(x)/\cos(x)$.

Note

The abbreviation for the tangent function is *tan*, for the cosine function, *cos*, and for the sine function, *sin*. The trigonometric functions are dependent on the units of measurement of the angles. Generally, computer languages assume units of radians, but in this section, degrees are used for clarity. If you are working in degrees, it is vital to convert your angle measurements to radians for use in computer programming languages. To do so, multiply them by $\pi/180$ before using them as arguments to the trigonometric functions.

Inspecting the triangles shown previously in Figure 4.12, you can see that $\sin(45)$ must be equal to $1/\sqrt{2}$, that $\sin(30)$ is equal to 0.5, and that $\sin(60)$ is $\sqrt{3}/2$. The values of \cos and \tan can be calculated in a similar manner.

If you measure the values of these functions for various right-angled triangles and plot them on a graph, the result is rather interesting, as is shown in Figure 4.13. On the top of Figure 4.13, you see the graphs of $y = \sin(x)$ and $y = \cos(x)$ for $x = 0^\circ$ to 360° . Notice that you can measure the lengths of the sides in both directions, just as is done when measuring gradients. Both of these graphs are the same shape, a continuous wave called a *sine wave*. The wave for $\cos(x)$ lags a little behind the one for $\sin(x)$. When this occurs, it is said that the two waves are 90° *out of phase*.

On the bottom of Figure 4.13 is the graph of $y = \tan(x)$. The shape of this graph differs from those of the sine waves. One major feature is that a continuous line is not generated. Instead, you see a series of curves with asymptotes every 180° .

The functions $\sin()$ and $\cos()$ are closely related to circular motion, as is emphasized in Chapter 16. Likewise, their behaviors are surprisingly similar to that of the $\exp()$ function. They can be calculated (in radians) by using the following infinite series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

Notice that when x is small, $\sin(x)$ is very nearly equal to x . Notice also that $\sin(0) = 0$ and $\cos(0) = 1$.

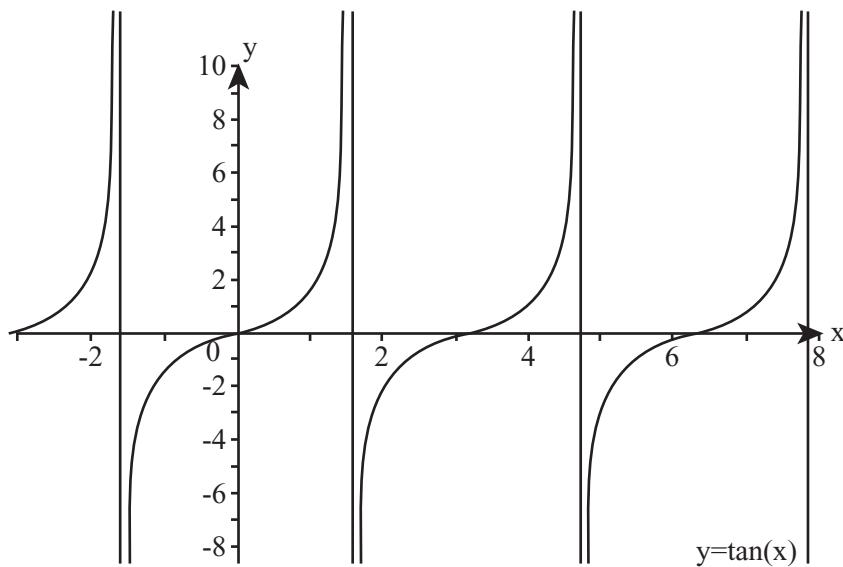
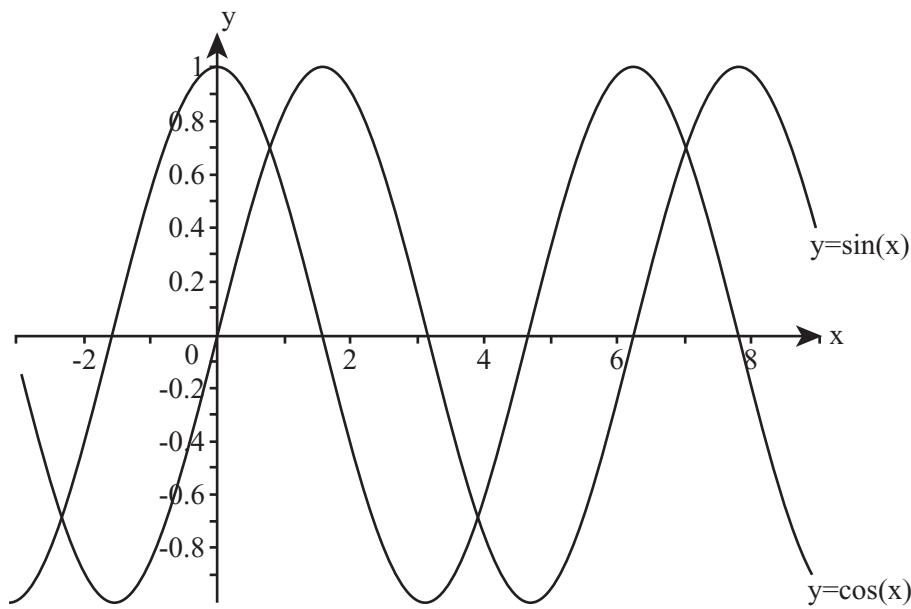


Figure 4.13

The graphs of $\sin(x)$, $\cos(x)$, and $\tan(x)$.

Trigonometric Identities

To extend the basic trigonometric functions, there are also a number of formulas known as the *trigonometric identities*. The following list states them without proof or even further discussion, but it is a useful exercise to think about how you might prove them. Accordingly, for any value of x and y :

- $\sin^2(x) + \cos^2(x) = 1$. Here $\sin^2(x)$ is shorthand for $(\sin(x))^2$.
- $\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$
- $\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$
- $\tan(x + y) = \frac{\tan(x) + \tan(y)}{1 - \tan(x)\tan(y)}$
- $\sin(2x) = 2\sin(x)\cos(x)$
- $\cos(2x) = \cos^2(x) - \sin^2(x)$
- $\tan(2x) = \frac{2\tan(x)}{1 - \tan^2(x)}$

The Inverse Trigonometric Functions

Just as it is important to be able calculate gradients from angles, so it is important to go the other way and calculate angles from gradients. Each of the three trigonometric functions has an *inverse function*. These are referred to as `asin`, `acos`, and `atan`, abbreviations for *arctangent*, *arccosine*, and *arcsine*. Each inverse function maps from the domain to a range. For all three the domain is $[-1, 1]$. The `sin` and `cos` functions then map to the range $[-\infty, \infty]$, while the `tan` function maps to the range $[0, 360]$. Each function takes a number and can map it to more than one angle, so they are multivalued. Still, for each of the functions, there is a standard mapping, as is summarized in the following list.

- The inverse of the `sin()` function is the `asin()` function, also expressed as `arcsin()` and `sin-1()`. This maps positive values between 0 and 1 to values between 0 and 90° , and negative values greater than or equal to -1 to values between 90° and 180° . For all values of x , $\arcsin(-x) = 180^\circ - \arcsin(x)$.

- The inverse of the $\cos()$ function is the $\arccos()$ function, also expressed as $\arccos()$ or $\cos^{-1}()$. This maps values in $[0, 1]$ to the range $[0, 90^\circ]$ and values in $[-1, 0]$ to the range $[-90^\circ, 0]$. For all values of x , $\arccos(-x) = -\arccos(x)$.
- The inverse of the $\tan()$ function is the $\arctan()$, also expressed as $\tan^{-1}()$ or $\arctan()$. This maps values in $[0, \infty]$ to the range $[0, 90^\circ]$. In the domain $[-\infty, 0]$ it maps to the range $(-90^\circ, 0]$. For all values of x , $\arctan(-x) = -\arctan(x)$.

Note

Domains and ranges are expressed as *intervals*. Square and curved brackets indicate inclusion and exclusion from intervals. The notation $[a, b]$ indicates the *closed interval* from a to b , which is the set of real numbers between a and b inclusive. The notation (a, b) in this context means the *open interval*, the set of real numbers between a and b , not including the numbers a and b . Similarly, $[a, b)$ means the set of numbers greater than or equal to a and less than b .

The inverse trigonometric functions are used in various ways in mathematics, but for programming purposes, used in conjunction with the Pythagorean Theorem, the $\arctan()$ function satisfies most of your needs. As becomes evident from an inspection of Figure 4.14, the following two equations show how the ratios given by the hypotenuse and the opposite side (x) of a triangle can be used as arguments to the $\arctan()$ function to generate values equal to the inverses of the \sin and \cos functions.

$$\cos^{-1}(x) = \tan^{-1}\left(\frac{\sqrt{1-x^2}}{x}\right)$$

$$\sin^{-1}(x) = \tan^{-1}\left(\frac{x}{\sqrt{1-x^2}}\right)$$

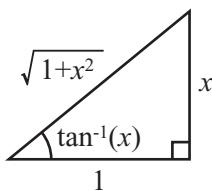


Figure 4.14

Calculating $\arcsin()$ and $\arccos()$ in terms of $\arctan()$.

Because the `arctan()` function maps infinite quantities to finite ones, it has the unusual ability to cope with fractions with a denominator of zero. Because you can't use these in programming, it can save time to create a special version of the `arctan()` function with two arguments instead of one. The two arguments represent the two legs of a right-angled triangle, one of which may be zero. This function is often offered by programming languages as `arctan2()` or `atan2()`. Here is how it can be implemented:

```
function atan2(y, x)
  set deg=1
  if x=0 and y<0 then deg=90
  if x=0 and y>=0 then deg=-90
  if y=0 and x<0 then deg=0
  if y=0 and x>=0 then deg=-180
  if deg=1 then return arctan(y/x)
  otherwise return deg*pi/180
end function
```

Calculations with Triangles

With the help of the trigonometric functions and the Pythagorean Theorem, you can solve problems that involve triangles and more complex figures. This section explores both mathematical and computational aspects of such solutions.

The Sine and Cosine Rules

One of the most common challenges to anyone working with triangles is *to solve the triangle*. Solving a triangle involves using partial information about a triangle to discover all the information, which might include its angles and the lengths of its sides. For example, referring to Figure 4.5, if you are given the triangle ABC and limited information about its angles and sides, using trigonometric functions and the Pythagorean Theorem, it is likely that you can deduce the values concerning sides a , b , c and angles α , β , and γ . To fully solve for a triangle, you seek the following information:

- The lengths of the three sides
- Any two angles and one side
- Any two sides and the angle between them
- In a right-angled triangle, the length of any two sides

Except for a right-angled triangle, you cannot solve a triangle given two sides and a non-included angle. The reason for this is that there may be two possible triangles, as illustrated in Figure 4.15. Additionally, you cannot solve a triangle if you know only the three angles. The reason is that there are an infinite number of triangles with the same three angles. These angles do not change if the lengths of the sides are proportionately increased or decreased.

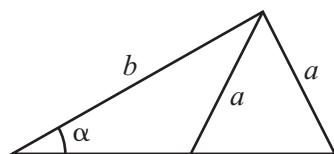


Figure 4.15

Two different triangles sharing the same values for sides a , b , and angle α .

The two most powerful methods for solving triangles are called the *sine rule* and the *cosine rule*. The sine rule relates angles to the lengths of their opposite sides, while the cosine rule relates one angle to the lengths of the three sides.

The sine rule is particularly neat, and says that for any triangle

$$\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)}$$

To see why this is so, examine the triangle depicted in Figure 4.16. In this figure, a perpendicular line has been dropped from the vertex B to intersect at point P with the line AC. Of such a triangle, you can say that the line BP has length l . Given this start, you can arrive at two equations:

1. $\sin(\alpha) = \frac{l}{c}$
2. $\sin(\gamma) = \frac{l}{a}$

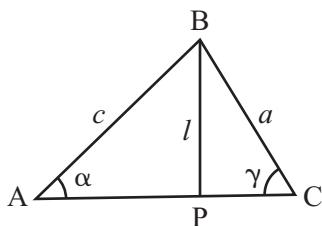


Figure 4.16

Proving the sine and cosine rules.

If you rearrange equation 2 and substitute the value of l into equation 1, you get this set of equations:

$$\sin(\alpha) = \frac{a \sin(\gamma)}{c}$$

$$\frac{a}{\sin(\alpha)} = \frac{c}{\sin(\gamma)}$$

A symmetrical argument applies to b and β .

The cosine rule is a little more complicated, but it is easy to remember because it is an extension of the Pythagorean Theorem. Here is how it reads:

$$a^2 = b^2 + c^2 - 2bc \cos(\alpha)$$

Proving this is a little more involved, but to do so, you can once again use the triangle from Figure 4.16. You've denoted the line PC as k units long, making the line AP $b - k$ units (since the whole line is b units). Now you can see from the Pythagorean Theorem that

$$a^2 = l^2 + k^2 \text{ (using the triangle BCP)}$$

$$l^2 = c^2 - (b - k)^2 \text{ (using the triangle ABP)}$$

Eliminating l , you have

$$\begin{aligned} a^2 &= c^2 + k^2 - (b - k)^2 \\ &= c^2 + k^2 - b^2 + 2bk - k^2 \\ &= c^2 - b^2 + 2bk \end{aligned}$$

You can now eliminate k by using the angle α , since $\cos(\alpha) = b - \frac{k}{c}$ (from triangle ABP) and $k = b - c \cos(\alpha)$. Substituting this value for k , you get

$$\begin{aligned} a^2 &= c^2 - b^2 + 2b(b - c \cos(\alpha)) \\ &= c^2 - b^2 + 2b^2 - 2bc \cos(\alpha) \\ &= b^2 + c^2 - 2bc \cos(\alpha) \end{aligned}$$

Using combinations of these two rules, along with the Pythagorean Theorem and the fact that angles in a triangle sum to 180° , you can solve any solvable triangle. (See Exercise 4.1 for a challenge in this regard.)

Similar Triangles

As you saw in the previous section, it is possible to have two triangles with the same three angles but sides of different lengths. Two triangles with the same angles are called *similar*. Since the sum of angles in a triangle is constant, you only need to show that two angles are the same to know the triangles are similar.

The principal fact about similar triangles is that their sides are in the same proportion. As with the rectangles presented in Chapter 2, similar triangles are essentially the same triangle, just drawn to a different scale. This means that if you know two triangles are similar, then you can use the lengths of a side in one to deduce the same measure in the other.

In Figure 4.17, you can see a way to measure the height of a building on a sunny day. Place a stick 1 m long vertically in the ground and measure the length of its shadow. Also measure the length of the shadow of the building. The triangle formed by the tip of the stick T, the base of the stick S, and the end of its shadow U is similar to the triangle formed by the top of the building B, the base of the building A, and the end of its shadow C. The angles at S and A are both right angles, and the angles at U and C are both equal to the angle formed by the sun and the ground. To get exactly the same angle of the sun, you need to align the stick so that U and C are in the same place. The illustration places the two in separate positions to make the basic ideas clearer.

Note

When describing two triangles as similar, you should represent them with the vertices in corresponding order. In other words, if triangles ABC and PQR are similar, the angles at A and P, B and Q, and C and R are the same respectively.

Because the triangles in Figure 4.17 are similar, their sides are in the same proportion. As a result, the ratio of the height of the building AB to the height of the stick ST is equal to the ratio of the length of the building's shadow AC to the length of the stick's shadow SU. Expressed in an algebraic form, with the letters in lower case, the ratio is represented like this:

$$\frac{c}{u} = \frac{b}{t}$$

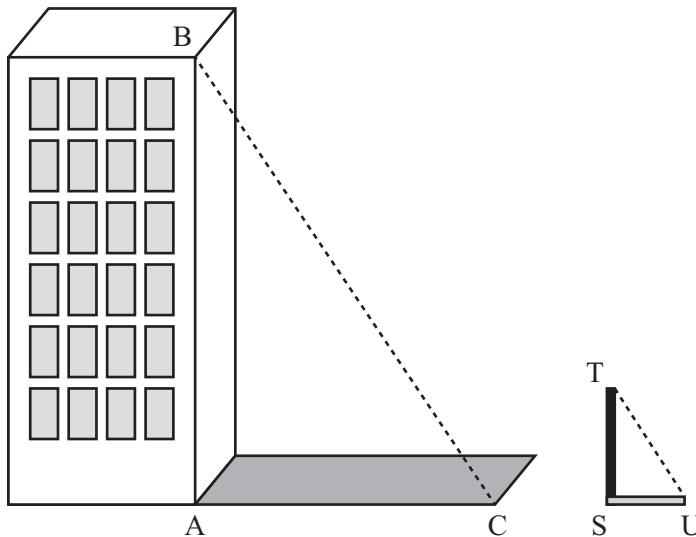


Figure 4.17

Using similar triangles to measure the height of a building.

Given this equation, you can calculate c from the values of the other three lengths, all of which you can measure on the ground.

Note

The notation AB for the length of the line from A to B is useful because it means you need fewer symbols in complicated problems. Notice that in this notation, the letters A and B are in normal typeface. When they are written in bold or with an arrow above them (as in \vec{AB}), they represent not a length but a vector. Vectors are covered in Chapter 5.

If two triangles have sides of the same lengths and angles of the same size, then they are said to be *congruent*, which means that they are exactly the same or at most mirror images of one another. While congruence is often used in the area of geometrical proofs, it seldom appears in programming problems.

The Area of a Triangle

Depending on which measurements you know, there are various ways to measure the area of a triangle. You can use the triangle depicted in Figure 4.16 again to derive one method. Using this method, you take a triangle ABC and drop a perpendicular of length l from the vertex B to the point P on the line AC . This divides the triangle into two right-angled triangles.

Then you draw congruent right-angled triangles alongside each of these to create two adjacent rectangles, each of which is twice the area of the corresponding right-angled triangle. The total area of this rectangle is twice the area of the triangle ABC, so you say that the area of a triangle is $\frac{1}{2}(base \times height)$.

You can also find the area knowing other measurements. For example, if you look at the triangle ACP, you can see that $l = b \sin(\gamma)$, so you can substitute this value into the previous formula to get the area as $\frac{1}{2}ab \sin(\gamma)$.

Rotations and Reflections

In programming you most commonly use the trigonometric functions when working with objects that need to rotate. The final sections of this chapter examine the relationships between the trigonometric functions and rotation.

Transformations

An object drawn on the computer screen could be described in terms of the positions of its vertices. In Figure 4.18, you see one triangle (T) represented in different positions on a monitor screen. Recall that, as previously mentioned, the (0, 0) coordinate of the y-axis is in the upper left of the screen. Accordingly, triangle T initially has vertices at (100, 150), (125, 125) and (150, 130). Given this start, there are then a number of different ways you can move this triangle to a new position. Each of these is called a *transformation*.

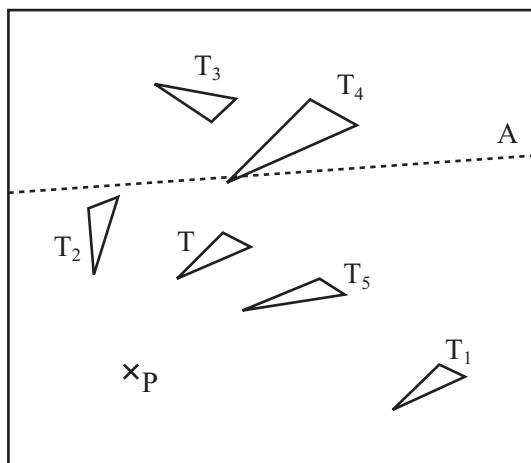


Figure 4.18

A triangle and various transformations of it.

With reference to Figure 4.18, here are some of the ways that triangle T can be transformed:

- **Translation.** If you move the figure to a new position without rotating it (as with triangle T_1), the change is called a *translation*. A translation is described by a vector, which is a topic examined Chapter 5. For the moment, just think of it as a value describing how far you are moving the figure in the x and y directions.
- **Rotation.** If you turn the figure around by an angle (see triangle T_2), the change is called a *rotation*. A rotation is described by a single point and an angle. You say that triangle T_2 is the result of “rotating triangle T by 45° about the point P.” The single point of rotation is sometimes called the *center of rotation*. The angle is called the *angle of rotation* and is usually measured clockwise on a graph or counter-clockwise on a computer.
- **Reflection.** If you flip the figure over (see triangle T_3), the change is a *reflection*. A reflection is defined by a single line known as the *axis of reflection*. You say that triangle T_3 is the result of “reflecting triangle T in the axis A (the line $y = x$).”
- **Scaling.** If you change the size of the figure (see triangle T_4), the change is a scale. *Scaling* an object can be defined by a single point and a number representing the proportion of the size of the new figure to the old one. The single point is the *origin of the scaled object* and should not be confused with the origin of a graph. The number representing proportion is called the *scale factor*. Given these specifics, T_4 is the result of “scaling T by a factor of 1.5 from the point P.” You can also perform more complex scaling by scaling different portions of the object in particular directions.
- **Shearing.** If you take hold of one point of a triangle and shift it across relative to the others, you *shear* the triangle (see triangle T_5). Shearing is defined in terms of a single line (the *line of invariance*) and a number representing the amount of shearing (the *shear factor*).

Each of these transformations can be calculated numerically by performing various operations on the vertices of the object. Many of these calculations involve the trigonometric functions explored in the previous sections.

Rotating an Object by an Angle

Figure 4.19 represents triangle T (ABC). This triangle is to be rotated by an angle α around the origin O to make a new triangle T' (A'B'C'). To accomplish this, you use the y-axis, and as mentioned previously, with respect to monitor coordinates, it is measured downward. To represent the rotation, lines are drawn from the origin O to the points $A = (x, y)$ and $A' = (x', y')$. Also lines are drawn from A and A' to the x- and y-axes and the points X, X', Y, and Y', where these lines meet the axes.

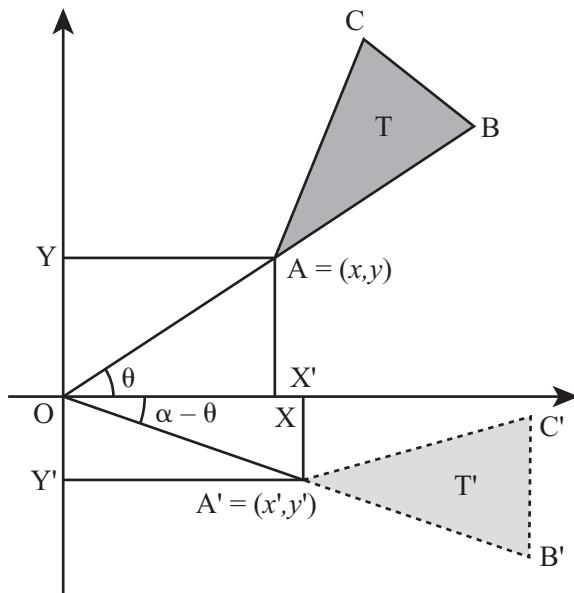


Figure 4.19

Rotating around the origin.

Additionally, the lines OX and OY have lengths x and y , respectively, while the lines for OX' and OY' have lengths x' and y' . Further, the lengths of OA and OA' are both the same (as this is a defining feature of a rotation).

What can you deduce from all this information? Well, first you know that since $OA = OA'$, using the Pythagorean Theorem, you can say that $OA' = \sqrt{x^2 + y^2}$. Second, you can calculate the value of the angle between OA and the x -axis (called θ in the diagram) as $\text{atan}(y, x)$.

Because the angle of rotation is α , you know that the angle θ' , between OA' and the x -axis, is equal to $\alpha - \theta = \alpha - \text{atan}(x, y)$. This means you can calculate the value of x' and y' as follows:

$$x' = OA' \times \cos(\theta) = \sqrt{(x^2 + y^2)} \cos(\alpha - \alpha \tan(y, x))$$

$$y' = OA' \times \sin(\theta) = \sqrt{(x^2 + y^2)} \sin(\alpha - \alpha \tan(y, x))$$

Repeating the operation for each of the vertices of the triangle rotates the whole triangle about the origin.

Note

When calculating x' and y' , you need to be careful about the signs of x and y . Since it ensures that the correct value of the angle is returned, it is best to use the two-argument implementation of `atan()`, `atan(y, x)`, in functions like these.

What if you want to rotate the object around a point other than the origin? To accomplish this, you can use a trick. First, you translate the object so that the center of rotation lies on the origin. Then you rotate it and translate it back. This approach to the problem is based on the idea of a vector. (Again, this will be covered in Chapter 5.)

To rotate a point (x, y) by an angle α around the point $P = (s, t)$, first you translate it by subtracting (s, t) from each point. Then you rotate by α around the origin. Finally you add (s, t) to the resultant points. This gives a complicated formula for the values of x' and y' :

$$x' \sqrt{((x-s)^2 + (y-t)^2)} \cos(\alpha - \alpha \tan((y-t), (x-s))) + s$$

$$y' \sqrt{((x-s)^2 + (y-t)^2)} \sin(\alpha - \alpha \tan((y-t), (x-s))) + t$$

You can see the relationship of these formulas to those you saw before. Each one is a result of replacing x and y with $x - s$ and $y - t$, respectively, and then adding s or t as appropriate. Combining transformations in this way provides a powerful tool that you will see much more of in later chapters.

Around a Center

When programming the movements of graphical objects, you often want to rotate an object around its own center. The center of an object is often called the *center of gravity* or *center of mass*. This is especially the case if you are working with a three-dimensional object. For a triangle in two dimensions, this point is called the *centrum*. As illustrated by Figure 4.20, the centrum is found at the intersection of the three lines joining the vertices to the midpoints of the opposite sides.

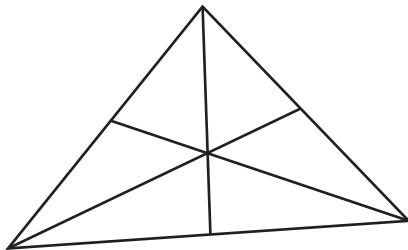


Figure 4.20

The centrum of a triangle.

To find the centrum, you find the mean of the three vertices. To accomplish this, consider that if the vertices are at (x_1, y_1) , (x_2, y_2) and (x_3, y_3) , the centrum is at the point $\frac{1}{3}(x_1 + x_2 + x_3, y_1 + y_2 + y_3)$. Discovering the centrum is a challenge presented in Exercise 4.2.

Note

The *arithmetic mean* (or just *mean*) of a number of values a_1, a_2, \dots, a_n is the sum of the values divided by the number of values: $\frac{a_1 + a_2 + \dots + a_n}{n}$. This may be written using

the notation $\frac{1}{n} \sum_{i=1}^n a_i$, where the capital Greek letter sigma (Σ) denotes a sum of values over the index i . The mean is often referred to as the average, but mathematicians use the word average to mean a number of different functions. These include the arithmetic mean, the geometric mean, harmonic mean, median, and mode, each of which is useful in different circumstances.

Quick Rotations by Special Angles

While it is useful to be able to rotate an object about any axis, it is also handy to know a few shortcuts. Rotating by certain common angles is much simpler. Here are a few shortcuts commonly used:

- To rotate the point (x, y) by 180° about the origin, multiply both coordinates by -1 , getting $(-x, -y)$.
- To rotate (x, y) by 90° about the origin, switch the coordinates around to get $(y, -x)$.
- To rotate (x, y) by -90° about the origin, switch the coordinates the other way to get $(-y, x)$.

These operations can be derived from equations given previously in this chapter.

Reflections

As you have seen in previous sections of this chapter, a reflection can be specified by giving a single line on the plane. This line is called the axis of reflection. As illustrated by Figure 4.21, the image of a point P reflected in a particular axis A is the point P' such that two conditions are met. First $AP = AP'$. Second the line PP' is perpendicular to A .

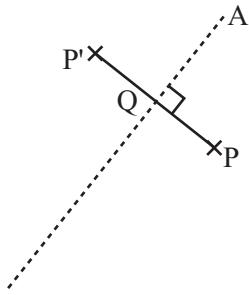


Figure 4.21

The image of the point P under reflection in the axis A .

Although you can calculate the position of P' from this information, it is hard to do so without vectors. However, you can also use rotations and translations to reduce this to a simpler problem. Reflecting in the x - or y -axis is very simple. Accordingly, the point (x, y) reflected in the x -axis gives the point $(x, -y)$, and in the y -axis it gives $(-x, y)$. Therefore, to reflect $P = (x, y)$ in an axis with the equation $y = mx + c$, you can do the following, as is illustrated by Figure 4.22:

1. **Translate** P by $-c$ units in the y direction to get $P_1 = (x, y - c) = (x_1, y_1)$.
2. **Rotate** P_1 by $\text{atan}(m)$ about the origin to get

$$\begin{aligned} P_2 &= (l \cos(\text{atan}(m) - \text{atan}(y_1, x_1)), l \sin(\text{atan}(m) - \text{atan}(y_1, x_1)) \\ &= (x_2, y_2) \end{aligned}$$

where $l = \sqrt{(x_1^2 + y_1^2)} = \sqrt{x^2 + (y + c)^2}$ is the length of the line OP' .

3. **Reflect** P_2 in the x -axis to get $P_2 = (x_2, -y_2) = (x_3, y_3)$.
4. **Rotate** P_3 by $-\text{atan}(m)$ about the origin to get

$$\begin{aligned} P_4 &= (l \cos(-\text{atan}(m) - \text{atan}(y_3, x_3)), l \sin(-\text{atan}(m) - \text{atan}(y_3, x_3))) \\ &= (x_4, y_4). \end{aligned}$$

5. **Translate** P_4 by c units in the y direction to get $P' = (x_4, y_4 + c) = (x', y')$.

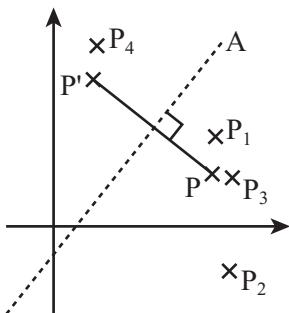


Figure 4.22

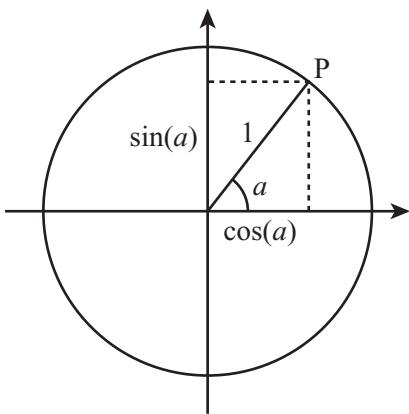
A series of transformations to find P' .

Note

These steps can, of course, be simplified, and by using the trigonometric identities, they can be subsumed into a single formula, which isn't worth going into here. Besides, the whole thing gets easier once you start using vectors and matrices.

sin(), cos() and Circular Motion

The trigonometric functions are not just mathematical abstractions introduced to make calculations easier. They represent a common phenomenon. Imagine that you have a point P at a distance of 1 unit from the origin, as is shown in Figure 4.23. As you can see from the definition of the $\sin()$ and $\cos()$ functions, the coordinates of P must be $(\cos(a), \sin(a))$, where a is the angle the line OP makes with the x -axis. If you plot all such points P, you make a circle around the origin.

**Figure 4.23**

$\sin()$ and $\cos()$ on a circle.

Note

Notice that the length of the line OP is 1. This proves the identity you saw earlier, which established that $\sin^2\theta + \cos^2\theta = 1$.

To draw a circle using the `sin()` and `cos()` functions, you represent the positions of a point moving around a circle at a constant speed. Assuming it starts horizontally, if you were to drive a pin into the side of a wheel, the vertical position of the wheel over time as the wheel spins expresses the output of the `sin()` function. Chapter 16, which covers oscillations, provides further discussion of this topic. For now, simply note that any point on a circle with radius r centered on the point (x, y) has coordinates $(r\sin(\alpha), r\cos(\alpha))$ for some value of α .

Exercises

EXERCISE 4.1

Write a function `solvetriangle(triangle)` that takes an array representing a triangle with incomplete information and returns the array filled in as far as possible.

Your function should accept a six-element array in which the first three elements are the lengths of the sides and the last three are angles. The angles may be in degrees or radians, depending on your preference. Any of these values may be replaced with the string “?” representing an unknown. The function should return 0 if the triangle is impossible, a complete array of six numbers if the triangle can be solved, and an incomplete array if it cannot be solved uniquely.

EXERCISE 4.2

Write a function `rotatetofollow(triangle, point)` that rotates a particular triangle around its centrum to aim at a given point.

The function should take two arguments, one array representing the three vertices of the triangle (the first of which is taken to be the “front”), another giving the point to aim at. It should return the new vertices of the triangle. If you have trouble with this exercise, you might want to come back to it after reading the next chapter.

Summary

In this chapter, you have begun to explore the world of geometry, and you have already learned a number of vital techniques that are useful in animation and games. In the next chapter, you will fill in a lot of the blanks left by this chapter.

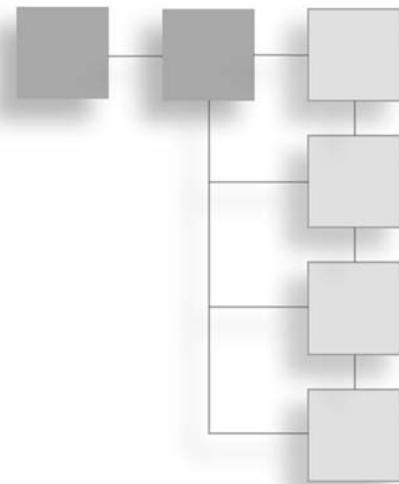
You Should Now Know

- The meaning of *angle* and the different ways that an angle can be measured
- The meaning of the term *area* and how to measure the areas of arbitrary rectangles, circles, or triangles
- The various types of triangles—*scalene, isosceles, equilateral, right-angled*—and their properties
- How to use the *trigonometric functions* and the Pythagorean Theorem to solve problems in right-angled and other triangles
- The meaning of the terms *similar* and *congruent* and what they imply for lengths and angles of shapes
- The meanings of *rotation, reflection, translation, scale*, and *shear*, and how to calculate the first three of these for a given point or shape on the plane

This page intentionally left blank

CHAPTER 5

VECTORS



In This Chapter

- Overview
- Getting from Here to There
- Vector Motion
- Vector Calculations
- Matrices

Overview

In this chapter, you will examine the concept of a vector, a mathematical object describing relative positions in space. You already used vectors informally in the previous chapter, but now you will look at them in more detail. You will finish the chapter by looking at some calculations with matrices and how to use them to describe changes in space.

Getting from Here to There

The first goal of this chapter is formally to describe vectors and see how to perform basic calculations with them.

The Vector as an Instruction

A *vector* is like an instruction that tells you where to move. For example, imagine a pirate's treasure map that says, "Take four steps north and three steps east; then dig." These instructions describe a vector in two dimensions. First you go north. Then you go east. Or imagine being told in a hotel, "Go to the first floor, pass along the corridor, and then take the first door on the right." These instructions describe a vector in three dimensions. You first go up. Then you go along the corridor. Then you turn right.

Vectors describe movement. With the pirate's treasure map, it wouldn't matter if you took one step west, four steps north, and then three steps east. You would still end up at the treasure—as long as you started in the right place. Vectors don't have an intrinsic position in space. They simply tell you that if you start here, you will end up there.

Vectors are normally denoted by a letter in **boldface**, such as **u** or **v**. When working in Cartesian coordinates, you can indicate a vector by identifying the distances moved in the *x*-direction and the *y*-direction. This information is usually provided in a *column array*, which takes the form $\begin{pmatrix} x \\ y \end{pmatrix}$. For example, the vector illustrated in Figure 5.1 is communicated with $\begin{pmatrix} -3 \\ 2 \end{pmatrix}$. Because you are moving in a negative *x*-direction, the value in the *x* position of the vector is negative. These values are called the *components* of the vector in the *x*- and *y*-directions.

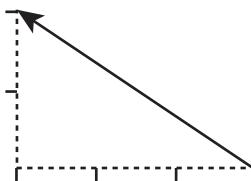


Figure 5.1

The vector $\begin{pmatrix} -3 \\ 2 \end{pmatrix}$.

Note

A vector is drawn as a straight line with an arrowhead indicating its direction. Although a duplicate is sometimes added in the middle of the line, the arrowhead is usually drawn at the end of the vector.

A vector can also be thought of as having two properties, a *magnitude* and a *direction*. The magnitude of a vector \mathbf{v} , written $|\mathbf{v}|$, is its length in space, which can be found from the Pythagorean Theorem. The magnitude of the vector $\begin{pmatrix} x \\ y \end{pmatrix}$ is $\sqrt{x^2 + y^2}$. In two dimensions, you can represent the direction by the angle the vector makes relative to an axis. A vector with magnitude 1 is called a *unit vector*. In two dimensions, it is equal to $\begin{pmatrix} \sin(\alpha) \\ \cos(\alpha) \end{pmatrix}$ for some angle α .

Note

Some people prefer to write the magnitude of a vector without using boldface, so the magnitude of \mathbf{v} would be written $|v|$. Since it can cause confusion, this convention is not observed in this book.

If you specify a starting point for a vector, it is called a *position vector*. An example of an instruction for a position vector is, “From the old oak tree, take three steps north.” Here, the old oak tree is the starting point. Mathematically, you usually choose a standard starting point, such as the origin in a Cartesian plane, and then measure all position vectors from this origin. As Figure 5.2 illustrates, if you draw a position vector on a graph starting from the origin, the coordinates of the end point are the same as the components of the vector. If you label the end points O and P , then you can write the vector as \overrightarrow{OP} .

Subscripts are commonly employed to represent the components of a vector. As an example, the vector \mathbf{v} might have the components (v_1, v_2) . This is similar to the practice used in programming of identifying the components an array with indexes. In the syntax of a programming language, the components of an array \mathbf{v} might be found by $v[0], v[1], v[2]$, and so on. Note that the first item in an array is usually indexed as 0. In this book, the first item will be identified using $v[1]$ rather than $v[0]$.

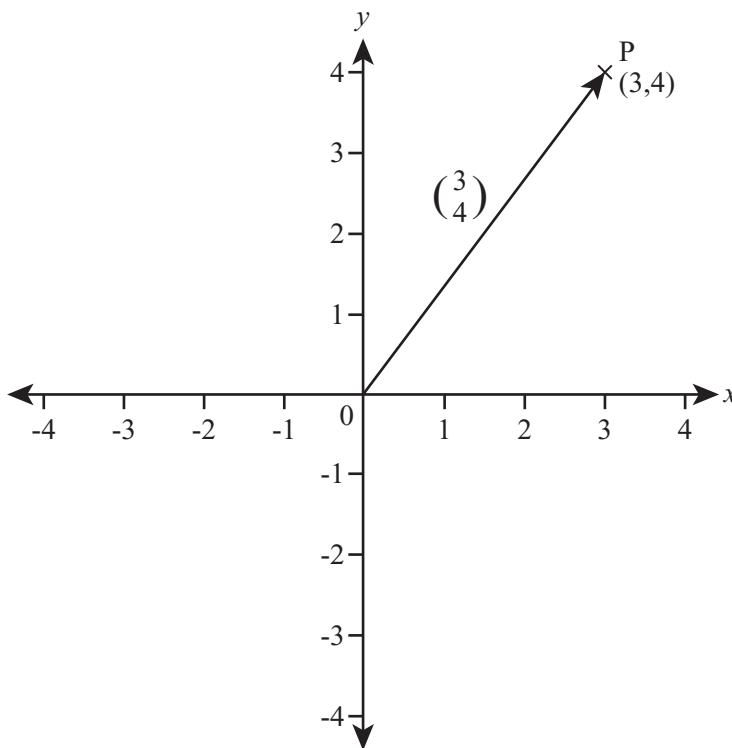


Figure 5.2
A position vector.

Vector Arithmetic

Although there are two operations that resemble multiplication that you can perform on them, two vectors can't be multiplied together in any simple way. You will see one approach in this chapter. Chapter 16 presents another. A third approach, used in programming, is *pairwise multiplication*. With this approach, the components are multiplied in pairs to create a new vector. While pairwise multiplication is not a standard mathematical tool, it will be explored in Chapter 19, where it is used on “vector-like” objects, such as colors.

Scalar Multiplication and Addition

In contrast to multiplication, addition of vectors is fairly straightforward. Likewise, you can easily multiply a vector by a scalar. Recall that a scalar value is a single value, in contrast to an array or vector.

To add two vectors together, add their components in pairs. Here is an example:

$$\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a+c \\ b+d \end{pmatrix}$$

To multiply a vector by a scalar, multiply each component by the scalar. Here is an example:

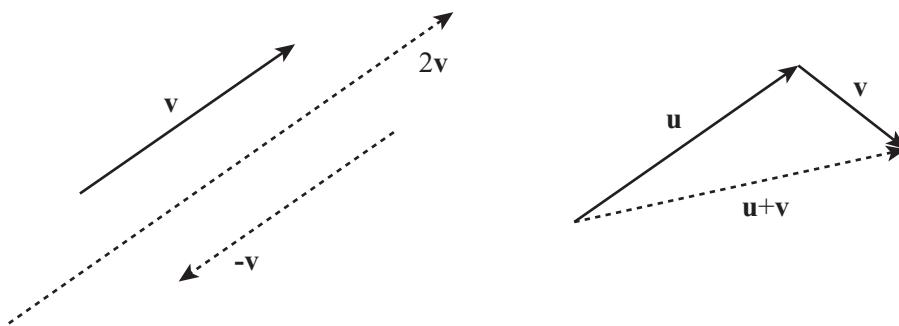
$$a \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ ay \end{pmatrix}$$

As illustrated by the vectors on the left in Figure 5.3, multiplying a vector by a scalar changes its length proportionally but leaves the direction unchanged. The vector \mathbf{v} , multiplied by 2, gives $2\mathbf{v}$. If the scalar is negative, however, the new vector faces in the opposite direction. Notice, that if \mathbf{v} is multiplied by -1 , then it becomes $-\mathbf{v}$, and the result is a line with the arrowhead at the opposite end of the first.

In addition to reversing the arrowhead, you can designate a negative change in at least two other ways. For one, if the vector \overrightarrow{AB} is \mathbf{v} , then the negative of this vector is \overrightarrow{BA} . The order of the points is changed while the arrow continues to point in the same direction. Another approach is to use a negative sign—as has already been shown. The negative of vector \mathbf{v} is expressed as $-\mathbf{v}$.

For a non-zero vector \mathbf{v} , if you divide it by its magnitude $|\mathbf{v}|$ or multiply it by the reciprocal of its magnitude $\left(\frac{1}{|\mathbf{v}|}\right)$, you get a *unit vector*. Unit vectors are sometimes indicated using the notation $\bar{\mathbf{v}}$ or $\hat{\mathbf{v}}$. A unit vector is called the *normalized vector* or *norm* of \mathbf{v} .

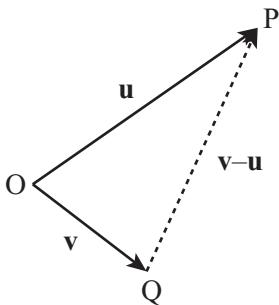
Figure 5.3 also illustrates the *sum* of two vectors. As is shown on the right side of Figure 5.3, the sum of two vectors is equal to the result of following one and then the other. (Recall that vectors don't care what route you take to the end point; they only care about the final position.) In this instance, vector \mathbf{u} takes a given direction. Then vector \mathbf{v} takes another direction. The sum is the point reached by the two vectors.

**Figure 5.3**

Adding two vectors and multiplying by a scalar.

Differences

As is shown in Figure 5.4, the difference between two vectors is slightly more involved than addition of two vectors or multiplication of a vector by a scalar. If you consider two vectors, \mathbf{u} and \mathbf{v} , to be position vectors, one of the point P and the other of the point Q, so that $\mathbf{u} = \overrightarrow{OP}$ and $\mathbf{v} = \overrightarrow{OQ}$, then the difference $\mathbf{v} - \mathbf{u}$ is the vector \overrightarrow{OP} .

**Figure 5.4**

The difference of two vectors.

The reason this works is that if you start at P, follow the vector $-\mathbf{u}$, and then the vector \mathbf{v} , you end up at Q, so $\mathbf{v} - \mathbf{u} = -\mathbf{u} + \mathbf{v} = -\overrightarrow{OP} + \overrightarrow{OQ} = \overrightarrow{PO} + \overrightarrow{OQ} = \overrightarrow{PQ}$.

Such calculations are surprisingly powerful. As an example of what they can accomplish, consider the position vector of the mid-point of the line PQ in Figure 5.4. How is this found? If you start at O, travel to P, then move half-way along the vector PQ, you reach the mid-point M. The vector \overrightarrow{OM} is equal to $\overrightarrow{OP} + \frac{1}{2}\overrightarrow{PQ} = \mathbf{u} + \frac{1}{2}(\mathbf{v} - \mathbf{u}) = \frac{\mathbf{u} + \mathbf{v}}{2}$, which is also the *mean* of the vectors \mathbf{u} and \mathbf{v} . Similarly, the position vector of the centrum of a triangle is the mean of the position vectors of its vertices.

Vector Programs

Some programming languages have innate support for vector calculations. They allow you to add arrays and multiply them by scalars using precisely the rules described in the previous sections. 3-D engines usually include functions for calculating the magnitude and norm of a vector. To demonstrate how such functions are implemented, here is a set of functions for the calculations introduced so far.

The first function, `addVectors()`, adds two vectors, **v1** and **v2**:

```
function addVectors(v1, v2)
    // assume v1 and v2 are arrays of the same length
    set newVector to an empty array
    repeat for i=1 to the length of v1
        append v1[i]+v2[i] to newVector
    end repeat
    return newVector
end function
```

The `scaleVector()` function takes a vector **v** and scales it by a factor *s*:

```
function scaleVector(v, s)
    repeat for i=1 to the length of v
        multiply v[i] by s
    end repeat
    return v
end function
```

The `magnitude()` function provides the magnitude of a vector **v**:

```
function magnitude(v)
    set s to 0
    repeat with i=1 to the length of v
        add v[i]*v[i] to s
    end repeat
    return sqrt(s)
end function
```

The `norm()` function normalizes a vector:

```
function norm(v)
    set m to magnitude(v)
    // you can't normalize a zero vector
    if m=0 then return "error"
    return scaleVector(v,1/m)
end function
```

Consider one more important calculation, the angle between two vectors. You'll see an easier way to calculate this later in the chapter, but for the moment, look back at Figure 5.4. Notice that the vectors \mathbf{u} , \mathbf{v} and $\mathbf{u} - \mathbf{v}$ form a triangle OPQ. The result of this is that you can use the cosine rule to find any of the angles from the magnitude of these three vectors. In particular, you can find the angle θ between \mathbf{u} and \mathbf{v} using the following approach:

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

Here is a function that implements this equation:

```
function angleBetween(vector1, vector2)
    set vector3 to vector2-vector1
    set m1 to magnitude(vector1)
    set m2 to magnitude(vector2)
    set m3 to magnitude(vector3)
    // it makes no sense to find an angle with a zero vector
    if m1=0 or m2=0 then return "error"
    if m3=0 then return 0 // the vectors are equal
    return acos((m2*m2+m1*m1-m3*m3)/(2*m1*m2))
end function
```

The Normal Vector

If two vectors are perpendicular, they are also called *normal*. In two dimensions, it is simple to find the perpendicular to a given vector $\begin{pmatrix} a \\ b \end{pmatrix}$. To do so, invert the vector and take the negative of one component: $\begin{pmatrix} -b \\ a \end{pmatrix}$. To see why this works, consider the vector as a position vector and remember your trick for fast rotation by 90° . Since any scalar multiple of this vector is still perpendicular to the original vector, it doesn't matter which component you make negative. Multiplying by -1 gives you the same vector in the other direction. Here is a function that accomplishes this task:

```

function normalVector(vector)
    return vector(-vector[2],vector)
end function

```

Two vectors are perpendicular if and only if the sum of the products of their components is zero. With the `normalVector()` function, the product of the x -component of each vector is $-ab$ and the product of the y -components is ab , so their sum is zero, as required. You'll look at this further in a moment.

Vectors and Scalars in Real Life

Many day-to-day quantities are best measured with vectors, and it is worth looking at them here since by doing so you gain a much clearer idea of what a vector is and why it is important. When applied to vectors, several terms used commonly in general contexts take on specific meanings. The following list provides a summary of a few of these terms.

- **Distance.** This is a scalar quantity, and it measures the length of the shortest line between two points.
- **Displacement.** If you identify two points connected by a vector, then the vector represents the displacement of the two points.
- **Speed.** This is a scalar quantity. It measures the distance something travels in a certain time.
- **Velocity.** A vector represents velocity, and velocity is the displacement in a given amount of time.
- **Mass.** Mass is measured as a scalar. It is how much force is required to move something. The direction of the movement is irrelevant.
- **Weight.** Weight is measured as a vector. It is the force required in a particular direction to keep something in the same place against the pull of gravity.

One common factor with these terms is that, informally, distance and displacement, speed and velocity, and mass and weight are often used interchangeably. For example, you describe an object as having a certain *weight* when, speaking precisely, you are describing its *mass*. Again, a train is described as traveling with a *velocity* of 100 km/hr when in fact this is its *speed*. Thinking in terms of vectors does not come naturally to most people.

However, in this context, it is important to distinguish between technical and non-technical usage. Later on, when you meet Newton's Laws, you will talk about forces changing an object's velocity. It is perfectly possible for an object to change velocity without changing speed. Think about a ball on the end of a string moving in a circle. Its direction of travel is changing constantly, but its speed is constant. Describing this in terms of vectors makes things simple. The velocity of an object is a vector, its speed is the magnitude of this vector. Thus, its velocity is changing, but its speed is not.

Vector Motion

At this point, it is beneficial to spend a little time looking at how you can use vectors to perform calculations. Pay close attention to this section. In later chapters, you will be using these ideas extensively, particularly when dealing with collision detection and resolution.

Describing Shapes with Vectors

Using vectors provides a convenient way to describe the relationship between points on a plane. For example, you have already encountered the word *parallel*. Two infinite lines are said to be parallel if either there is no point that lies on both lines, or if all points on one line are also on the other (in other words, if they are the same line). While this works well for infinite lines, it proves to be slightly more complicated for the kinds of lines you deal with most of the time, which have end points. Such lines are called *line segments*.

Parallel Lines

Using vectors, you can simplify the definition. Two lines are parallel if for any distinct points P and Q on one line, and P' and Q' on the other, there is some scalar a such that $\overrightarrow{PQ} = a\overrightarrow{P'Q'}$.

A Square

You can also use vectors to provide a recipe for creating shapes on the plane. For example, as is illustrated in Figure 5.5, given two points A and B with position vectors \mathbf{a} and \mathbf{b} , you can draw a square. You do so by first finding the normal of \overrightarrow{AB} . This is also expressed as $\mathbf{b} - \mathbf{a}$. You call this vector \mathbf{n} and assume that \mathbf{n} has the same magnitude as $\mathbf{b} - \mathbf{a}$. (If this is not so, then you must scale it to the right length—a fairly easy task.) Now construct the points C and D using the calculations $\mathbf{c} = \mathbf{a} + \mathbf{n}$ and $\mathbf{d} = \mathbf{b} + \mathbf{n}$. Notice that $\mathbf{d} - \mathbf{c} = \mathbf{b} - \mathbf{a}$ and that \overrightarrow{AB} is perpendicular to \overrightarrow{AC} . Given these preliminaries, the points A , B , D , C form a square.

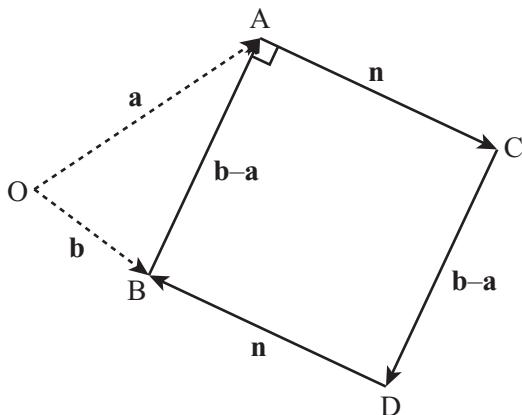


Figure 5.5

Constructing a square with vectors.

Note

With respect to Figure 5.5, notice that there are two possible squares that can be drawn on the line segment, depending on the direction chosen for the normal vector \mathbf{n} .

An Equilateral Triangle

Constructing an equilateral triangle is just as simple as constructing a square. You start with a little trigonometry. The length of the line from a vertex of an equilateral triangle to the mid-point of the opposite side is $\frac{\sqrt{3}}{2}$ times the length of a side. (This is proved using the Pythagorean Theorem.) If you have two vertices, A and B, you can construct the third vertex C using $\mathbf{c} = \frac{\mathbf{a} + \mathbf{b}}{2} + \frac{\sqrt{3}\mathbf{n}}{2}$. Here, \mathbf{n} is the normal vector to $\mathbf{b} - \mathbf{a}$. Recall that $\frac{\mathbf{a} + \mathbf{b}}{2}$ is the position vector of the mid-point of A and B.

Other Shapes and a Function

You can use similar constructions to create more complex shapes. At the end of this chapter, Exercise 5.1 challenges you to write a set of functions for creating shapes such as arrowheads and kites. The great advantage of such functions is that they can be easily parameterized to create a large number of variants on the same theme. As an example, here is a function that will create a whole family of letter As:

```
function createA(legLength, angleAtTop, serifProp, crossbarProp,
                crossbarHeight, serifAlign, crossbarAlign)
    // serifProp, crossbarHeight, crossbarProp,
    // serifAlign and crossbarAlign should be values from 0 to 1
    // angleAtTop should be in radians
    set halfAngle to angleAtTop/2
    set leftLeg to legLength*array(-sin(halfAngle), cos(halfAngle))
    set rightLeg to array(-leftLeg, leftLeg[2])
    set crossbarStart to leftLeg*crossbarHeight
    set crossbarEnd to rightLeg*crossbarHeight
    set crossbar to crossbarProp*(crossbarEnd-crossbarStart)
    add crossbarAlign*(1-crossbarProp)*
        (crossbarEnd-crossbarStart) to crossbarStart
    set serif to serifProp*(rightLeg-leftLeg)
    set serifOffset to serifAlign*serif
    set start to array(0,0)
    drawLine(start, leftLeg)
    drawLine(start, rightLeg)
    drawLine(crossbarStart, crossbarStart+crossbar)
    drawLine(leftLeg-serifOffset, leftLeg-serifOffset+serif)
    drawLine(rightLeg-serifOffset, rightLeg-serifOffset+serif)
end function
```

Figure 5.6 illustrates letters drawn using an implementation of this function. One of the most interesting side-effects of this approach is that you can take parameters of this kind and carry them across to related letters, creating a font in a similar style.

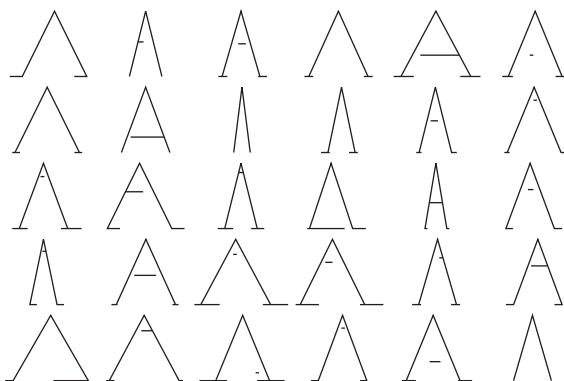


Figure 5.6

Letter As drawn with the createA() function.

Note

With respect to Figure 5.6, if you find this kind of thing interesting, you might also want to look at Douglas Hofstadter's Letter Spirit project, which explores the question of what it means for a font to be in a similar "style."

Moving from P to Q

The discussion of vectors thus far in this chapter has been building up to one of the most fundamental questions in programming, one that is especially important in game development. How do you move this from here to there? Consider Figure 5.7: the stick figure is moving from point P to point Q. Say that the stick figure's game name is Jim, and its gender is male. If Jim is at (a, b) and walks to (c, d) , what path does he follow?

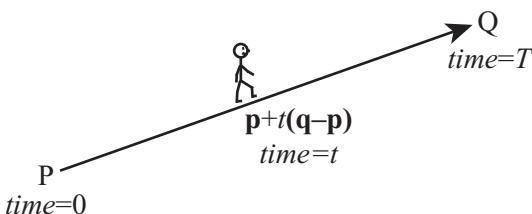


Figure 5.7

Jim's path.

To solve this problem, you can break it down into its constituent parts:

- At time 0, Jim is at $P = (a, b)$.
- At time T , Jim is at $Q = (c, d)$.
- You want to know Jim's coordinates at time t , where $0 \leq t \leq T$

To proceed, it is necessary to introduce the concepts of *speed* and *velocity*. Start by looking at the problem in terms of vectors. In the time period of length T , Jim moves straight along the vector \overrightarrow{PQ} , which is $\begin{pmatrix} c \\ d \end{pmatrix} - \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} c-a \\ d-b \end{pmatrix}$. This is called his *displacement* z . The length of this vector, $\sqrt{(c-a)^2 + (d-b)^2}$, is the distance Jim travels.

If you divide distance traveled by the time taken, you get the *speed* of the journey, measured as a unit of distance divided by a unit of time, such as a meter *per* second, or ms^{-1} . The speed is the distance traveled in each time unit. *Velocity* is the displacement vector divided by the time taken, which is the vector traveled in each time unit. Given this work, you know that Jim's velocity is $\frac{1}{T} \begin{pmatrix} c-a \\ d-b \end{pmatrix}$.

Now you want to know where Jim is to be found at a time t . To discover this, consider that he has gone a proportion t/T of the way along the vector \overrightarrow{PQ} . Considering that $t = mT$, you say that this proportion is m , Jim's position vector is given by

$$\overrightarrow{OP} + m\overrightarrow{PQ} = \begin{pmatrix} a \\ b \end{pmatrix} + m \begin{pmatrix} c-a \\ d-b \end{pmatrix} = \begin{pmatrix} mc + (1-m)a \\ md + (1-m)b \end{pmatrix}$$

Note

Speed is a little more subtle than is presented in the previous discussion. If you travel in a long circle, ending up where you started, your displacement is zero, so your total velocity is also zero. Likewise, your mean velocity is zero. However, your mean speed is not zero. Instead, it is equal to the circumference of the circle (the distance traveled) divided by the time taken. Speed is found using the length of the path traveled, not the length of the eventual vector. When moving in a straight line, however, this is immaterial.

When programming motion, it is often useful to pre-calculate values. In object-oriented programming, you usually represent a sprite on the screen with a specific object. You usually send this object from one location to another by a method that takes the new location and time as a parameter. If you develop a function to accomplish this task, it is likely to take the form of the `calculateTrajectory()` function:

```
function calculateTrajectory(oldLocation, newLocation, travelTime)
  if time=0 then
    justGoThere(newLocation)
  otherwise
    set displacement to newLocation-oldLocation
    set velocity to displacement/travelTime
    set startTime to the current time
    set stopPosition to newLocation
    set startPosition to oldLocation
  end if
end function
```

Having calculated the trajectory, whenever you want to update the position of the sprite, you can calculate the new position directly. The `currentPostion()` function accomplishes this task:

```
function currentPosition()
  set time to the current time-startTime
  if time>travelTime then
    set current position to stopPosition
  otherwise
    set current position to startPosition+velocity*time
  end if
end function
```

Note

The `currentPostion()` function method uses one variable you can manage without—`startPosition()`. Can you think of how you could do the same thing without remembering your starting position? As a hint, try counting down instead of up.

Generally, the best approach to constructing functions for determining the current position of an object is to give your object a standard speed and calculate the time to travel from there. This approach is addressed in Exercise 5.2 at the end of this chapter.

More Complicated Vector Paths

Vector motion is not useful only for straight lines. You have already seen how simple shapes can be described by a sequence of vectors. In this section, you see this activity extended to encompass curved motion created when the velocity of a particle changes as the particle moves.

Note

The word *particle* is mathematical shorthand for something that is moving in an indeterminate way. Particles are supposed to be infinitely small, although they can have properties like electrical charge or mass, depending on the circumstances. Since it is not a word that has an alternative meaning in programming, you frequently use the term *particle* to describe a moving element (a *sprite*) on the screen.

Consider Jim encountering a different scenario of motion, as illustrated in Figure 5.8. As shown on the left side of Figure 5.8, instead of approaching point Q directly, in this instance, Jim wants to skirt around it. He can do this by adding a multiple of the normal vector to his trajectory. Given this approach, Jim does not move directly along the line PQ; instead, he travels a short distance along it and a short distance perpendicular to it, to the point P'. Then, with the next step, he does the same thing, moving a little perpendicular to P'Q to P'', and so on.

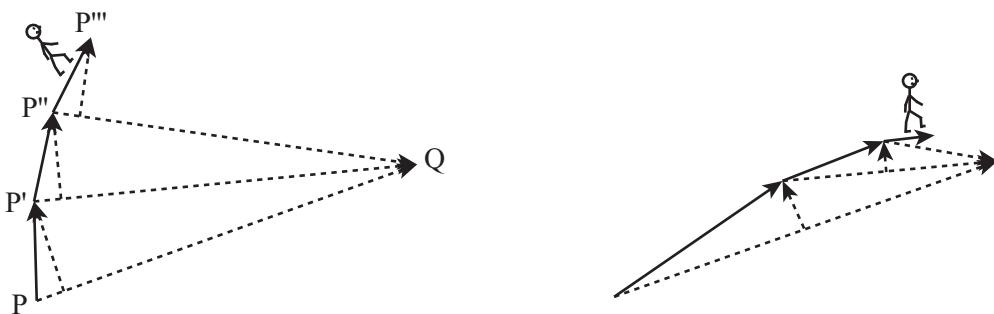


Figure 5.8

Jim skirting Point Q.

If you extend this scenario far enough, you can see that the path Jim follows can be a spiral. This becomes apparent in the path represented on the right side of Figure 5.8. Here, the tightness of the spiral depends on how large the normal vector is when compared to the inward speed. If the tangential component is zero, then Jim travels along a straight line. If it is greater than zero but small, he travels along a slightly curved path. If it is large, he spirals in gradually. If it is infinite, on the other hand, he simply moves in a circle around Q.

A Function for a Path

Generally speaking, because the time steps of a particle moving on a screen are not infinitely small, the path that the particle travels will not be mathematically accurate. Still, the behavior will resemble what has been described with respect to Jim. The `curvedPath()` function moves a particle in a curved path of this kind:

```

function curvedPath(endPoint, currentPoint,
                     speed, normalProportion, timeStep)
  set radius to endPoint-currentPoint
  if magnitude(radius)<speed*timeStep then
    set current position to endPoint
  otherwise
    set radialComponent to norm(radius)
    set tangentialComponent to
      normalVector(radialComponent)*normalProportion
    set velocity to speed*norm(radialComponent+tangentialComponent)
    set current position to currentPoint+velocity
  end if
end function

```

Wacky Paths

You can do some wacky things if you remove vectors from real life and imagine them as things in their own right. For example, with reference to the `curvedPath()` function presented in the previous section, the value of `normalProportion` is equivalent to a unit-length vector $\begin{pmatrix} \sin(\alpha) \\ \cos(\alpha) \end{pmatrix}$, where α is equal to `atan(normalProportion)`. What happens if you let this vector vary? For example, consider what happens if you give the vector its own “velocity” by varying α at a constant speed around the circle. As illustrated by Figure 5.9, if you do this, some fantastic paths result.

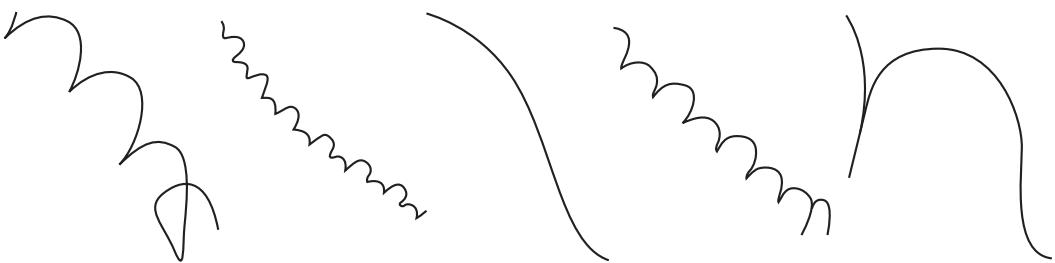


Figure 5.9

Paths generated by varying velocity.

Creating a function that generates such paths involves altering the approach used in the previous example. The `madPath()` function shows one approach:

```
function madPath(endPoint, currentPoint,
                 currentAlpha, speed, alphaSpeed, timeStep)
    set radius to endPoint-currentPoint
    if magnitude(radius)<speed*timeStep then
        set current position to endPoint
    otherwise
        set radialComponent to norm(radius)
        set newAlpha to currentAlpha+alphaSpeed*timeStep
        set tangentialComponent to
            normalVector(radialComponent)*tan(newAlpha)
        set velocity to speed*norm(radialComponent+tangentialComponent)
        set current position to currentPoint+velocity
    end if
end function
```

Thinking of an abstract quantity in terms of vectors provides you with a powerful tool. As powerful a tool as it is, however, it is often difficult to use it to arrive at the results you want. For now, remember the idea of varying the velocity vector over time. This leads to the idea of acceleration.

Vector Calculations

In this section, you examine techniques for creating and solving vector equations.

Separating Vectors into Components

Any two non-parallel vectors that share a single defined origin can be used to describe any point on a plane. If \mathbf{u} and \mathbf{v} are non-parallel vectors, then any vector on the plane can be described uniquely in the form $a\mathbf{u} + b\mathbf{v}$, where a and b are scalars. When you employ this approach to manipulating, you use \mathbf{u} and \mathbf{v} as a *basis*. One example of this involves the vectors $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, often denoted \mathbf{i} and \mathbf{j} . Since the vector $\begin{pmatrix} a \\ b \end{pmatrix}$ is equal to $a\mathbf{i} + b\mathbf{j}$, the components of the vector translate directly into the basis description. Because the basis vectors are *orthogonal* (perpendicular to each other) and *normalized* (of unit length), their relationship to each other is described as an *orthonormal* basis.

Sometimes it is useful to use a different basis. For example, as illustrated by Figure 5.10, it is often practical to think in terms of the radial and tangential components of a motion, which is the same as describing the velocity vector in terms of a new orthonormal basis directed toward the goal.

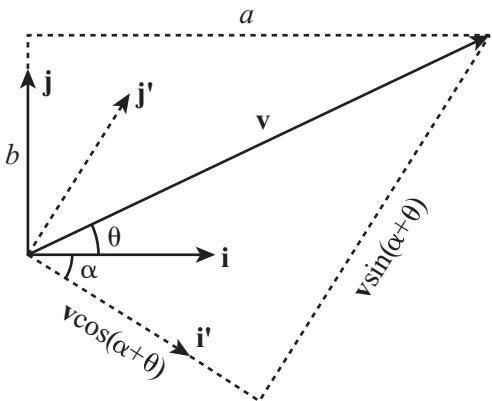


Figure 5.10

Converting a vector to a new basis.

The advantage of doing this is that often the components in these two directions can be considered independently. As is described in later chapters, when a force is directed along one vector, the velocity perpendicular to that vector is unchanged.

Note

The word *component* has a dual meaning. With a given orthonormal, if you have $\mathbf{v} = p\mathbf{a} + q\mathbf{b}$, then you can use the expression “component of \mathbf{v} in the \mathbf{a} -direction” to designate either the vector $p\mathbf{a}$ or just the number p . It is usually clear from context which of these is intended. When programming, you can define two functions, `component(vector1, vector2)` and `componentVector(vector1, vector2)`, to distinguish between them.

In Figure 5.10, the vector \mathbf{v} is to be converted from the basis \mathbf{i}, \mathbf{j} to the basis \mathbf{i}', \mathbf{j}' . The angle between \mathbf{i} and \mathbf{i}' is α and the angle between \mathbf{v} and \mathbf{i} is θ . You have drawn a rectangle around the vector parallel to the new axes, and you can see that the magnitude of the component in the direction \mathbf{i}' is equal to $|\mathbf{v}|\cos(\theta - \alpha)$. In the direction \mathbf{j}' , the magnitude is equal to $|\mathbf{v}|\sin(\theta - \alpha)$. Further, you can calculate the angles θ and α directly from the components of \mathbf{v} and \mathbf{i}' in the directions of \mathbf{i} and \mathbf{j} . For example, if \mathbf{v} is the vector $\begin{pmatrix} a \\ b \end{pmatrix} = a\mathbf{i} + b\mathbf{j}$ then $\theta = \text{atan}(b, a)$.

To summarize this discuss in terms of a function, the `switchBasis()` function takes two arguments, the vectors `v` and `k`, and returns four values. The four values are the orthonormal vectors `i'` and `j'` (where `i'` is the normalized version of `k`) and the components `a` and `b` of `v` in the directions of `i'` and `j'`. At the end, then, you know that $v = ai' + bj'$.

```
function switchBasis(vector, directionVector)
    set basis1 to norm(directionVector)
    set basis2 to normal(basis1)
    set alpha to atan(basis1[2],basis1[1])
    set theta to atan(vector[2],vector[1])
    set mag to magnitude(vector)
    set a to mag*cos(theta-alpha)
    set b to mag*sin(theta-alpha)
    return new array(basis1, basis2, a, b)
end
```

Note

Notice that you have used the two-argument version of the `atan()` function introduced in Chapter 4.

You can also write two simpler functions that find a single component in a new basis. The first is the `component()` function:

```
function component(vector, directionVector)
    set alpha to atan(directionVector [2], directionVector [1])
    set theta to atan(vector[2],vector[1])
    set mag to magnitude(vector)
    set a to mag*cos(theta-alpha)
    return a
end function
```

The second is the `componentVector()` function:

```
function componentVector(vector, directionVector)
    set v to norm(directionVector)
    return component(vector, directionVector)*v
end function
```

As becomes evident after a little study, the `componentVector()` and `component()` functions are likely to be more useful than the `switchBasis()` function.

The Scalar (Dot) Product

Although the methods you've seen so far in this chapter for finding angles and components of vectors are fairly simple, there is another way that is much more versatile. As you saw earlier, there is no natural way to multiply two vectors. However, one common operation you can perform involves the *scalar product*. As its name suggests, the scalar product is a function that combines two vectors to get a scalar answer.

The scalar product of the vectors \mathbf{u} and \mathbf{v} is written using a dot: $\mathbf{u} \cdot \mathbf{v}$. Use of the dot is why the *scalar product* has become commonly known as the *dot product*. To calculate the scalar product, you take the sum of the products of the corresponding vector components. Here is an example:

$$\begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} c \\ d \end{pmatrix} = ac + bd$$

What is the use of this type of operation? One approach to answering such a question is to consider what happens with the dot product of a given vector and the *basis vector* \mathbf{i} .

A basis vector is expressed as $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. This basis vector identifies the x -component of the vector. The dot product of this vector with a given vector \mathbf{j} gives the y -component. Similarly, when you take the dot product of vector \mathbf{v} with any unit vector \mathbf{u} , you end up with the component of \mathbf{v} in the direction of \mathbf{u} .

The dot product of two vectors \mathbf{v} and \mathbf{w} is equal to $|\mathbf{v}| \times |\mathbf{w}| \times \cos \alpha$, where α is the angle between the two vectors. This means that you can use the dot product for a number of useful calculations. For example, the dot product of a vector with itself is the square of its magnitude. (This follows from the Pythagorean Theorem.) Also, you can find the angle between two vectors by using this approach:

$$\alpha = \cos^{-1} \left(\frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}| |\mathbf{w}|} \right)$$

The dot product has the following properties:

- It is commutative: $\mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{v}$
- It is distributive over addition: $\mathbf{v} \cdot (\mathbf{u} + \mathbf{w}) = \mathbf{v} \cdot \mathbf{u} + \mathbf{v} \cdot \mathbf{w}$
- Multiplying by a scalar gives $\mathbf{v} \cdot (a\mathbf{u}) = a(\mathbf{v} \cdot \mathbf{u})$
- If two vectors are perpendicular then their dot product is zero, and vice versa.

Vector Equations

As with regular numbers, you can do algebra with vectors. In fact, when working with simultaneous equations in Chapter 3, you have already done so in a disguised way. A vector equation might be something like this:

$$a\mathbf{u} + b\mathbf{v} = \mathbf{w}$$

Any of the variables in this equation might be unknowns—the scalars a and b , or the vectors \mathbf{u} , \mathbf{v} , and \mathbf{w} . However, with such equations, the most common situation is one in which you know the values of vectors but not the values of the scalars.

As illustrated by Figure 5.11, consider a problem in which you are trying to find the intersection point of the two lines AB and CD. You know the position vectors \mathbf{a} , \mathbf{b} , \mathbf{c} , \mathbf{d} of the four points A, B, C, D, and you are looking for the position vector of the point P, where the two lines cross.

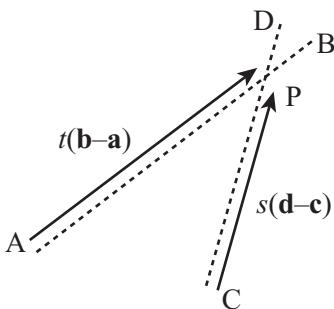


Figure 5.11

Finding the intersection of two lines.

The trick here is to try to parameterize P. In other words, you must find a way to describe P in terms of the points A, B, C, D. All you know about P is that it lies on lines AB and CD. So what can you say about a point that lies on a particular line AB? To get to such a point, you can start at the origin O, travel to A, and then travel some distance along the vector AB. Since you don't know how far you travel, you say it's some value t . Therefore, to express the problem, you write

$$\begin{aligned}\overrightarrow{OP} &= \overrightarrow{OA} + t\overrightarrow{AB} \\ &= \mathbf{a} + t(\mathbf{b} - \mathbf{a})\end{aligned}$$

What is more, because P also lies on CD, you can say that

$$\overline{OP} = \mathbf{c} + s(\mathbf{d} - \mathbf{c})$$

for some scalar s .

This gives you an equation for s and t :

$$\mathbf{a} + t(\mathbf{b} - \mathbf{a}) = \mathbf{c} + s(\mathbf{d} - \mathbf{c})$$

$$t(\mathbf{b} - \mathbf{a}) + s(\mathbf{c} - \mathbf{d}) = \mathbf{c} - \mathbf{a}$$

Now this appears to be a single equation in two variables. However, it is really two equations in disguise, because it must be true separately for both the x - and y -coordinates of the basis. This means that you can separate the vector equation into two simultaneous linear equations:

$$t(b_1 - a_1) + s(c_1 - d_1) = c_1 - a_1$$

$$t(b_2 - a_2) + s(c_2 - d_2) = c_2 - a_2$$

where a_1, b_1, c_1, d_1 are the x -components of the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$. In other words, they are the x -coordinates of the points A, B, C, D. This is similar for a_2, b_2, c_2, d_2 .

Note

It can be confusing to keep these various concepts distinct. For this reason, it is advisable to maintain a clear separation between the points, vectors, and components in a problem. Establishing this practice early on helps you when the problems get more complicated.

Explore Vectors with Code

While simultaneous equations can be solved by the same methods you used previously—assigning values to t and s —you really need only one of the values to solve the problem. You've already gone through the process of solving simultaneous equations, so there is no need to retread that ground here. The `intersectionPoint()` function uses a simplified approach to the problem to take four vectors as arguments and return the intersection of the lines between them.

```

function intersectionPoint(a, b, c, d)
  set tc1 to b[1]-a[1]
  set tc2 to b[2]-a[2]
  set sc1 to c[1]-d[1]
  set sc2 to c[2]-d[2]
  set con1 to c[1]-a[1]
  set con2 to c[2]-a[2]
  set det to (tc2*sc1-tc1*sc2)
  if det=0 then return "no unique solution"
  set con to tc2*con1-tc1*con2
  set s to con/det
  return c+s*(d-c)
end function

```

Note

In the `intersectionPoint()` function, you might be wondering why the variable name `det` is used for the value $(tc2*sc1-tc1*sc2)$. It stands for *determinant*, a term explained shortly.

Returning the Value of *t*

You can also do the same thing in a different way starting with the position vectors of A and C and the vectors \overrightarrow{AB} and \overrightarrow{CD} :

```

function intersectionTime(p1, v1, p2, v2)
  set tc1 to v1[1]
  set tc2 to v1[2]
  set sc1 to v2[1]
  set sc2 to v2[2]
  set con1 to p2[1]-p1[1]
  set con2 to p2[2]-p1[2]
  set det to (tc2*sc1-tc1*sc2)
  if det=0 then return "no unique solution"
  set con to sc1*con2-sc2*con1
  set t to con/det
  return t
end function

```

With the `intersectionTime()` function, instead of returning the point of intersection, the function returns the value of t . There is a good reason for this. As is illustrated by Figure 5.12, the values t and s are useful for more than determining the position of P . They also tell you the relationship of P to the points A , B , C , D . Suppose, for example, the value of t turns out to be 0.5. This means that P is equal to $\mathbf{a} + 0.5(\mathbf{b} - \mathbf{a})$, which is half-way along the line AB .

In general, if t is between 0 and 1, then P lies between the points A and B (with a value of 0 representing the point A , and 1 representing B). If t is greater than 1, then P lies somewhere beyond the point B . And if t is less than 0, then P lies behind the point A .

Similarly, s determines how far along the line, CD , P lies. If both s and t are in $[0,1]$ point P actually lies on the intersection of the line segments AB and CD . In any other case, P lies on the projection of the lines to infinity.

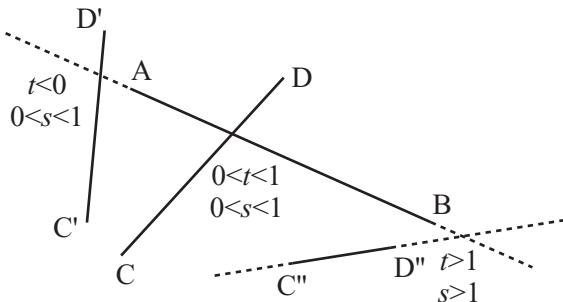


Figure 5.12

Various intersections of lines and their vector parameterization.

Intersecting Lines

In addition to the `intersectionPoint()` and `intersectionTime()` functions, you can create a slightly different function, one that finds the point where two line segments intersect. In addition to returning the value of t , this function is useful when working on collision detection.

```

function intersection(a, b, c, d)
  set tc1 to b[1]-a[1]
  set tc2 to b[2]-a[2]
  set sc1 to c[1]-d[1]
  set sc2 to c[2]-d[2]
  set con1 to c[1]-a[1]
  set con2 to c[2]-a[2]
  set det to (tc2*sc1-tc1*sc2)
  if det=0 then return "no unique solution"
  set con to tc2*con1-tc1*con2
  set s to con/det
  if s<0 or s>1 then return false
  if tc1<>0 then set t to (con1-s*sc1)/tc1
  otherwise set t to (con2-s*sc2)/tc2
  if t<0 or t>1 then return "none"
  return t
end function

```

Final Notes

In the functions given in the three previous sections, situations might arise in which the value \det is zero. In this case the lines AB and CD are parallel. If this happens, at least two possibilities arise:

- If the segments AB and CD lie along the same line (the points A , B , C , D are collinear), then either they intersect for some continuous stretch, or they are separate.
- If they don't lie on the same line, then they do not intersect at all.

Distinguishing these three possibilities isn't particularly difficult. To do so, you use another vector equation. If A , B , C , D are collinear then any one of the points can be described in terms of the other two. For example, $\mathbf{c} = \mathbf{a} + k(\mathbf{b} - \mathbf{a})$ for some k . And, if the value of k in this equation is between 0 and 1 then C lies somewhere between A and B , with the same holding true for the equivalent parameter l and the point D . As long as either C or D lies within the segment AB , the lines intersect.

Matrices

A *matrix*, like a vector, is a mathematical array representing tabular information. The plural of matrix is *matrices*. This section briefly discusses matrices and the basics of matrix arithmetic.

Matrix Fundamentals

To understand the difference between a matrix and a vector, consider first that a vector is a one-dimensional array. It is a list of components representing motion in a fixed number

of directions. Even a vector in three dimensions, such as the vector $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, is still a one-dimensional object. It has one value for each of the three directions.

In contrast, a matrix has values in two directions. To represent values, the values of a matrix are represented like a table enclosed in brackets: $\begin{pmatrix} 1 & 3 \\ -2 & 2 \end{pmatrix}$. The horizontally aligned values are *rows*. The vertically aligned values are *columns*. You can think of a matrix, then, as a table of values relating rows to columns. For example, here is a table of prices of electrical goods:

Item	Small	Medium	Large
Widgets	\$ 1.20	\$ 3.00	\$ 4.00
Gizmos	\$10.00	\$15.00	\$20.00
Whatsits	\$ 5.25	\$ 8.50	\$11.00

Looking along the Gizmo row and locating the price under the Large column, you can determine the price of a large gizmo—\$20.00.

Representing matrices as rows and columns allows you to define them, but if you then want to use matrices as parts of equations, an abbreviated form is needed. Accordingly, matrices are usually represented by small or capital letters in boldface (**G**, for example). To show that a given letter designates a given matrix, you assign the matrix to the letter, as follows:

$$\mathbf{G} = \begin{pmatrix} 1.2 & 3 & 4 \\ 10 & 15 & 20 \\ 5.25 & 8.5 & 11 \end{pmatrix}$$

To relate the rows and columns of a matrix, a times sign (\times) is used. Thus, because it has three rows and three columns, \mathbf{G} is a 3×3 matrix. With four columns, \mathbf{G} would be a 3×4 matrix. If it has the same number of rows as columns, a matrix is said to be *square*. While you're reviewing definitions, you can also define the *transpose* of a matrix. Represented with a superscript T, a transpose matrix is a matrix with the rows and columns exchanged. Here is a transpose of matrix \mathbf{G} .

$$\mathbf{G}^T = \begin{pmatrix} 1.2 & 10 & 5.25 \\ 3 & 15 & 8.5 \\ 4 & 20 & 11 \end{pmatrix}$$

The transpose of an $n \times m$ matrix is $m \times n$ in size. Notice that taking the transpose leaves the numbers on the diagonal extending from the left at the top to the bottom on the right unchanged. This is called the *leading diagonal* of the matrix.

Since a vector can be represented as a $n \times 1$ matrix, you can use the transpose notation to represent it in row form. In this case, then, the $n \times 1$ matrix becomes a $1 \times n$ matrix. As

a result, instead of writing $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, you may write $(1 \ 2 \ 3)^T$. In addition to using rounded braces to identify matrices, angled brackets are also commonly employed. Using this convention, a row vector is shown as $\langle 1, 2, 3 \rangle$.

In programming, a matrix must be represented as an array of arrays. Consider how to represent a 3×3 matrix. One array represents the rows. Within this array, there are three more arrays. Each of the arrays must have three elements, each of which represents a value in a column. So the matrix \mathbf{G} would be the array $[[1.2, 10, 5.25], [3, 15, 8.4], [4.5, 20, 11]]$.

The Determinant

The notion of a *determinant* was introduced previously in this chapter. A square matrix has an associated value called the *determinant* of the matrix. Somewhat equivalent to the length of a vector, the determinant is a fundamental feature of the matrix. To denote the determinant, vertical bars are used. The determinant of \mathbf{M} is written $|\mathbf{M}|$. If this were a function, you could express it as $\det(\mathbf{M})$.

The determinant of a matrix is the same as the determinant of its transpose. For a 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, the determinant is equal to $ad - bc$. For larger matrices, the formula is more complicated, although you can write it reasonably simply using a recursive function that takes a square matrix \mathbf{M} as an argument. The `determinant()` function provides one approach to this technique:

```
function determinant(m)
    set size to the number of elements in m
    if size=1 then return m[1][1]
    set mult to 1
    set sum to 0
    repeat for i=1 to size
        set el to m[1][i]
        set newmatrix to an empty array
        repeat for j=2 to size
            append m[j] to newmatrix
        remove the i'th element of this row
    end repeat
    add el*mult*determinant(newmatrix) to sum
    multiply mult by -1
    end repeat
    return sum
end function
```

Note

Matrices, vectors and scalars are all special cases of a general class of mathematical arrays called *tensors*, which are ways to describe a variation in values over regions of mathematical space. Working with tensors tends to be more an exercise in number-juggling than anything else. Tensors are an essential part of any physics of fields, such as general relativity or electromagnetism. They also turn up in rotational physics, where you encounter the phrase “inertia tensor.”

Matrix Arithmetic

Matrix arithmetic, like vector arithmetic, usually begins with operations involving scalars and matrices. It progresses from there to operations involving matrices and matrices. To multiply a matrix by a scalar, you multiply each element within the matrix by the scalar:

$$2 \times \begin{pmatrix} 1 & 3 \\ 4 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 6 \\ 8 & 10 \end{pmatrix}$$

To explore the simplest of operations involving two matrices, you can add two matrices of the same size by simply adding their equivalent elements:

$$\begin{pmatrix} 1 & 3 \\ 4 & 5 \end{pmatrix} + \begin{pmatrix} 2 & -1 \\ 0 & 3 \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 4 & 8 \end{pmatrix}$$

Such operations are easy to follow, but the situation changes when matrices are multiplied by matrices. Even then, however, the process is not overwhelming if expressed in simple terms. Consider, for example, multiply an $l \times n$ matrix **L** by an $n \times m$ matrix **M**.

To accomplish this, take the first element in the first row of **L** and multiply it by the first element in the first column of **M**. Do the same for each element of the first row and column of the matrices. Put the sum of these values in the top-left column of a new matrix **N**.

Continue this process for each pair of a row i of **L** and column j of **M** to get a value for the i 'th row and j 'th column of **N**. This will eventually give you an $l \times m$ matrix. Notice that this is only possible if the number of columns of **L** is the same as the number of rows of **M**.

The `matrixMultiply()` function takes two arguments, `l` and `m`, which are matrices to be multiplied. It then returns a matrix contained by the multiplied values of the two matrices.

```
function matrixMultiply(l, m)
  set n to a blank array
  repeat with i=1 to the number of rows of l
    set r to a blank array
    repeat with j=1 to the number of columns of m
      set sum to 0
      repeat with k=1 to the number of columns of l
        add l[i][k]*m[j][k] to sum
      end repeat
      append sum to r
    end repeat
    append r to n
  end repeat
  return n
end function
```

As becomes evident if you try it, multiplication of matrices is not commutative. In other words, $\mathbf{LM} \neq \mathbf{ML}$. In fact, in general it is not even meaningful to multiply matrices in the reverse order. Even for square matrices, the result of multiplying is, as a rule, different depending on the order. On the other hand, the situation differs with transposes. You can say, for example, that if $\mathbf{LM} = \mathbf{N}$, then $\mathbf{M}^T \mathbf{L}^T = \mathbf{N}^T$.

Note

Notice that the product $\mathbf{u}^T \mathbf{v}$, where \mathbf{u} and \mathbf{v} are column vectors, gives the dot product of \mathbf{u} and \mathbf{v} (or rather, a 1×1 matrix whose sole element is the dot product).

Although matrix multiplication is not commutative, it is associative: $\mathbf{L}(\mathbf{MN}) = (\mathbf{LM})\mathbf{N}$. It is also distributive over addition: $\mathbf{L}(\mathbf{M} + \mathbf{N}) = \mathbf{LM} + \mathbf{LN}$. For square matrices, matrix multiplication also preserves the determinant: $|\mathbf{LM}| = |\mathbf{L}| |\mathbf{M}|$.

For each size of square, there is a special identity matrix \mathbf{I} , which leaves other matrices unchanged under multiplication: $\mathbf{IM} = \mathbf{MI} = \mathbf{M}$. To apply this rule, it is necessary to select an identity matrix of the appropriate size for each multiplication. In each of the positions on the leading diagonal, the identity matrix has the value 1. Elsewhere, the value is 0. For example, the 2×2 identity matrix is $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

For any square matrix \mathbf{M} whose determinant is not zero, there is a unique inverse matrix \mathbf{M}^{-1} such that $\mathbf{MM}^{-1} = \mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$. For a 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, the inverse is equal to $\frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$. This gives you yet another way to solve simultaneous equations. You start by encoding a set of simultaneous equations as a matrix. Suppose the equations are as follows:

$$ax + by = p$$

$$cx + dy = q$$

This is equivalent to

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix}$$

where the vector $(x \ y)^T$ is a single unknown with two dimensions.

Now you left-multiply both sides of the equation by the inverse of the matrix:

$$\frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

Note

Note the use of the expression *left-multiply*. Because matrix multiplication is non-commutative, multiplying on the left or on the right are different operations.

So you now have a single matrix calculation to find the values of x and y , giving you the following:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{ad-bc} \begin{pmatrix} dp-bq \\ aq-cp \end{pmatrix}$$

As it happens, you can use a process similar to the one you used to solve simultaneous equations to find the inverse of matrices larger than 2×2 . You perform linear operations (multiplying by a scalar and adding linear combinations) on the rows of the matrix in order to turn it into an identity matrix. If you perform the same operations on an original identity matrix, it turns into the inverse of your original matrix.

The Matrix as a Transformation

Like a vector, a matrix is best thought of as an instruction. A vector is an instruction to move in a certain direction. If you multiply a matrix and a vector together, you get a new vector, so a matrix is an instruction for how to interpret the vector. In fact, a matrix is a transformation of space. Here are some examples of matrix transformations in two dimensions:

- If you take the matrix $\begin{pmatrix} n & 0 \\ 0 & n \end{pmatrix} = n\mathbf{I}$ and multiply it by any vector (\mathbf{v}) , you get the same vector multiplied by n . Since $(n\mathbf{I})\mathbf{v} = n(\mathbf{I}\mathbf{v}) = n\mathbf{v}$, this is not surprising. This vector represents a scale transformation, and any matrix with a determinant other than 1 (or zero) includes some element of scaling.

- If you left-multiply the matrix $\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$ by any vector, you get the vector with its x -coordinate reversed. In other words, the vector is reflected in the y -axis. Any matrix with a negative determinant is a reflection of some kind—as well as a possible scale.
- If you left-multiply the matrix $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$ by any vector, you get the vector with its x - and y -coordinates switched. In addition, the x -coordinate is reversed, resulting in a normal vector, one rotated clockwise by 90° about the origin.
- If you multiply the matrix $\begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$ by any vector, you get the vector rotated clockwise by θ about the origin. Notice that because $\cos^2(\theta) + \sin^2(\theta) = 1$, the determinant of this matrix is 1, so the vector is not scaled or reflected.
- If you multiply the matrix $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ by any vector, you get the vector skewed parallel to the x -axis by an amount proportional to the y component. This matrix represents a shear and has a determinant of 1.

Not all transformations can be represented by a matrix in this way. In particular, translations are achieved by adding a constant vector, rather than by a matrix multiplication. Every transformation centered on the origin can be represented by a matrix, and you can use matrix multiplication rules to calculate the combined results of several such transformations. For example, consider what you must do if you want to rotate a triangle by 75° clockwise about the origin, scale it to twice its size, and reflect it in the x -axis. One way to accomplish this is involves the following steps:

- Multiply the position vector of each vertex of the triangle by the matrix $\mathbf{R} = \begin{pmatrix} \cos(75^\circ) & \sin(75^\circ) \\ -\sin(75^\circ) & \cos(75^\circ) \end{pmatrix}$ to perform the rotation.
- Multiply the resultant vector by the matrix $\mathbf{S} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$.
- Multiply the resultant vector by the matrix $\mathbf{T} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.

This means that the end position of the vertex with position vector \mathbf{a} is $\mathbf{T}(\mathbf{S}(\mathbf{R}\mathbf{a})) = (\mathbf{TSR})\mathbf{a}$. To transform it in one pass, if you start by calculating the matrix \mathbf{TSR} , you can apply this matrix to all the vertices of the triangle and any others you happen to need.

As before, remember that matrix multiplication, and thus transformation, is not commutative. If you perform the three operations in a different order, you will get a different end result. To see how this makes sense, imagine turning left and then looking in a mirror. You get a different end result than when looking in the mirror and then turning the mirror image left.

Before finishing this topic, quickly have a look at what happens to the basis vectors

$\mathbf{i} = (1 \ 0)^T$ and $\mathbf{j} = (0 \ 1)^T$ under the transformation $\mathbf{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$. You can see that you have $\mathbf{Mi} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a \\ c \end{pmatrix}$ and $\mathbf{Mj} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} b \\ d \end{pmatrix}$. The columns of the matrix \mathbf{M} give

the results of transforming the basis vectors. This results in a couple of outcomes. First, if you know the result of a transformation on the basis vectors, you know everything there is to know about it. Second, you can use this result to calculate the transformation matrix.

One final useful property of a matrix is its set of *eigenvectors* and associated *eigenvalues*. These are a very handy way to characterize a matrix. In essence, an eigenvector is a vector that maps to a multiple of itself under a particular transformation. The multiple is called the eigenvalue for that vector. If \mathbf{p} is an eigenvector of \mathbf{M} , and λ is the corresponding eigenvalue, you have

$$\mathbf{Mp} = \lambda \mathbf{p}$$

Note

Strictly speaking, you should distinguish between right-eigenvectors and left-eigenvectors. While left-eigenvectors are similar to the eigenvector just explained, they are multiplied on the left of the matrix. Multiplied on the right of the matrix, right-eigenvectors are more common, and unless stated otherwise, when people talk about an eigenvector, they are talking about a right-eigenvector. As it is, however, even if the eigenvectors on left and right are different, the set of eigenvalues on each side is the same for any given matrix.

Exercises

EXERCISE 5.1

Write a set of functions such as `drawArrowhead(linesegment, size, angle)` and `drawKite(linesegment, height, width)`, which create complex shapes from simple initial parameters.

Don't just stick to these two shapes. Try making as many as you can think of. Try drawing letterforms as shown in the chapter. You might create variable fonts based on parameters of widths, heights, and angles. You might also create a simple 3-D effect that results in a "bevel" for a set of points.

EXERCISE 5.2

Working from the example given in this chapter, write a function named `calculateTrajectory(oldPosition, newPosition, speed)`, which pre-calculates the velocity vector and other necessary parameters of a movement.

The function should end up with the same set of parameters given by the `calculateTrajectory()` in this chapter.

Summary

In this chapter, you have covered the essentials of vector and matrix arithmetic and algebra, reviewing a few basic tools for thinking about space. You've seen how you can use vectors to describe positions and movement, and how to use them to perform complicated calculations like finding intersections of lines. In addition, you've also seen how matrices can be used to transform space and looked at the concept of a basis.

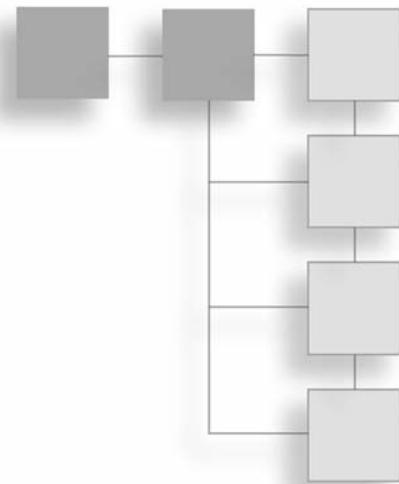
In the final chapter of Part I, you'll return to algebra and, with the help of the topics presented in this and previous chapters, look at further ways to analyze functions.

You Should Now Know

- What a *vector* is, how to describe it in terms of its components, and how to perform basic arithmetic such as adding, subtracting and scaling vectors
- The meanings of the terms *magnitude*, *norm* and *normal*, and how to calculate them for a given vector
- How to calculate the *scalar (dot)* product of two vectors, and what this operation means
- How to use combinations of known vectors to describe a known position in space
- How to parameterize a vector description to communicate a concept such as “a vector on the line AB”
- How to find the components of a vector in a different direction (putting it into a different *basis*) both geometrically and by using the dot product
- How to solve vector equations using linear simultaneous equation techniques, and how to use this to find intersections between lines
- What a matrix is and how to perform matrix calculations
- How to solve simultaneous linear equations with matrices
- How to use matrices to create multiple *transformations* of space

CHAPTER 6

CALCULUS



In This Chapter

- Overview
- Differentiation and Integration
- Differential Equations
- Approximation Methods

Overview

Covering the mathematics of limits, otherwise known as *calculus*, this chapter is the final chapter of Part I on the fundamentals of mathematics. While the demand for calculus in the programming that is dealt with in this book is limited, with respect to several of the problems concerning physics, it is necessary. Needless to say, calculus is an involved topic, and the discussion in this chapter covers only selected and essential topics that anticipate problems dealt with in subsequent chapters.

Differentiation and Integration

The two principal techniques in calculus extend the work discussed in Chapter 3, on graphs, and in Chapter 4, on gradients. Calculus uses graphs and gradients in a number of interesting and involved ways.

The Gradient of a Function

You examined gradients earlier when looking at the slope of a straight line graph, such as $y = 2x + 1$. However, the concept of a gradient is even more general than that. As illustrated by Figure 6.1, suppose you have a continuous, or smooth, curve (one that has no corners), and a point P_0 on it. Now imagine taking a series of points on the line, P_1, P_2, \dots , each of which is closer to P_0 than the one before. When you draw straight lines joining P_0 to each of these points, they gradually tend toward a particular line, which is called the *tangent* to the curve at P_0 . The gradient of the tangent is called the gradient of the curve at P_0 .

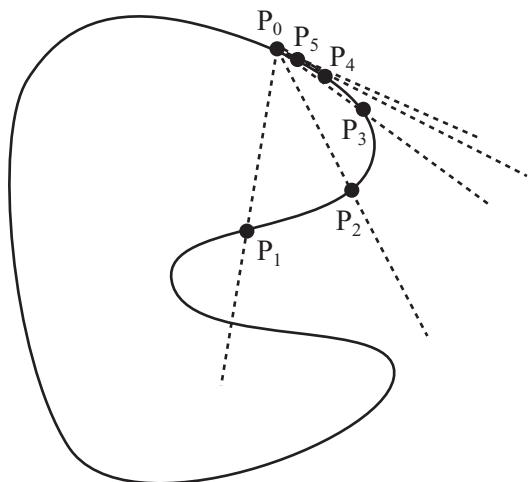


Figure 6.1

Finding the gradient of a curve at a particular point.

The Greek letter δ (delta) is often used when dealing with the concept of a limit. Accordingly, you can use the method reviewed in Figure 6.1 to find the gradient of a function at a particular value of x , say x_0 . As shown in Figure 6.2, if you take a small value for δ and find the value of $f(x_0 + \delta)$, then as δ gets smaller and smaller, a series of points on the curve results. These points are successively closer to $(x_0, f(x_0))$.

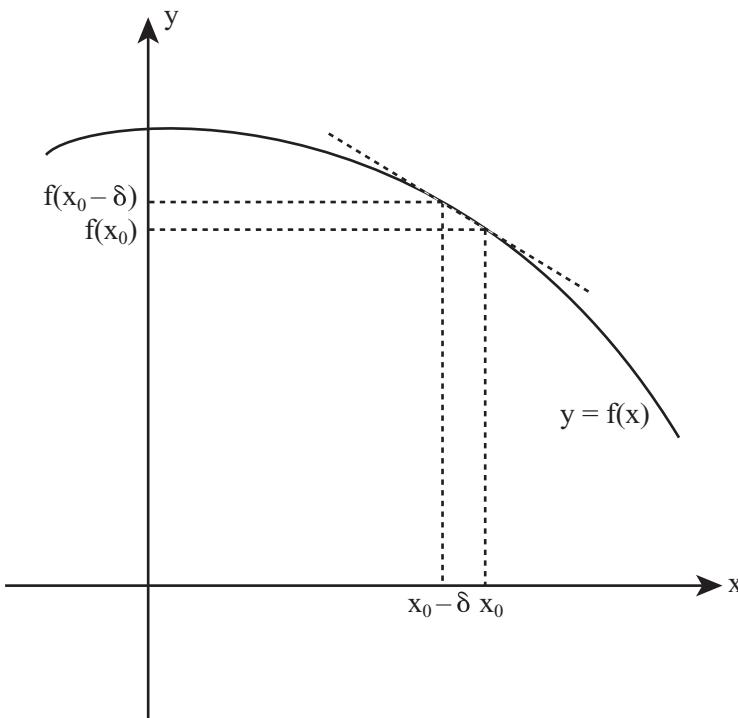


Figure 6.2

The gradient of a function.

As before, the gradient of the line joining one of these points to the required point is given by the vertical distance between them divided by the horizontal distance. The gradient, then, it is equal to

$$\frac{f(x_0 + \delta) - f(x_0)}{\delta}$$

This means that the gradient of the function is equal to the limit of this value as δ approaches zero. You express the limit as

$$\lim_{\delta \rightarrow 0} \frac{f(x_0 + \delta) - f(x_0)}{\delta}$$

To make this a little more concrete, suppose you perform these operations with a parabola. Say your function is $ax^2 + bx + c$, and you want to find the tangent at x_0 . Now you have

$$\begin{aligned} f(x_0 + \delta) &= a(x_0 + \delta)^2 + b(x_0 + \delta) + c \\ &= ax_0^2 + 2ax_0\delta + a\delta^2 + bx_0 + b\delta + c \\ &= (2ax_0 + b)\delta + a\delta^2 + ax_0^2 + bx_0 + c \end{aligned}$$

You also know that

$$f(x_0) = ax_0^2 + bx_0 + c$$

To find the gradient, you need to find the value of

$$\begin{aligned} \lim_{\delta \rightarrow 0} \frac{f(x_0 + \delta) - f(x_0)}{\delta} &= \lim_{\delta \rightarrow 0} \frac{((2ax_0 + b)\delta + a\delta^2 + ax_0^2 + bx_0 + c) - (ax_0^2 + bx_0 + c)}{\delta} \\ &= \lim_{\delta \rightarrow 0} \frac{(2ax_0 + b)\delta + a\delta^2}{\delta} \\ &= \lim_{\delta \rightarrow 0} 2ax_0 + b + a\delta \end{aligned}$$

As the value of δ approaches zero, the expression gets nearer to $2ax_0 + b$. Given this reasoning, the gradient at $x = 1$ is $2a + b$.

Differentiating

The process detailed in the previous section is called *differentiating* a function. While the example used the variable x_0 , note that x_0 is just a variable, and you can call it what you like. Differentiating begins with a function $f(x)$ and produces a new function $g(x)$. The new function represents the gradient of the function f at each value of x . You normally write $g(x)$ as either $\frac{df}{dx}$ or $f'(x)$. The expression $\frac{df}{dx}$ is read, “dee-eff by dee-ex.” You call it the *derivative* of f .

The small d is shorthand for “very small change in,” so the expression $\frac{df}{dx}$ means “the amount of change in $f(x)$ for a small change in x .” In other words, the derivative represents the rate of change of the function f with respect to the variable x . Still another way of saying this is that it represents how fast the value of $f(x)$ changes as you vary the value of x .

You can take more than one derivative. To take a second derivative, you differentiate the first derivative. This results in a new function, $\frac{d^2f}{dx^2}$. This is read as “dee-two-eff by dee-ex squared.” It is also represented as $f''(x)$. It represents the rate of change of the gradient of the function f . You can continue on, taking more derivatives of derivatives. The notation follows the same pattern.

In addition to the notation of derivatives just shown, there is another commonly used approach. This approach is often used for physical functions whose main variable represents time, or for parametric equations with the parameter t . The derivative with respect to time is represented by a dot. For a function $y(t)$, for instance, the first derivative is represented by $\dot{y}(t)$, or \dot{y} . The second derivative is represented by $\ddot{y}(t)$, or \ddot{y} . As with the first approach, as you take on more derivatives, the pattern is extended.

The process just reviewed is called *numerical differentiation* or, sometimes, *differentiating from first principles*. Using this process, you can differentiate most smooth functions. However, there are a number of shortcuts, and the following list reviews the most common of them:

1. If a is a constant, then

$$\frac{d}{dx}(af(x)) = a \frac{d}{dx}(f(x))$$

2. If f and g are functions, then

$$\frac{d}{dx}(f'(x) + g(x)) = f'(x) + g'(x)$$

3. If f and g are functions, then

$$\frac{d}{dx}(f(x)g(x)) = f'(x)g(x) + g'(x)f(x)$$

4. Identified as what is known as the *chain rule*, if f and g are functions, then

$$\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$$

5. The derivative of a constant is zero.

6. The derivative of x is 1.

A Few Applications

The chain rule illustrates an important principle about differentiation, which is that in many circumstances the dx 's and dy 's act like normal variables. For example, you can cancel them just as you would if you were dealing with a fraction. The chain rule tells you that by “cancelling out the dg 's, you get $\frac{df}{dg} \times \frac{dg}{dx} = \frac{df}{dx}$.

If you apply the chain rule or the other rules to a polynomial such as $ax^3 + bx^2 + cx + d$, you can see how powerful they are. By rule 2, you can deal with each of the terms separately. By rule 1, you can ignore the coefficients. You can start by differentiating the function $f(x) = x^n$ for some integer n . If you use rule 3, you can see that

$$\frac{d}{dx}(x^n) = \frac{d}{dx}(x^{n-1} \times x)$$

Given that by rule 5, $\frac{d}{dx}(x) = 1$, you arrive at

$$\begin{aligned} \frac{d}{dx}(x^n) &= \frac{d}{dx}(x^{n-1}) \times x + x^{n-1} \\ &= \frac{d}{dx}(x^{n-2}) \times x^2 + x^{n-2} \times x + x^{n-1} \\ &= \dots \\ &= \frac{d}{dx}(x) \times x^{n-1} + x^{n-1} + \dots + x^{n-1} \\ &= nx^{n-1} \end{aligned}$$

If you don't quite follow this, try it with a small value of n , such as 4, and you'll see that you end up with $\frac{d}{dx}(x^4) = 4x^3$.

If you now use this result, along with rules 1, 2, 5, and 6, you get the result:

$$\frac{d}{dx}(ax^3 + bx^2 + cx + d) = 3ax^2 + 2bx + c$$

In general, the derivative of a polynomial of degree n is another polynomial of degree $n - 1$.

As another example, if you want to differentiate $y = (3x + 2)^2$, you can use the chain rule. For example, suppose you start with $g = 3x + 2$, so $y = g^2$. You then have

$$\frac{dy}{dx} = \frac{dy}{dg} \times \frac{dg}{dx} = 2g \times 3 = 6(3x + 2)$$

You can check this result by multiplying out the brackets in the expression for y and differentiating the polynomial directly.

Things Derivatives Can Tell You

As is illustrated by Figure 6.3, the derivative of a function can tell you several important things:

- If the derivative of a function is zero for some value of x , then the function has what is called a *turning point* at x . This means that any *maximum* or *minimum* value of a smooth function must occur at a root of the first derivative. The reverse is not true, however. Each turning point is not necessarily a maximum or minimum. It might also be points of inflection. (Note that the plurals of *maximum* and *minimum* are *maxima* or *minima*.)
- If a function has an *asymptote* at a particular value of x , then its derivative will also have an asymptote at the same value.
- If the derivative of a function is negative, the function slopes down from left to right. If it is positive, the function slopes from right to left.

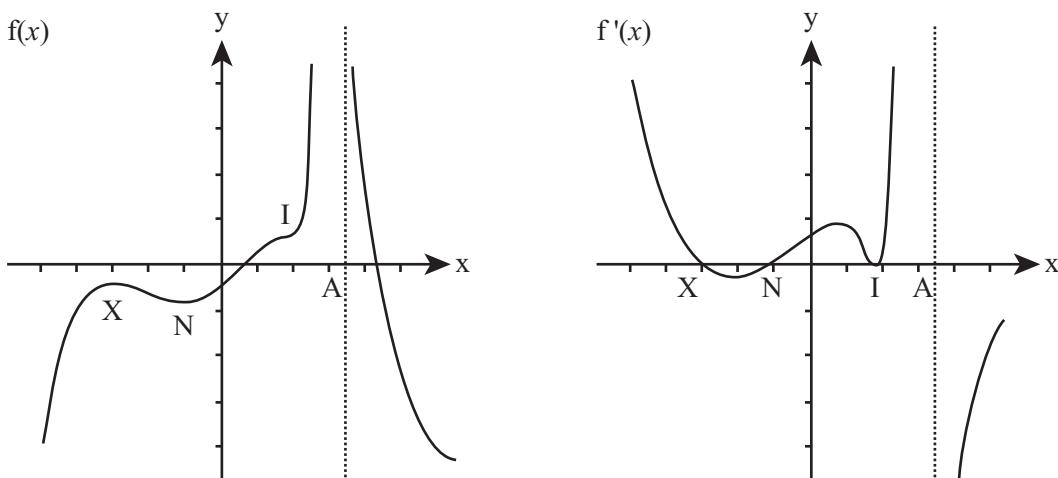


Figure 6.3

A function and its first derivative.

In Figure 6.3, the points marked N, X, and I identify minima, maxima, and *points of inflection* of $f(x)$. The terms *maximum* and *minimum* in this context refer to what is called a *local maximum* or a *local minimum*, as opposed to the *global minimum* or *global maximum*. Since the function has no global maximum or minimum, it has an asymptote at A.

If you look at the graph of the derivative in Figure 6.3, you can see that for each of the turning points, the derivative is zero. Maxima and minima are points where the derivative crosses the x -axis. At these points, the derivative touches the x -axis without crossing. These, then, are points of inflection.

If you take the second derivative, you can work out the kind of turning point you are looking at. A positive second derivative indicates a minimum. If the second derivative is negative, it represents a maximum. If it is zero, you have discovered a point of inflection.

Note

Strictly speaking, a point of inflection does not have to have a zero first derivative. It is any point at which the first derivative has a maximum or minimum. At this point, the second derivative is zero.

Differentiating Logarithms and Exponentials

Polynomials are not the only functions that can be differentiated. Many common functions have simple derivatives, including two you have met before, $\exp()$ and $\log()$. The function $\exp()$, representing the exponential function, is perhaps the most fundamental of all functions usually dealt with in calculus. Recall from Chapter 2 that the base of the natural logarithm, e , is equal to $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$. It turns out that in general

$$e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

If you substitute 1 for x in this expression, you get the original value of e . (Proving this equation is tricky, but you might want to try it for one or two special cases, such as $x = 2$.)

If you differentiate the expression, something interesting happens:

$$\frac{d}{dx} \left(\frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \right) = 0 + \frac{1}{0!} + \frac{2x}{2!} + \frac{3x^2}{3!} + \dots$$

Each term in this expression is equal to $\frac{nx^{n-1}}{n!}$. Given the definition of a factorial, $\frac{n}{n!} = \frac{1}{(n-1)!}$, the expression becomes

$$0 + \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \dots$$

This is the expression for e^x again. The derivative of e^x is e^x itself! This is something that is uniquely true of the function e^x .

While logarithms are not as elegant as exponentials, they have a surprising property of their own. It is as follows:

$$\frac{d}{dx} \log_e(x) = \frac{1}{x}$$

Recall that functions with an asymptote have an asymptote at the same position in their derivative, and this is true here. In other words, the logarithm function and the function $\frac{1}{x}$ both have an asymptote at $x = 0$. A good challenge is to see if you can prove this derivative by differentiating the logarithm function from first principles.

Differentiating Trigonometric Functions

Given the close relationship between e and the trigonometric functions, you might expect trigonometric functions to have some pleasing properties when it comes to differentiation. This turns out to be true, and to see how this is so, recall the infinite series for $\sin(x)$ and $\cos(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

If you differentiate these series, you find that $\frac{d}{dx} \sin(x) = \cos(x)$ and $\frac{d}{dx} \cos(x) = -\sin(x)$.

This means that both functions are equal to the negative of their second derivatives:

$$\frac{d^2}{dx^2} \sin(x) = -\sin(x) \text{ and } \frac{d^2}{dx^2} \cos(x) = -\cos(x).$$

You can differentiate $\tan(x)$ by using the product rule and the chain rule. Here is how this is done:

$$\begin{aligned}\frac{d}{dx} \tan(x) &= \frac{d}{dx} \frac{\sin(x)}{\cos(x)} \\ &= \frac{1}{\cos(x)} \times \frac{d}{dx} \sin(x) + \sin(x) \times \frac{d}{dx} \frac{1}{\cos(x)}\end{aligned}$$

Setting $g = \cos(x)$, you have

$$\begin{aligned}\frac{d}{dx} \frac{1}{\cos(x)} &= \frac{d}{dg} \frac{1}{g} \times \frac{dg}{dx} \\ &= \frac{d}{dg} g^{-1} \times \frac{d}{dx} \\ &= (-g^{-2}) \times (-\sin(x)) \\ &= \frac{\sin(x)}{\cos^2(x)} \\ &= \frac{\tan(x)}{\cos(x)}\end{aligned}$$

This gives you

$$\begin{aligned}\frac{d}{dx} \tan(x) &= \frac{1}{\cos(x)} \times \frac{d}{dx} \sin(x) + \sin(x) \times \frac{\tan(x)}{\cos(x)} \\ &= \frac{1}{\cos(x)} \times \cos(x) + \sin(x) \times \frac{\tan(x)}{\cos(x)} \\ &= 1 + \tan^2(x)\end{aligned}$$

As is shown in the following list, as with the $\exp()$ function, the inverse trigonometric functions have derivatives that don't include a trigonometric part:

- $\frac{d}{dx} \tan^{-1}(x) = \frac{1}{1+x^2}$
- $\frac{d}{dx} \sin^{-1}(x) = \frac{1}{\sqrt{1-x^2}}$
- $\frac{d}{dx} \cos^{-1}(x) = -\frac{1}{\sqrt{1-x^2}}$

Notice how taking the square root means that neither \sin^{-1} nor \cos^{-1} have a well-defined derivative when $|x| > 1$.

Parametric Equations and Partial Derivatives

One example of how you can “cancel” the dx terms can be seen when finding the derivative of a function defined parametrically as $y(t)$, $x(t)$. Suppose you start at a particular point with parameter t , and you vary t by a small amount. What happens? Both the x - and y -coordinates change by an amount $\frac{dx}{dt}$ and $\frac{dy}{dt}$, respectively. To find the gradient of the curve at this point using your dot notation, you can write the following equation:

$$\frac{dy}{dt}(t) = \frac{\dot{y}}{\dot{x}}$$

It is important to remember that this formula is a function of t . This proves useful, for t is usually the value you know. As an example, go back to the parabola depicted in Chapter 3 (Figure 3.6). With that parabola, the parametric formula was $x = at^2$, $y = 2at$. Taking the derivative of each function with respect to t , you have $\dot{x} = 2at$ ($= y$), $\dot{y} = 2a$, so $\frac{dy}{dx} = \frac{2a}{2at} = \frac{1}{t}$.

Another generalization of the differentiation process concerns functions in more than one variable. Consider, for example, the function $z = x^2 - 2xy + y^2$, which factors to $z = (x - y)^2$. You can plot this function on a surface in three dimensions, taking the x - and y -axes to be the horizontal plane and the z -axis to be vertical.

A few observations are important with respect to surfaces. For one, a surface no longer has a tangent line like a function in one variable. Instead, it has a *tangent plane*. A plane can be defined in two ways. It can be defined by a (3-D) vector describing its normal, or by two (3-D) vectors that lie on the plane. You can find two such vectors by using a process called *partial differentiation*. Using this approach, you find that for any point on the surface, what you look at is the two curves through the point in the x - and y -directions. If you find the gradients of these two curves, then you have the two vectors you need.

To find such a gradient, you calculate the partial derivative of the surface. This is found the same way as a standard derivative, by differentiating the formula. As you do so, however, you assume that one variable is constant. For example, the surface $z = x^2 - 2xy + y^2$ gives $\frac{\partial z}{\partial x} = 2x - 2y$, $\frac{\partial z}{\partial y} = 2y - 2x$. The character ∂ , sometimes referred to as a *German d* or *rounded d*, denotes a partial derivative.

Integration

In addition to differentiating a function, you can also go the other way, taking a function g and finding another function f whose derivative f' is equal to g . This process is called *integration*, and the function f is called the *integral* of g . There are two general forms of integration. The first is called *indefinite* integration and has to do with finding functions rather than specific values. The second concerns finding a value known as the *definite integral*.

With respect to the previous discussion, if you consider the movement from function f to function g , you can see that the function f is not unique. Since you can add any constant to a function f and leave its derivative unchanged, for some value of the parameter c , the integral of g could be any one of a family of functions $f(x) + c$. As a result, the indefinite integral is unique only apart from variation of this value c .

Integration turns out to have another meaning. The integral of g gives the area under the curve $g(x)$. This is the area of the shape drawn between the curve and the x -axis. On the face of it, this doesn't seem to make any sense. How can the value of $f(x)$ represent an area? The area of what? Just as with differentiation, how this is so is based on the concept of limits. As illustrated by Figure 6.4, the value of the integral of g at x is equal to the area of an “infinitely thin slice” of the curve area. If you inspect the curve, you can see how integration ties in with differentiation. The steeper the curve at x , the greater the area of a slice under the curve at that point.

An indefinite integral is a function. As the discussion so far implies, however, you can also use integration to find specific areas. The gray area in Figure 6.4 represents the area under the curve between two specific x -values. This area is known the *definite integral*. Rather than a function, a definite integral is just a number, a measurement of an area. It can be found by first finding the indefinite integral $f(x)$ and then plugging in the values of x at the two end points. Given this approach, the definite integral between x_1 and x_2 is equal to $f(x_2) - f(x_1)$. Notice that the annoying constant value c , which you are forced to introduce for the indefinite integral, cancels out neatly in this calculation. As a result, the definite integral is unambiguous.

You've got this far without using any notation specific to integration, as it is, integration is represented by the symbol \int . As with differentiation, you must mark the variable over which you are integrating using the letter d , which as before stands for “a very small change in.” So the integral of the function $g(x) = 2x - 5$ is written as $\int g(x)dx = \int (2x - 5)dx = x^2 - 5x + c$, meaning that for a very small change in x , g increases by $x^2 - 5x$.

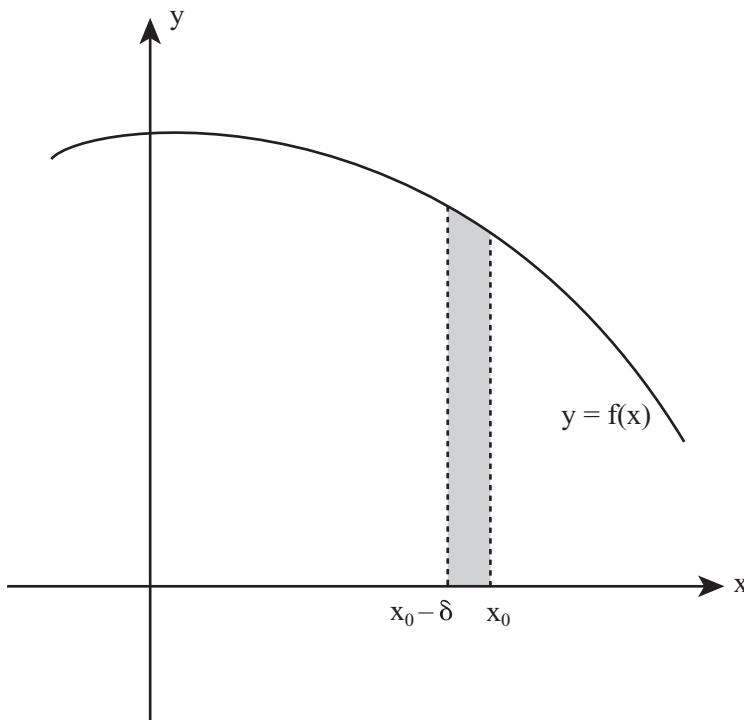


Figure 6.4

Integration is finding the area under a curve.

Definite integrals are written the same way, with the exception that start and end values for the main variable are placed at either end of the integration sign, indicating the interval of integration. Here is an example of definite integration over the interval $[1,3]$:

$$\begin{aligned}
 \int_1^3 g(x) dx &= \int_1^3 (2x - 5) dx \\
 &= [x^2 - 5x]_1^3 \\
 &= (3^2 - 5 \times 3) - (1^2 - 5 \times 1) \\
 &= -2
 \end{aligned}$$

This calculation gives the area under the line $y = 2x - 5$ between $x = 1$ and $x = 3$.

Differential Equations

Particularly in areas of physics, it commonly happens that while you don't know a precise formula for some function $y(x)$, you do know something about a relationship between x , y , and the derivative(s) of y . This relationship is called a *differential equation*. Strictly speaking, to distinguish it from a partial differential equation in more than one variable, it is called an *ordinary differential equation*, or an ODE.

Characteristics of Ordinary Differential Equations

Here are some examples of ordinary differential equations:

- $y' = 2x$
- $(y'')^2 + 2y = 0$
- $y' - 2xy + y^2 - x = 0$

What distinguishes a differential equation from other equations? Generally what you are trying to discover is an algebraic solution, which is the function $y(x)$. If that fails, you can at least try to find one or more particular values of y or a function that approximates the differential equation. If the function approximates a differential equation, it is known as a *numeric* solution.

Very few differential equations are easy to solve. Among those that are, consider any differential equation of the form $y' = f(x)$. Solving this is simply an exercise in integration (which is not always possible algebraically, however). A common example of this kind of problem concerns a snowplow. Suppose that the speed of a snowplow is inversely proportional to the depth of snow and that the plow is trying to clear snow that is still falling. You want to find how long it takes for the plow to travel a certain distance x .

Suppose the snow is at an initial depth of s and is falling at a rate of σ . You know that the speed of the plow is inversely proportional to the amount of snow, so you have

$$\frac{dx}{dt} = \frac{k}{s + \sigma t}$$

You can integrate the right-hand side to get

$$x = \frac{k}{\sigma} \ln(s + \sigma t) + c$$

Suppose, however, that at $t = 0$ you have $x = 0$. You can then say that $c = -\frac{k}{\sigma} \ln(s)$. This becomes

$$x = \frac{k}{\sigma} (\ln(s + \sigma t) - \ln(s)) = \frac{k}{\sigma} \ln \left(1 + \frac{\sigma}{s} t \right)$$

Now to find t in terms of x , you invert this equation to get

$$t = \frac{s}{\sigma} \left(\exp \left(\frac{\sigma x}{k} \right) - 1 \right)$$

Notice that a part of this calculation involved the unknown constant of integration, c . Before you could fully solve the problem, you needed to know an additional piece of information about the problem, namely the starting position of the plow. As you've seen before, this implies that the differential equation has a family of solutions, with a valid solution to the equation for any particular choice of c . This is a general feature of differential equations.

Setting integration parameters (or any other unknowns) by means of initial conditions of the system makes sense because a differential equation is like a vector. It tells you nothing about where you are now. Instead, it only tells you that, "if you're currently here, doing this, then you'll shortly be there, doing that."

As Figure 6.5 illustrates, representing a differential equation diagrammatically shows how its behavior is analogous to that of a vector. Suppose your differential equation is the third of the equations in the list at the start of this section, $y' - 2xy + y^2 - x = 0$. You can graph this equation by choosing any particular point (x, y) and calculating the value of y' at that point. Then you graph a short line at this point with the appropriate gradient. Repeating this several times generates something like the graph represented by Figure 6.5.

Each of the lines in Figure 6.5 represents a particular solution of the ODE, while the general solution continues to have one or more unknown parameters introduced by the integration process. Notice that this graph shows certain patterns. Regardless of the initial conditions you choose, all the functions $y(x)$ converge for high values of x on the line $y = 2x$. This is to be expected given the range of values chosen. For large values of x and y , the linear term in x becomes less significant. Other differential equations can involve loops, singularities, strange attractors, and many other interesting phenomena.

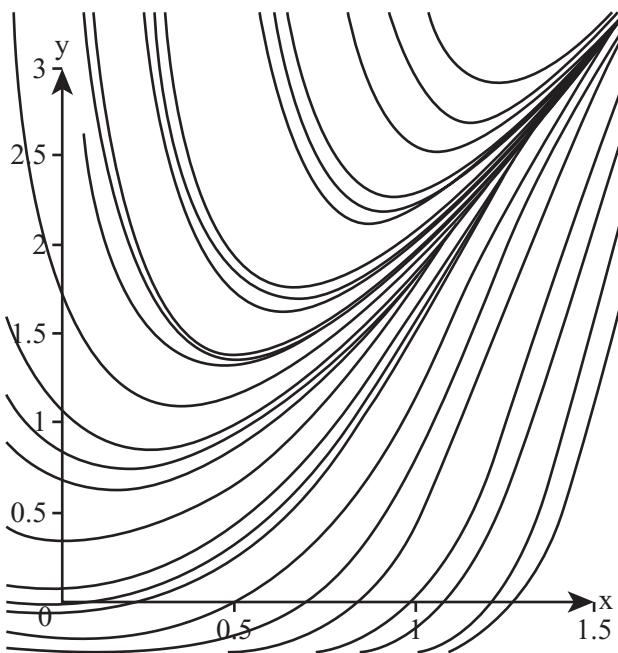


Figure 6.5

A numeric plot of the differential equation $y' - 2xy + y^2 - x = 0$.

Solving Linear ODEs

As has been mentioned previously, while the majority of ODEs cannot be solved algebraically, a significant number can be. Among these are linear ODEs, which are of the form

$$f_0(x)y + f_1(x)y' + f_2(x)y'' + \dots + f_n(x)y^{(n)} = 0$$

(where $y^{(n)}$ is the n th derivative of y and all the f 's are functions of x).

As you initially inspect this generalized example of a linear ODE, suppose first that all the functions are just constants. The key to solving an equation like this is the `exp()` function. Since this function is its own derivative, any function of the form $y = e^{rx}$ has a derivative that is a multiple r of itself. For example, suppose your differential equation is $2y - 5y' + 3y'' = 0$. You can try the solution $y = e^{rx}$ and see what happens. If you plug this function back into the differential equation, you get

$$2e^{rx} - 5re^{rx} + 3r^2e^{rx} = 0$$

Because e^{rx} is always positive, you can factor it out, giving you $2 - 5r + 3r^2 = 0$. This is a quadratic equation, and its solution is $r = 2$ or $\frac{1}{3}$.

On the other hand, solving an ODE can require a lot more effort because there are two variants that also work. If you multiply a valid function by a constant, $y = Ae^{rx}$, this factor A cancels out, so any multiple A will still give a valid solution to the differential equation. Similarly, if you add a constant c to the exponent, $y = Ae^{rx+c}$, this constant disappears when differentiating, so again it doesn't affect the validity of the solution. Therefore, your family of ODEs has two parameters that are affected by the initial conditions. (This is discussed in a more concrete form in Chapter 16.)

Approximation Methods

One important technique for which calculus is very useful is in finding approximate solutions to a problem. Even outside the realm of differential equations, many equations cannot be solved by algebraic methods. These are not necessarily complicated equations. Consider, for example, that an equation as simple as $\sin(x) = x$ needs a non-algebraic solution.

Since any equation in one variable can be reduced to the form $f(x) = 0$, your problem becomes one of finding the roots of the function $f(x)$. While there are a number of methods that can be used for this, none of them is foolproof. However, when dealing with a relatively smooth function, f , they can work well.

Bracketing Methods

As illustrated in Figure 6.6, a simple approach useful in some circumstances involves homing in on, or bracketing, a solution using the *bisection method*. In Figure 6.6, you can see that you have found two values of x for which $f(x)$ lies on different sides of the x -axis. There is a simple way to check for this. Calculate $f(x_1)f(x_2)$. If the result is negative, one value must be positive and the other negative. If the result is positive, either they are both positive or both negative. If the result is exactly zero, then one of the values is an exact root. When you have found two such values, you know that, for a continuous function, somewhere between them there must be a root.

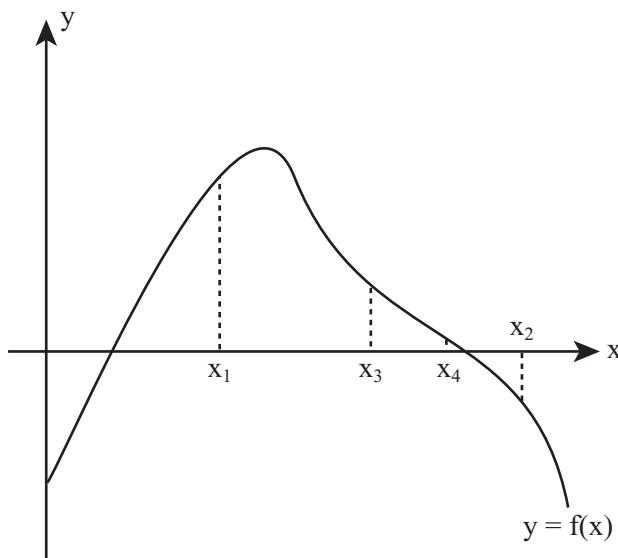


Figure 6.6
Homing in on a root.

This gives you a simple algorithm for getting arbitrarily close to (bracketing) a root. If $f(x_1)$ and $f(x_2)$ lie on opposite sides of the x -axis, you can look at $x_3 = \frac{1}{2}(x_1 + x_2)$. By checking $f(x_1)f(x_3)$ and $f(x_3)f(x_2)$, you can narrow down the position of the root to one or the other half of the interval $[x_1, x_2]$. Because halving is an exponential process, the binary method homes in on the root very rapidly. In addition, you can specify from the start exactly how closely you want to find the approximation.

The `bisectionMethod()` function shows how to program the bisection method. Since they depend on how the function is specified, the details of how you calculate the value of $f(x)$ are left vague.

```
function bisectionMethod(func, x1, x2, resolution)
  // check that f(x1)*f(x2)<0 to get the process going
  set f1 to calculateValue(func, x1)
  set f2 to calculateValue (func, x2)
  if f1*f2>0 then return "may be no root in range"
  // if you are already very near to the solution then return one value
  if abs(x1-x2)<=resolution then return x1
  // (in some circumstances you might choose to return
  // the value of x giving a positive value of f,
  // so you'd return x1 if f1>0, x2 otherwise.)
```

```

set x3 to (x1+x2)/2
set f3 to calculateValue (func, x3)
if f3=0 then return x3
if f1*f3<0 then return bisectionMethod (func, x1, x3, resolution)
return bisectionMethod (func, x3, x2, resolution)
end function

```

A faster bracketing method is called *Regula Falsa* (“false position”). This method takes advantage of the fact that, generally, you’re more likely to find the root nearer to the bracket with the smallest absolute value. Regula Falsa weights the choice of the new test point by using a method illustrated in Figure 6.7.

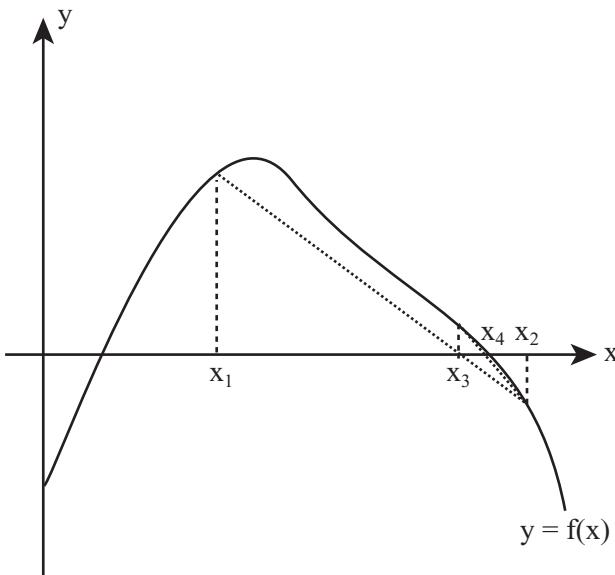


Figure 6.7
The Regula Falsa method.

Suppose your current brackets are a and b . You can find the new point c by choosing $c = \frac{af(b) - bf(a)}{f(b) - f(a)}$. Implementing this on the computer involves a simple variant on the `bisectionMethod()` function.

Bracketing methods have their limitations. The most obvious is that if the function has two roots close to one another, unless you are extremely lucky in your initial choice of x_1 and x_2 , you’re going to miss them. The result is that you will never find a double root.

The methods also fail if there is a discontinuity, as with the function $y = x^{-1}$. Rather than a root, this function has a discontinuity at $x = 0$. When this is the case, the function returns the discontinuous point.

Finding suitable initial values for an arbitrary function can be painstaking. It is essentially a matter of trial and error. However, in many circumstances, particularly with a function on which you can perform some preliminary analysis, bracketing works just fine. Also, it even works if you don't know the function $f(x)$ that you are using. Consider, for example, a situation that involves an arbitrary list of heights generated by the function at particular values of x .

Gradient Methods

For reasonably smooth functions—and especially those that don't have a maximum or minimum near the x -axis—there is an algorithm that avoids some of the difficulties associated with bracketing methods. This is called the Newton-Raphson method. This method takes advantage of a trick illustrated in Figure 6.8. The trick begins with taking the tangent of the function at a particular point and projecting it onto the x -axis. The point where the tangent intersects the axis is much closer to the root than the original value of x .

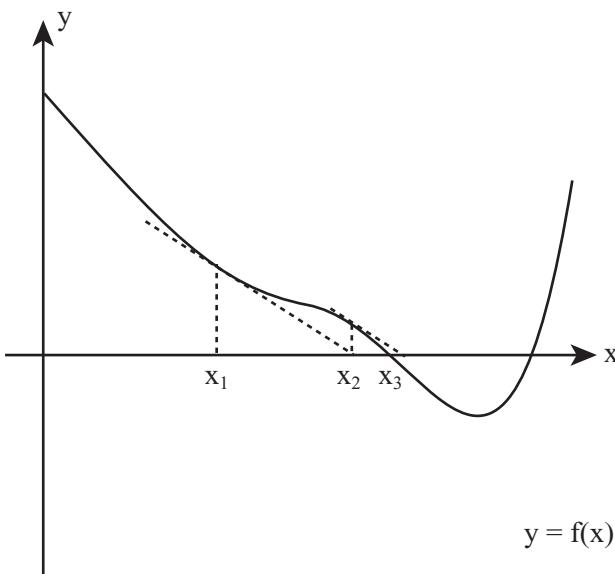


Figure 6.8

Using the tangent of a function to move closer to the root.

It may not be clear that the trick shown in Figure 6.8 applies generally. It might seem, for example, that it is specific to the kind of function drawn here. As shown in Figure 6.9, to some extent, this is true. If the value of x you initially choose happens to be near a turning point, or if there is a turning point between x and the root, then you are going to hit problems. But when your initial value of x is sufficiently close to a root—which for most functions is still quite far away—this method works extremely well. What is more, it is much faster than any other common method.

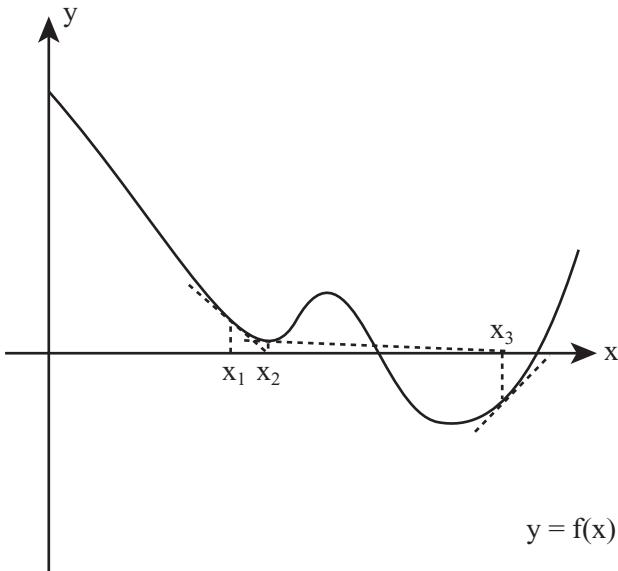


Figure 6.9

An example of a situation where the Newton-Raphson method fails to find a nearby root.

Using the Newton-Raphson method is quite simple and can be calculated using standard graph techniques. The gradient of the tangent at x_1 is given by $f'(x_1)$, so the equation of the line is $y - f(x_1) = f'(x_1)(x - x_1)$. This crosses the x -axis at a point x_2 , where $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$.

This `newtonRaphson()` function provides an iterative approach to the Newton-Raphson method that is relatively easily programmed.

```
function newtonRaphson(func, deriv, x1, resolution)
    set f1 to calculateValue (func, x1)
    if abs(f1)<resolution then return x1
    set g1 to calculateValue (deriv, x1)
    if g1=0 then return newtonRaphson(func,
        deriv, x1+resolution, resolution)
    set x2 to x1-f1/g1
    return newtonRaphson(func, deriv, x2, resolution)
end function
```

The only major disadvantage of the Newton-Raphson method is that unlike the other examples you've seen, it is impossible to apply unless you know the derivative of the function. If you're dealing with a complicated or random function, this is not going to be possible.

A final example of using gradients is the *secant* method. It has an advantage over the Newton-Raphson method because it is somewhat less involved to code. Figure 6.10 illustrates the secant method.

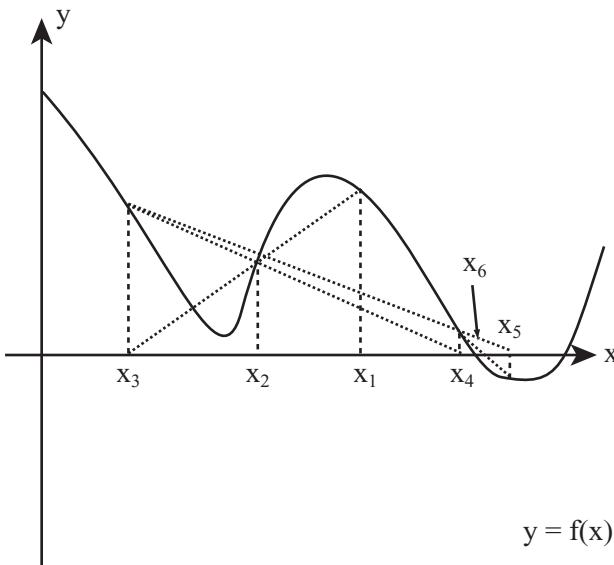


Figure 6.10
The secant method.

Essentially, the procedure is identical to Regula Falsa, but you remove the necessity of having values of $f(x)$ of opposite signs. Instead, working on the assumption that the gradient of the function is linear near your test points, you extrapolate the line down to the x -axis, just as in Newton-Raphson.

Again, this method may hit problems for a function with a turning point near the x -axis, and like the Newton-Raphson method, it is easy to miss a root near the starting point while catching one further away. Nevertheless, the secant method is probably the most reliable (and fast) method if all you want to do is find some root. If you want a root within a particular region (say within the interval $[0,1]$), you might be better off using a bracketing method.

As a piece of advice concerning programming approximation method, be sure to include some checks and balances. Such methods involve iteration, and you don't want to get stuck in an infinite loop looking for a root that doesn't exist.

Exercises

EXERCISE 6.1

Create a function that draws a diagram of a differential equation similar to Figure 6.5.

The function used to draw this figure was based on the `drawGraph()` function from Chapter 3. See if you can adapt this function to create this kind of graph.

EXERCISE 6.2

Write a function to implement the secant approximation method.

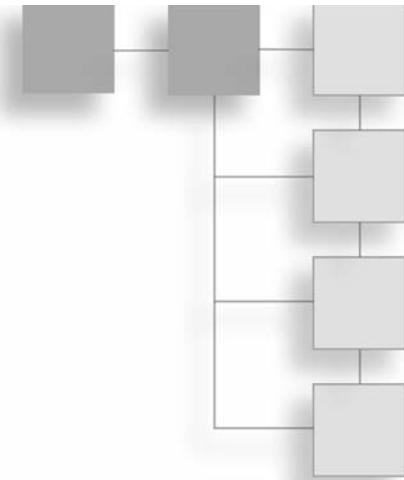
This should be a reasonably straightforward extension of the `bisectionMethod()` function in this chapter.

Summary

This chapter introduced some key terminology relating to calculus. In this book, it is not necessary to work exhaustively with integrals or solving differential equations, but the concepts are an essential part of many of the topics covered later on, especially in Part III. For this reason, this chapter has concentrated on examining the theory behind differentiation and integration. It has shown some applications using differential equations to solve physical problems. It has also examined how to employ bracketing and gradients to create fast approximation methods for solving equations. This chapter also concludes the first, largely theoretical, part of the book. In Part II you'll begin to put these ideas into practice by using them to simulate physically realistic motion.

You Should Now Know

- The meaning of the terms *integration* and *differentiation*, *derivative*, *partial derivative*, and *indefinite integral*
- How to differentiate and integrate simple functions, such as polynomials
- The derivatives of the exponential and trigonometric functions
- How to recognize a differential equation, and how to solve certain simple examples
- Using bracketing and gradient methods for finding the approximate solutions to equations and their advantages and disadvantages



PART II

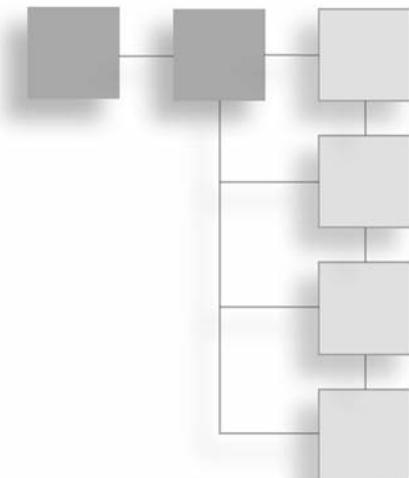
ESSENTIAL TOPICS IN PHYSICS

The second part of the book covers the basic physics of motion. Among these are calculating how a particle moves in the real world and translating it onto a computer. At the heart of this material is the understanding of vectors that you examined in Chapter 5, which allowed you to turn one complex problem into a number of simpler ones. At the end of these few chapters you will reach the stage where you can explore how to construct a game of pool.

This page intentionally left blank

CHAPTER 7

ACCELERATION, MASS, AND ENERGY



In This Chapter

- Overview
- Ballistics
- Mass and Momentum
- Energy

Overview

With this chapter, you leave the world of abstract mathematics and enter the world of physical objects, otherwise known as *mechanics*. However, this chapter depends on a few topics examined previously. In Chapter 5, for instance, you looked at some simple examples of vector motion. The concepts of velocity, speed, displacement, distance, and time were introduced. In this chapter, you will add the concept of acceleration. With acceleration, among other things, you can simulate the motion of a particle under gravity.

Ballistics

The area of engineering physics called *ballistics* involves studying particles moving with a constant acceleration. In most cases, the particles are regarded as *projectiles*, and as the name implies, projectiles fly through the air and fall under gravity. Strictly speaking, bodies moving through the air also experience resistance, but this is beyond the scope of this chapter. Anything that affects the speed of the projectile as it moves through the air is affecting its acceleration.

Acceleration and Deceleration

Velocity is the rate of change of a particle's position. Acceleration is the rate of change of velocity. Like velocity, acceleration is a vector. Its components are measured in units of velocity divided by units of time. If your units of distance are meters and your units of time are seconds, how things are measured is then set. Distance and displacement are measured in meters (m). Speed and velocity are measured in meters per second (ms^{-1}). Acceleration is measured in meters per second per second, or meters per second squared (ms^{-2}).

Note

Unlike the pairs of *speed/velocity* and *distance/displacement*, there is no equivalent pair for acceleration. The same word is used for both the rates of change of speed (a scalar quantity) and of velocity (a vector). Generally, how the word is intended is clear from the context of its use. In this book, reference is almost always made to the vector quantity.

Because acceleration is a vector, it can be measured in any direction. If a particle is moving in a straight line and its speed is decreasing at a constant rate, the particle is experiencing a constant acceleration in the direction opposite to its motion. This is somewhat different from the common usage of the word “accelerate,” which refers only to increasing speed. When the speed of a car is decreasing, for example, you usually say that it is *decelerating*. You might say that its deceleration is the negative of its acceleration. However, this doesn't really tell you a great deal, and in physics, the term *deceleration* is not usually used.

The Equations of Motion Under Constant Acceleration

With \mathbf{a} and \mathbf{u} representing vectors, if a particle's acceleration \mathbf{a} is constant, for each unit of time, the velocity changes by \mathbf{a} . If the particle starts with a velocity \mathbf{u} , then after time t , its velocity \mathbf{v} will equal $\mathbf{u} + \mathbf{a}t$. What will its position be at that time?

You can calculate the position quite simply. Take the mean velocity over the time period t . If at time t the velocity is \mathbf{v} , then the mean velocity is $(\mathbf{u} + \mathbf{v})/2$, and the displacement \mathbf{s} is $t(\mathbf{u} + \mathbf{v})/2$. Substituting the previous value for \mathbf{v} you get

$$\begin{aligned}\mathbf{s} &= \frac{t(\mathbf{u} + \mathbf{v})}{2} \\ &= \frac{t(\mathbf{u} + \mathbf{u} + \mathbf{a}t)}{2} \\ &= \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2\end{aligned}$$

If you look back at Chapter 6, you can see that this result makes perfect sense. If you differentiate with respect to time, you get

$$\frac{ds}{dt} = \mathbf{u} + \mathbf{a}t = \mathbf{v}$$

$$\frac{d^2\mathbf{s}}{dt^2} = \mathbf{a}$$

This fits your definitions of velocity and acceleration. Velocity is the rate of change of displacement, and acceleration is the rate of change of velocity. Notice also that if acceleration is zero, the formulas reduce to a simple version for motion with a constant speed, $\mathbf{s} = \mathbf{u}t$.

You now have three formulas relating the values \mathbf{s} , \mathbf{u} , \mathbf{v} , \mathbf{a} and t . Each formula relates four of the five quantities. By artful substitution you can derive still more equations. A fourth equation relates \mathbf{v} , \mathbf{u} , \mathbf{a} and \mathbf{s} . A fifth relates \mathbf{v} , \mathbf{a} , \mathbf{s} and t . In the end you have a complete set of formulas for the five quantities:

1. $\mathbf{v} = \mathbf{u} + \mathbf{a}t$
2. $\mathbf{s} = t(\mathbf{u} + \mathbf{v})/2$
3. $\mathbf{s} = \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2$
4. $\mathbf{s} = \mathbf{v}t + \frac{1}{2}\mathbf{a}t^2$
5. $v^2 = u^2 + 2as$

Since there is no simple way to multiply vectors together, the fifth formula doesn't make very much sense in terms of vectors; it is presented in terms of scalars, and it assumes that the vectors \mathbf{s} , \mathbf{a} , \mathbf{v} , and \mathbf{u} are all collinear. It is really just a restatement of the previous formulas. As you can see, Equation 3 and Equation 4 can be derived from Equation 1 and Equation 2.

Using these formulas, you can calculate any one of the parameters in terms of any three of the others. Say that you know the current velocity and acceleration of a particle and how long it has been moving. You can calculate both its relative initial position and its initial velocity:

If $\mathbf{v} = \mathbf{u} + \mathbf{a}t$,
 then $\mathbf{u} = \mathbf{v} - \mathbf{a}t$
 and $\mathbf{s} = t(\mathbf{u} + \mathbf{v})/2$.

It's unlikely you will need to do this particular calculation often. Since you usually know \mathbf{u} , \mathbf{a} and t , it is the velocity or position that you need to find. The formulas allow you to find this directly.

As a general approach to getting the position, the `calculatePosition()` function takes four arguments and returns the position:

```
function calculatePosition(initialPosition,
                           initialVelocity,
                           acceleration, time)
    return initialPosition + initialVelocity * time
           + acceleration * time * time/2
end function
```

Acceleration Due to Gravity

The story goes that Galileo wondered if a heavy ball and a light ball would fall at different speeds. He climbed to the top of the leaning Tower of Pisa bearing a heavy ball and a light ball, dropped the two balls off the top, and watched. They fell at the same speed. By this experiment, he proved that the speed at which an object falls is independent of its weight. As it stands, however, he never performed such an experiment. Instead, he stayed in his studio and arrived at his law by pure logic. It is worth looking at his proof because it's very ingenious.

Suppose you have two objects, A and B, where A is heavier than B. Now suppose that it is true that heavier objects fall faster than light ones. If you tie A to B and drop them from a height, B will fall more slowly, so it will act as a drag on A, like a parachute. This would mean that if A and B were tied together, they would fall more slowly than A. But this makes no sense, because A and B tied together are heavier than A, which should fall faster than A. So something is wrong, and it must be the original hypothesis.

Note

This kind of argument is known as a *reductio ad absurdum* (or proof by contradiction, as it is called in pure mathematics). You prove that something is true by assuming the opposite and showing that it leads to a contradiction. It's a very powerful technique, although many people feel it's less elegant than a direct proof.

But how exactly does a ball fall through the air? What does gravity do? This is a complicated question, one that is fundamental to science and to which science does not yet have a final answer. For now, you can give a simple answer. As long as the ball doesn't significantly change height, while it is flying it experiences a constant downward acceleration. The acceleration is due to gravity, \mathbf{g} . Ignoring the effects of air resistance, this is the only way that the speed and direction of the ball change. At sea level on earth, $|\mathbf{g}|$ is approximately 9.8 m/s^2 . However, to simplify calculations, you'll use the value of 10 in this chapter.

Using this beginning, you can make some simple calculations straight away. For example, consider a question. If you throw a ball straight upward with a speed of 5 ms^{-1} , how high will it go? There is a small trick here. When the ball reaches the top of its motion, its speed is zero. This means you can use your equation of motion:

$$v^2 = u^2 + 2as$$

$$0 = 5^2 - 2 \times 10 \times s$$

$$s = 25/20 = 1.25\text{m}$$

It's worth looking at exactly what you did here. The values s , u , v , and a are all vector quantities, but in this calculation you have treated them as plain numbers. Your only concession to their vector nature is that you used the negative value -10 for \mathbf{g} . This is done because it is in the opposite direction from the other quantities. This is possible because the whole of the motion takes place in a single vertical line, a one-dimensional space. The vectors have just one component, a single-directed number.

In case this seems to be stressing the point too much, think about another question. How fast will the ball be traveling when you catch it again? This time, you are looking for a value of v such that $s = 0$. You can use the same formula as before:

$$v^2 = u^2 + 2as = 25 + 0$$

Solving, you arrive at

$$v = \pm 5$$

Of course, when $v = 5$, this is the beginning of the motion, and by definition, $s = 0$. At the end of the motion, $v = -5$, which is to say that the speed at which the ball lands is the same as it was when it was thrown, but the direction of motion has been reversed. Understanding the directed nature of the quantities you are measuring is essential to using the formulas correctly.

To see how useful these equations are, ask one last question. If the ball leaves your hand 1.5 m from the ground and you do not catch it, how long will it be in the air? To phrase it slightly differently, if the ball is launched upward with a velocity of 5ms^{-1} , after how long will it have a displacement of -1.5 m ? Here is one approach to solving the problem:

$$\begin{aligned}s &= ut + \frac{1}{2}at^2 \\ -1.5 &= 5t - \frac{1}{2}t^2 \\ t^2 - 10t - 3 &= 0 \\ t &= \frac{10 \pm \sqrt{100 + 12}}{2}\end{aligned}$$

So t is either 1.2 s or -0.2 s . Ignoring the negative value, the answer is that the ball lands about 1.2 s after being thrown. (Ignoring the negative makes sense if you imagine projecting the motion backward in time as if the ball had been thrown up from the floor instead of your hand.)

The Motion of a Cannonball

What about when the ball is not thrown straight up? In this case, the velocity and acceleration of the ball are no longer collinear, and so you cannot simply make calculations with single numbers. However, you can use the trick mentioned in Chapter 5, separating the vectors into components. If you take your basis as the horizontal vector $(1 \ 0)^T$ and the vertical vector $(0 \ 1)^T$, then the acceleration due to gravity acts only along one of the basis vectors. This means that the component of motion in the horizontal direction is unaffected by gravity. Horizontal velocity is constant. Only the vertical component of motion experiences an acceleration.

Suppose this discussion is applied to cannonballs. A cannon is a device for launching a projectile at an exact angle and speed. The barrel of the cannon is a straight tube that forces the cannonball to travel along a particular path, and the velocity can be calibrated, more or less precisely, according to the amount of gunpowder and the weight of the ball. (This is an example of conservation of energy, which you will look at in a moment.)

As is illustrated in Figure 7.1, if the cannon is aimed at an angle of θ from the ground, and the ball emerges with a speed u , then the horizontal component of the ball's initial velocity is $u \cos(\theta)$, while the vertical component is $u \sin(\theta)$. What is more, if the length of the

cannon is l , then the height of the ball as it leaves the barrel is $l \sin(\theta)$. How far will the ball travel? Assuming that the ground is flat, the vertical distance traveled by the ball is $-l \sin(\theta)$, so ignoring both the height of the cannon and the horizontal component of the motion, you can use the same technique as before to calculate how long it will be in the air:

$$s = ut + \frac{1}{2}at^2$$

$$-l \sin(\theta) = u \sin(\theta)t - \frac{1}{2}gt^2$$

$$5t^2 - u \sin(\theta)t - l \sin(\theta) = 0$$

$$t = \frac{u \sin(\theta) + \sqrt{u^2 \sin^2(\theta) + 20l \sin(\theta)}}{10}$$

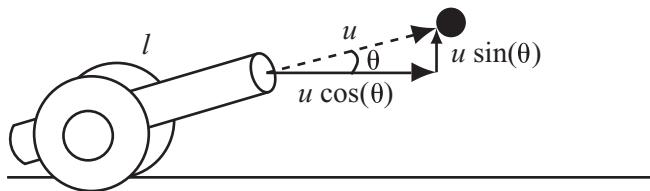


Figure 7.1

A cannon firing a projectile at a particular angle.

This is a complicated formula. To make it clearer, substitute in some numbers. Suppose $\theta = 30^\circ$, so $\sin(\theta) = 0.5$. Likewise, suppose the cannon is 2 m long and fires the cannon-ball at 20 ms^{-1} . The time of flight is

$$t = \frac{20 \times 0.5 + \sqrt{20^2 \times 0.5^2 + 20 \times 2 \times 0.5}}{10}$$

$$= 1 + \frac{\sqrt{120}}{10} = 2.1 \text{ s}$$

You can use this information along with the (constant) horizontal velocity to calculate the distance traveled:

$$s = ut = u \cos(\theta)t = 20 \times \frac{\sqrt{3}}{2} \times 2.1 = 36.3 \text{ m}$$

Using calculus, it is possible to calculate at which angle the ball will travel the farthest. (You might want to try your luck with this question. It isn't exactly easy, but you have all the tools you need.) You might want to ignore the height of the cannon for simplicity. Assume that the vertical displacement when hitting the ground is zero.) In Exercise 7.2, you are asked to provide a function that will aim a cannon so that its projectile hits a particular point.

These techniques are quite general. They work just as well to calculate the behavior of a person jumping up in a platform game or a slingshot firing pellets in a firing range. The only tricky part is calculating the initial speed of the projectile. This is the topic of the next section.

Mass and Momentum

Up to now, the weight of the particles being discussed has been irrelevant. After the projectile is in the air, its path of flight is the same no matter how heavy it is. However, if you want to throw it somewhere, you know from experience that it makes a big difference how heavy something is. In this section you look at the concept of *mass*, and why throwing a brick is so much harder than throwing a ping pong ball.

Mass and Inertia

There are two words used to describe how heavy something is. The most common in English is *weight*. In physics, this word is used for a force, specifically the downward force on an object due to gravity. It is this (vector) quantity that is measured by, for example, a weighing scale (or scales). The other word is *mass*, or, equivalently, *inertia*. You need the language of forces to explain the term precisely, but in this context, you can think of it as a measurement of "how hard it is to move X." Mass is a scalar quantity, measured in units such as grams, kilograms, or pounds. These units of mass are the units shown on a weight scale because, after all, as long you remain on the same planet, weight and mass are proportional.

A particle's mass is generally constant—although as is discussed in Chapter 15, this is not so when rockets are considered. More importantly, unlike weight, mass is the same wherever the particle is, whether on a neutron star or in outer space. While an object on the moon weighs approximately a sixth of what it weighs on Earth, its mass is unchanged, so in either place it will take the same amount of effort to get it to the same speed.

So what is mass, anyway? One answer is that it is a fundamental property of matter, going right down to the level of atoms and below. In the simplest terms, it is a measure of how much “stuff” there is in the object. This means that the mass of an object is distributed over its whole volume. In terms of calculating solutions to problems, this might be a problem in itself. Consider that up to now the discussion has been about abstract, point-sized objects, such as particles. How can you be sure that an object with its mass spread over a wider space will behave the same way?

Fortunately, it turns out that you can. To a very good approximation, and apart from rotation, at a particular point inside the object, an object moves along exactly the same path as a point particle. The point inside the object is known as its *center of mass* or *center of gravity*. The exact position of the center of mass depends on the shape of the object and the distribution of matter within it, but for a symmetrical object, such as a ball, the center of mass is at the center of the object. This topic is addressed at greater length in Chapter 13.

Calculating Momentum

Mass tends to be most useful in calculations when it is combined with other quantities. One particularly important value is *momentum*, which is the product of a particle’s mass and velocity. As the product of a vector and a scalar, momentum is a vector, parallel to the velocity.

You can think of momentum as a measure of “how hard it is to stop X.” It is much more difficult to stop a 10-ton truck traveling at one mile per hour than it is to stop a toy car traveling at 15 miles per hour. This is why in feats of strength people are able to pull a truck with their teeth. Although it requires a huge effort to get the thing moving, once it is under way, its own momentum helps to keep it going.

The most important thing to know about momentum is that it is *conserved*. This means that in the absence of any external influences on a particle—or for that matter on a whole system of particles—the total momentum will be unchanged.

When you are dealing with cannonballs, this is not a very useful thing to know. After all, a cannonball does have an outside influence—the force of gravity. This means that the momentum of a cannonball changes from moment to moment as it arcs through the air. But again, this only applies to the vertical momentum. Horizontally, perpendicular to gravity, velocity and momentum are unchanged. Still, momentum does not really come into its own until you start looking at collisions.

Energy

In the final section of this chapter, you examine the concept of *energy* and see how it gives you a different way of looking at ballistic motion.

Kinds of Energy

Energy is a measure of how much an object can change its environment. The more energy a particle has, the more effect it can have on the rest of the world. A cannonball lying on the ground can't do anything to anyone, but one flying through the air can knock down a stone wall. Energy is a scalar quantity, measured in a unit called the *joule* (J), which is equal to 1 kg ms^{-1} (one kilogram-meter per second). In addition to the joule, another unit is used to measure energy units. This is the *calorie*. As it stands, however, in the world today, use of this term is often restricted to problems concerning food (as in a calorie of nutrition).

Energy can take a number of different forms. Here are a few:

- **Kinetic energy (k.e.):** This form of energy is held by any moving object. If a particle of mass m has velocity \mathbf{v} , then its kinetic energy is equal to $\frac{1}{2}m|\mathbf{v}|^2$. Notice how closely related this is to the formula for momentum.
- **Gravitational potential energy (g.p.e.):** While this form of energy is held by all objects in the universe, how this happens is complicated. For starters, it is mostly a relative term. You can say that "this object has 5 J more g.p.e. than that one," but it doesn't make much sense to say "this object has a g.p.e. of 100 J." The g.p.e of an object is its potential to fall. Returning to the discussion of the cannonball, if the cannonball is sitting on top of a parapet rather than on the ground, it clearly has the potential to have a much greater influence on a person beneath the parapet. A particle of mass m at height h above another particle has a relative g.p.e of $mh|\mathbf{g}|$, where \mathbf{g} is the local acceleration due to gravity. To simplify the use of g.p.e in problems, it is generally easiest to define a particular height as 0, and measure g.p.e from that point.
- **Elastic potential energy:** Among other things, this form of energy is stored by a stretched rubber band or spring, which when released can fire itself or a projectile some distance. Elastic potential energy is further discussed in Chapter 16.
- **Heat energy:** This form of energy is a measure of how badly something can burn you. Generally it's more useful to think in terms of the amount of energy released by a hot object over any particular period of time. As *power*, it is measured in units of *watts* (W), which are joules per second.

- **Chemical potential energy:** This form of energy is stored in a *reactive* substance, such as gunpowder. It is a function of the stability of the substance as compared to the stability of the substance(s) it becomes after reacting. For example, iron reacts with air to produce iron oxide or rust. Because rust is more stable than pure iron, the process of rusting releases energy.
- **Electrical energy:** This form of energy is used by objects with an electric current flowing through them. An electric cell (battery) has a certain *voltage*, such as 10 V, which indicates the amount of power used (in watts) for each *ampere* of current passing across a circuit containing the battery.

There are many more types of energy, but all of them can be sorted into two categories. One is *kinetic*. Kinetic energy, such as heat and electricity, is actually doing something. It is able to do so with movement. Heat is a large-scale view of the kinetic energy of the atoms and molecules in a substance, and electrical energy is the result of the movement of electrons (or other charged particles) in a conductor.

The other form of energy is *potential* energy. Potential energy represents the ability of an object to act, given a change of circumstances. Potential energy is usually relative. For example, chemical potential energy depends on the particular chemical reaction involved. Gravitational potential energy depends on relative heights. Elastic potential energy is a little different, being entirely a property of a particular spring.

Conservation of Energy

Like momentum, energy is *conserved* within a system, apart from the action of any external force. In fact, in certain cases you can say energy is conserved even when external forces do act, when these forces are included in your energy calculations. Gravity is an obvious example. The force of gravity does not really count as an “external” force since you take it into account in the form of gravitational potential energy. (All potential energies are associated with some kind of force.)

Conservation of energy allows you to make calculations about different parts of a situation using different techniques. For example, when dealing with the cannonball, one of the problems you couldn’t solve was determining the initial speed of the ball as it leaves the cannon. Using conservation of energy and a number of known facts, you can theoretically calculate this speed.

For starters, a chemical reaction between the gunpowder and oxygen in the air causes the cannonball to be propelled outward. The reaction converts some of the chemical potential energy of the gunpowder into heat, giving kinetic energy to the molecules in the air behind the cannonball and causing them to expand rapidly. As illustrated by Figure 7.2, expansion transfers kinetic energy to the cannonball. This means that if you can calculate the energy given off when a quantity of gunpowder is ignited, you also might be able to calculate the energy of the cannonball.

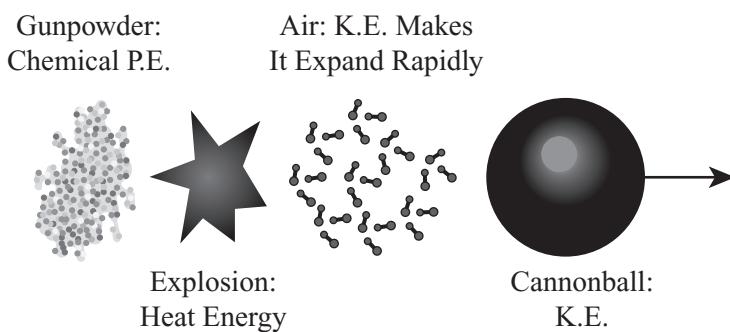


Figure 7.2

The energy transfer during the firing of a cannon.

Of course, in real life, energy transfer is not perfect. A great deal of the heat given off by the gunpowder is absorbed by the surrounding air. This means that the transfer of energy is not 100% efficient. However, you can experiment to discover the particular efficiency.

Another important issue is that, while the cannonball is energized by the reaction, in accordance with the law of conservation of momentum, the cannon itself is propelled *backward*. The explosion within the cannon counts as an “internal force.” If the total horizontal momentum of the cannon and cannonball is zero before the gun fired, it must be zero afterward. (This does not hold for the vertical momentum, because gravity is acting on the ball and the cannon.) As a result, if the cannonball has a forward momentum p after the explosion, the cannon must have a momentum of $-p$ to compensate.

Suppose the cannonball has a mass of 2 kg and the cannon has a mass of 200 kg. After the gun fires, if the cannonball has a horizontal speed of 30 ms^{-1} , its momentum is 60 kg ms^{-1} . Therefore, the cannon must have the same momentum in the opposite direction, which means that its speed must be $\frac{60}{100} = 0.3 \text{ ms}^{-1}$. This is the classic “recoil” phenomenon.

Notice that because the cannon is so much heavier than the ball, its speed is much less than the speed of the ball it is firing.

Another consideration is the kinetic energy involved in the explosion. For simplicity, assume that the cannon is aimed horizontally. The kinetic energy of the cannonball after firing is $0.5 \times 2 \times 30^2 = 900$ J. The kinetic energy of the cannon is $0.5 \times 200 \times 3^2 = 9$ J. Although the momentum in both directions is equal, the difference in speed makes a major difference to the energy. The energy of the cannon is significantly less than the ball. Nevertheless, it does have some energy. If the cannon is braced to prevent it from moving backward, the ball will emerge faster! All the energy that previously went into moving the cannon backward is now available to give extra kinetic energy to the ball, which means that the ball will have approximately 909 J instead of 900 J. Its speed will be equal to roughly $\sqrt{909} = 30.1$ ms⁻¹.

Note

Why does conservation of momentum no longer apply when the cannon is braced? Because the brace is exerting an external force on the cannon to hold it in place. This is an example of Newton's Third Law, which you'll see in Chapter 12.

Using Conservation of Energy to Solve Ballistics Problems

The law of conservation of energy gives you different ways to approach a ballistics problem. Ignoring air resistance, after a projectile is flying freely, its energy must remain constant. When it flies up, kinetic energy is transformed to gravitational potential energy (g.p.e.), and when it flies back down, g.p.e. is converted back into kinetic energy. You can use this fact to perform many of the same calculations performed previously. For example, you can ask how high the ball will climb.

As before, suppose the cannon fires the cannonball at 20 ms⁻¹. Suppose also that the cannonball has a mass of m . The ball's initial kinetic energy is therefore $400 m$. You'll measure g.p.e. from the muzzle of the cannon, so at the point at which the ball leaves the cannon—its g.p.e.—is 0. Therefore, its total energy (the sum of the kinetic energy and g.p.e.) at all times during the flight must be $400 m$.

Assume that the ball is fired straight up (not a wise move when you consider that the ball lands at the same speed it was fired!). At the highest point of the motion, the ball must have a kinetic energy of 0, so its g.p.e. will be $400 m$. From the formula for g.p.e., given that $mgh = 400 m$, $h = 40$ meters (m). The highest point reached by the ball is 40 m above the muzzle of the cannon. Notice that the mass term cancels out on the two sides of the equation, leaving a height independent of the mass of the ball. Since the trajectory of a projectile is independent of mass, this is to be expected.

Note

It can be a little confusing that the variable m represents mass, while the letter “m” refers to the units of meters. Remember that only letters in italics are variables.

If the cannon is fired at an angle, again you have to deal with the motion in horizontal and vertical components. Suppose as before that the cannon is at 30° to the horizontal, so its (constant) horizontal velocity is $20 \times \frac{\sqrt{3}}{2} = 17.3 \text{ ms}^{-1}$. At the top of the motion, vertical velocity is 0, but horizontal velocity is unchanged, so the kinetic energy is $\frac{1}{2} \times m \times 1.73^2 = 150m \text{ J}$. Therefore, since the total energy must still be $400m$, you have:

$$150m + mgh = 400m$$

$$gh = 250$$

$$h = 25\text{m}$$

Both of these values need to be adjusted to account for the difference of height above the ground. Try solving the same problem using the equations of motion and see if you get the same answer.

Conservation of energy or momentum considerations will not tell you anything about time. They will not tell you how long anything takes to happen. But in many situations they will enormously simplify calculations.

Exercises

In the following exercises assume units of meters, seconds, and kilograms, and that $g = 10$.

EXERCISE 7.1

Write a function named `javelin(throwAngle, throwSpeed, time)` that calculates the position and angle of a javelin over time.

The function should return an array of two values: a vector (array) representing the position of the javelin at time `time` after firing and the angle it makes with the horizontal. During its flight, a javelin is more or less oriented along the tangent to the curve. In other words, it is parallel to the velocity vector.

EXERCISE 7.2

Write a function named `aimCannon(cannonLength, muzzleSpeed, aimPoint)` that will return the correct firing angle for a cannon if it is to hit a particular point.

Your function should take two scalar arguments, the length of the cannon and the firing speed. It should also take one vector (array) argument, the target position relative to the base of the cannon. It should calculate the best angle at which to aim the cannon so as to hit the point. It should return this angle in either degrees or radians (your preference). In some cases, it may not be possible to hit the target. If this occurs, you should return an error message of some kind. In other cases, there may be more than one possible firing angle. If this occurs, you can choose any valid angle.

EXERCISE 7.3

Write a function named `fireCannon(massOfBall, massOfCannon, energy)` that returns the speed of the cannonball as it emerges from the barrel.

This function should use the law of conservation of momentum to calculate the speeds of both the cannonball and the cannon after it is fired. Assume that the variable `energy` represents the total kinetic energy after firing. In other words, chemical energy from the gunpowder is not lost to heat.

Summary

This chapter has gone through the basics of ballistics very quickly, but this is because the concepts are really fairly simple. In any event, they are simpler than the fundamental ideas of vectors and algebra! The toolset is small and constant, and most problems are just variations on the same theme.

In the next two chapters, you will make your first forays into the exciting world of collision detection. Then you will return to the laws of conservation of momentum and energy to learn how to find out what happens when objects collide.

You Should Now Know

- The meaning of the word *acceleration* as the rate of change of *velocity*
- How *gravity* acts on objects near sea level as constant downward acceleration
- How to use the equations of motion to calculate unknowns in ballistics problems
- The meaning of the words *mass* and *inertia*
- What *momentum* is and how and when it is *conserved*
- The different forms of energy and how they are transformed one into another
- How to use the *law of conservation of energy* to solve ballistics problems

CHAPTER 8

DETECTING COLLISIONS BETWEEN SIMPLE SHAPES



In This Chapter

- Overview
- Ground Rules
- When Circles Collide
- When Squares Collide
- When Ellipses Collide
- When Things Collide

Overview

Collision detection and resolution are the most fundamental and mathematical topics of game programming, and for this reason, in previous chapters, it has been necessary to review mathematical subjects to prepare for working with these topics. At this point, then, you are ready to take things to greater depth, applying the mathematics and also extending the discussion to include physics. The path leads to two-dimensional (2-D) and then three-dimensional (3-D) applications. In this chapter, you examine 2-D collision. Examining 2-D collision makes it possible, in Part III, to look at rotational physics. After that, in Part IV, you can extend the work into 3-D contexts.

In this chapter, you'll look at how to tell if and when two simple geometrical shapes will collide and what you can determine about the point of impact. As part of this effort, the pseudocode examples in this chapter are more specific than in others. As you work with these examples, remember that one of the primary goals is to focus on exploring mathematics, and as a result, the examples tend to be more verbose and less efficient than they would be if they were refined for use in industrial applications. As is commonly the case in contexts of study and learning, it is hoped that the verbosity of the code will make it easier to understand. A few notes are offered concerning how the examples might be optimized, but this is by no means a central objective of this or any other chapter.

In the chapters that follow, you'll look at more complex collision situations. However, since the topics easily become complex, to ensure that you have a good grounding of understanding, the first order of business is to consider at length two-dimensional collisions.

Ground Rules

To set up a few basic rules, first assume that you are making calculations based on one or more moving objects. Generally, for a given object, you know its current position and the vector along which it is moving during the current step of time. As mentioned in previous discussions, this is the *displacement* of the object. Consider, for example, an object moving with a velocity v pixels per second. If it has been 150 milliseconds since the last time you checked on the object, then the object has a displacement of $\frac{100}{50} \times v$ pixels.

Even for more complicated motion, where the object is accelerating, you can make the same calculation based on the current velocity. Assuming that you check collisions frequently, it is likely that velocity won't change a great deal in each step of time. Consequently, if you assume that the velocity remains constant during this time, you will not end up with a significant error in your calculations.

A second rule that is important throughout this chapter is the *principle of relativity*. This principle is related to Einstein's theorem of the same name. The principle of relativity stipulates that if you add a constant velocity or position to any physical system of particles, all other measurements will remain the same. When this occurs, the isolation of measurement is referred to as a *frame of reference*.

To examine frames of reference more closely, consider a situation in which two people, Sam and Ella, are sitting across from each other in a van and tossing a ball back and forth. As they toss the ball, whether the van is parked or driving at 100 mph, Sam and Ella use exactly the same amount of force to throw the ball, and the ball takes exactly the same trajectory through the air relative to them. This establishes their frame of reference.

Of course, the ball moves through a different trajectory relative to the ground, but the ground is not part of Sam and Ella's frame of reference. You can get a sense of how this is so by considering a stronger version of the principle of relativity. This version depicts the car windows as blacked out, so Sam and Ella cannot know by any experiment at all whether they are moving or standing still. The two terms "moving" and "standing still" become meaningless to them.

Note

Einstein's theory of relativity followed from discoveries about the nature of light. He contended that, according to the principle of relativity, time passes at different rates depending on how fast you are moving. Reformulating the equations of motion to remove anomalies concerning the behavior of light, he overturned most people's notions about space and time. As far as is possible with respect to a theory, countless observations and experiments have proven Einstein's contentions to be correct.

As a third ground rule, to make it easier to discuss colliding entities, a practice typical of object-oriented programming will be used in this and subsequent chapters. Accordingly, each colliding entity can be identified as an *object*. Generally, each object has characteristics, which are represented by *properties*. An object might be named *circle*. One of its properties might be *radius*. Another property might be *circumference*. To access a property, a dot operator (.) is used. For example, to assign a value of 20 to the *radius* property of the *circle* object, you write *circle.radius = 20*. Such syntax conventions eliminate the need to use functions with a multitude of arguments. Appendix B provides a brief explanation of object-oriented programming.

When Circles Collide

The topics in this section address the simplest possible form of collision detection. This form of collision detection involves objects that are circular. A circle makes collision detection easier to work with because, since it is completely symmetrical, it offers no point on its circumference that is in any way distinct from any other point. Not having to compensate for differing points simplifies collision calculations.

Circles

You've looked at circles in previous chapters. A circle is the set of points equidistant from a particular point. The particular point is the center, \mathbf{O} , of the circle. The distance from the circumference to the center is the radius of the circle. The vector from \mathbf{O} to any point \mathbf{P} on the circumference is given by $r(\sin\theta \cos\theta)^T$ for some angle θ . The tangent of a circle at \mathbf{P} is perpendicular to the radius $\overrightarrow{\mathbf{OP}}$. It is convenient to use the notation $C(\mathbf{o}, r)$ to indicate a circle of radius r centered on the point with position vector \mathbf{o} .

It is easy to tell if a point \mathbf{P} is inside a circle. All you do is calculate the distance OP and compare it to the radius. If the distance is less than the radius, the point is inside; if is not less than the radius, then it is outside. If it is equal to the radius, the point is on the circumference.

Note

You can speed up this process by calculating the squared distance OP^2 and comparing it to the square of the radius. This avoids the need to repeatedly take square roots using the Pythagorean Theorem.

One more useful fact is that if two circles with centers \mathbf{O} and \mathbf{O}' are touching at a point \mathbf{P} , the points \mathbf{O} , \mathbf{P} , \mathbf{O}' are collinear. This follows directly from the fact that the tangent at \mathbf{P} is perpendicular to both $\overrightarrow{\mathbf{OP}}$ and $\overrightarrow{\mathbf{O}'\mathbf{P}}$. Remembering this relationship helps you to solve collision problems geometrically.

A Moving Circle and a Wall

As illustrated in Figure 8.1a, to deal with a moving circle and a wall, you can start with a circle $C(\mathbf{o}, r)$ moving at some velocity \mathbf{v} toward a straight line. Consider, for example, the game of Pong.

Given the time step illustrated by Figure 8.1a, you know the position \mathbf{O} of the center of C at time 0. You also know a point \mathbf{A} with position vector \mathbf{a} on the line. Additionally, you know a vector \mathbf{w} along the line. (For now, assume that the wall is infinitely long.) Given particular values for \mathbf{v} , \mathbf{w} , \mathbf{o} , \mathbf{a} , and \mathbf{r} , two questions then arise. Will the circle hit the wall during this time-step? If so, when?

Assuming it hits the wall, in Figure 8.1b, you can see the circle C at the point of impact. The circle C is touching the wall at a point \mathbf{P} , and the center is now at \mathbf{O}' . You know that $\overrightarrow{\mathbf{O}'\mathbf{P}}$ is perpendicular to \mathbf{w} since the wall must be a tangent to C at \mathbf{P} . What's more, since it is the vector along which C has moved, you know that $\overrightarrow{\mathbf{OO}'}$ is some multiple of \mathbf{v} .

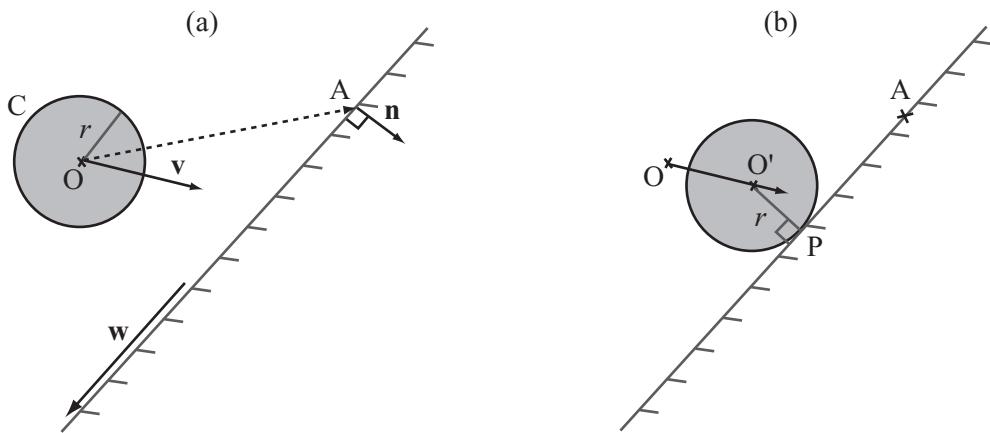


Figure 8.1

A circle moving toward a wall (a) at its start position and (b) touching the wall.

So what can you deduce from this information? Do you have all that you require to define a function for collision detection? To answer these questions, start with \overrightarrow{OP} , which can be called, tentatively, vector \mathbf{r} . You know that \mathbf{r} is perpendicular to \mathbf{w} and that its length is r . All you need to derive is the direction in which it points. You can work this out using the vector \overrightarrow{OA} . To do so, take either of the two unit normals to \mathbf{w} and call it \mathbf{n} . If the angle between \mathbf{n} and \overrightarrow{OA} is greater than 90° , then \mathbf{n} is basically pointing toward O , and $\mathbf{r} = r\mathbf{n}$. If the angle is less than 90° , then \mathbf{n} is pointing away from the circle, and $\mathbf{r} = -r\mathbf{n}$. To distinguish between these possibilities, take the dot product of \mathbf{n} and \overrightarrow{OA} . This identifies the component of \overrightarrow{OQ} in the direction \mathbf{n} . If the absolute value of this component is less than r , then the circle becomes embedded in the wall and your function should return an error.

After you have calculated \mathbf{n} , you can also make another quick check. If the dot product of \mathbf{v} and \mathbf{n} is positive, the circle is already moving away from the wall. In this situation, no collision occurs. Otherwise, you now have a simple vector equation. You know that P is on AB and that it is a vector \mathbf{r} away from a point on the trajectory of O . For the scalars s and t , this gives you the equation

$$\mathbf{o} + t\mathbf{v} + \mathbf{r} = \mathbf{a} + s\mathbf{w}$$

Having derived this equation, you can create the `circleWallCollision()` function, which detects the collision of the circle and the wall. Here is the pseudocode for the function. Again, note that object-oriented syntax is used to indicate the properties of the circle and wall objects:

```
function circleWallCollision(cir, wal)
    // calculate the normal to the wall
    set n to wal.normal
    set a to wal.startPoint-cir.pos
    set c to dotProduct(a, n)
    if abs(c)<cir.radius then return "embedded"
    if c<0 then set r to n*radius
    otherwise set r to -n*radius

    // check if the circle is approaching the wall
    set v to dotProduct(displacement, n)
    if v>0 then return "none"

    // calculate the vector equation
    set p to cir.pos+r
    set t to intersectionTime(cir.pos,
        cir.displacement,
        wal.startPoint,
        wal.vector) // see Chapter 5
    if t>1 then return "none"
    return t
end function
```

The code in the `circleWallCollision()` function makes use of functions discussed in Chapter 5. (Review Chapter 5 for extended details.) However, with reference to the call to the `intersectionTime()` function, the arguments you supply are p_1 , v_1 , p_2 , v_2 . The function returns the proportion t of v_1 from p_1 that must be traveled before you meet the line from p_2 along the vector v_2 . As discussed in Chapter 5, you can draw some helpful conclusions about the intersection point from this value. For example, if t lies on $[0,1]$, an intersection occurs within the time period you are interested in. Given this frame of reference, if the circle is approaching the wall, t will always be greater than or equal to 0. This concept arises often throughout this and following chapters.

A Stationary Circle and a Moving Point

Another case concerning collisions with circles involves small objects, such as a point particle moving through space. In this scenario, your objective is to determine if the point collides with a circle in the same space occupied by the circle.

As you have already seen, determining if a point is inside a circle is trivial. To deal with the question at hand, you need to determine when, if ever, the particle's trajectory enters the radius of the circle. In Figure 8.2, you can see an example of this. In this case, in addition to the position P and vector \mathbf{v} of the point, you know the position O and radius r of the circle.

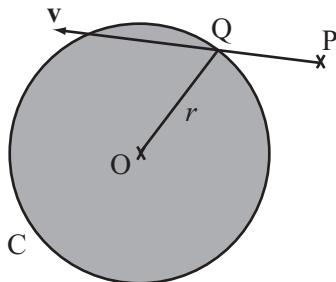


Figure 8.2

A stationary circle and a moving point.

To solve this problem of where the point lies with relation to the circle, look at the point Q in Figure 8.2. This is where the particle enters the circle. The point Q lies on the circle C , but it is also on the trajectory of the particle. Its position vector is given by $\mathbf{p} + t\mathbf{v}$ for some t , and also the magnitude of the vector from this point to O is r . With this information, you can derive the following vector equation:

$$(\mathbf{p} + t\mathbf{v} - \mathbf{o}) \cdot (\mathbf{p} + t\mathbf{v} - \mathbf{o}) = r^2$$

Because you know all of these values except t , this equation is fairly easy to solve. You multiply out the brackets and see what you get. (Note that the name \mathbf{w} is given to the vector $\mathbf{p} - \mathbf{o}$.) Working through the multiplication, you end up with the following formulation:

$$t = \frac{-\mathbf{w} \cdot \mathbf{v} \pm \sqrt{(\mathbf{w} \cdot \mathbf{v})^2 - (\mathbf{w} \cdot \mathbf{w} - r^2)(\mathbf{v} \cdot \mathbf{v})}}{|\mathbf{v} \cdot \mathbf{v}|}$$

While this is a fairly involved formula, it remains that it has resulted from simple calculations, and you can deduce that if t is less than 0, the point is moving away from C. Likewise, if t is greater than 1, the point will not hit C in this time-step. Finally, if t is imaginary—if the value under the square root is less than zero—then the trajectory doesn’t meet the circle at all. To make use of the calculation, you can develop the `pointCircleCollision()` function.

```
function pointCircleCollision(pt, cir)
  set w to pt.pos-cir.pos
  set ww to dotProduct(w,w)
  if ww<cir.radius*cir.radius then return "inside"
  set v to pt.displacement-cir.displacement
  set a to dotProduct(v,v)
  set b to dotProduct(w,v)
  set c to ww-cir.radius * cir.radius
  set root to b*b - a*c
  if root<0 then return "none"
  set t to (-b-sqrt(root))/a
  if t>1 or t<0 then return "none"
  return t
end function
```

If you use the principle of relativity, you will see that the problems involving a circle and a line and a circle and a point on or within a circle are in a sense “duals” of one another. With respect to the `circleWallCollision()` function, instead of thinking of the circle C as approaching the wall with velocity \mathbf{v} , think of C as stationary and the wall approaching it with velocity $-\mathbf{v}$. The wall will first hit the point Q on C, and this point is where the tangent at Q is parallel to the wall. If you draw a line from Q along the velocity vector, you find the point P. This problem then looks a lot like the problem addressed by the `pointCircleCollision()` function, except that instead of knowing the point P and trying to find Q, you know Q and are trying to find P.

Two Moving Circles on a Straight Line

Imagine a long track made of two parallel rails, with two balls sitting in it some distance apart. If you roll one ball (ball 1) along the track toward the other (ball 2), when will they collide? How about if both balls are moving? Figure 8.3a illustrates this situation. Both balls have radius r , and they begin with their centers d units apart. They have velocity \mathbf{v} and $k\mathbf{v}$, respectively. Likewise, since the balls are moving along the same straight lines, their velocities are parallel.

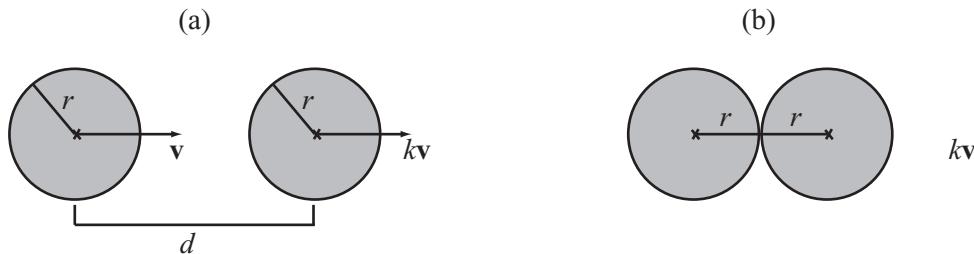


Figure 8.3

Two moving circles on a straight line (a) before collision and (b) during collision.

Consider the second question first. This question concerns what happens if both balls are in motion. You can answer this question first since, applying the principle of relativity, it makes no difference whether one ball or both are moving. If you subtract the velocity of ball 1 from both balls, then ball 1 is stationary and ball 2 is moving with velocity $(k - 1)\mathbf{v}$.

For the first question, look at Figure 8.3b. At the moment of contact, the centers of the circles are $2r$ units apart. This means that all you need to do is use standard equations of motion to discover at what time t , the circles are this distance apart. Here is one approach:

$$|\mathbf{v}|t = d - 2r$$

$$t = \frac{d - 2r}{|\mathbf{v}|}$$

As long as their centers are along the line of collision, this approach extends cases in which the circles are of different sizes. With this perspective, you can develop the `circleCircleStraightCollision()` function:

```
function circleCircleStraightCollision(cir1, cir2)
    set relspeed to cir1.speed-cir2.speed
    set d to cir1.pos-cir2.pos // linear position
    set r to cir1.radius+cir2.radius
    if d<r then return "embedded"
    set t to (d-r)/relspeed
    if t>1 or t<0 then return "none"
    return t
end function
```

The `circleCircleStraightCollision()` function does not specify a velocity and position vector. Instead it makes use of a speed and linear position. This approach is used because this function addresses situations in which you know beforehand that the circles are colliding head on. If this is the case, you know the velocity must lie on the line joining the two centers. Still, in order to determine the point of contact, you must calculate the velocity.

Two Circles Moving at an Angle

The previous examples have dealt with objects meeting head on, but in most cases, the objects that collide meet at an angle to each other. To deal with circles, a more general solution is needed, one that copes with circles of variable sizes colliding at variable angles. Fortunately, while this might seem to present difficulties, the fact is that you have already completed much of the work required to solve it.

As shown in Figure 8.4, the circle $D(\mathbf{a}, p)$ is approaching the circle $C(\mathbf{0}, r)$ with a relative velocity \mathbf{v} . A larger circle of radius $p + r$ has been drawn around O . At the moment of collision between the two circles, the line from A in the direction \mathbf{v} enters the larger circle.

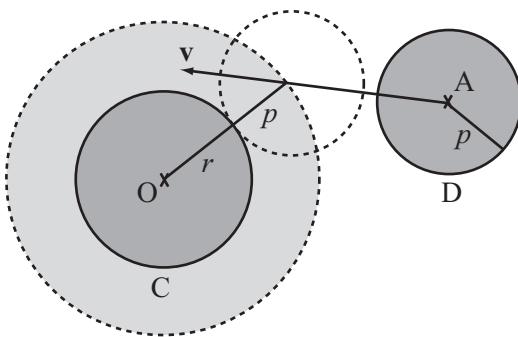


Figure 8.4

Two moving circles at an angle.

Given this information, it becomes possible to rewrite the `pointCircleCollision()` function so that it accommodates circles moving at an angle. What results is the `circleCircleCollision()` function:

```

function circleCircleCollision(cir1, cir2)
  set w to cir1.pos-cir2.pos
  set r to cir1.radius+cir2.radius
  set ww to dotProduct(w,w)
  if ww<r*r then return "embedded"
  set v to cir1.displacement-cir2.displacement
  set a to dotProduct(v,v)
  set b to dotProduct(w,v)
  set c to ww-r*r
  set root to b*b-a*c
  if root<0 then return "none"
  set t to (-b-sqrt(root))/a
  if t>1 or t<0 then return "none"
  return t
end function

```

One Circle Inside Another

In the preceding sections, you have considered circular objects colliding on the outside edge; however, as illustrated by Figure 8.5, you can also consider situations in which one circle bounces around inside a larger circle.

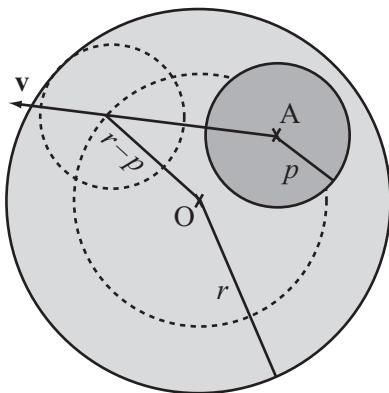


Figure 8.5
One circle inside another.

When one circle bounces around inside another, the situation is much like an external collision. However, in this case, at the moment of collision, the centers of the circles are separated by a distance, A , that is equal to the difference of the radii. Were the collision external, then they would be separated by the sum of the radii.

To create a function that addresses one circle bouncing around inside another, you can create a function that is almost identical to the `circleCircleCollision()` function. Since you are subtracting the radii rather than adding them, the `circleCircleInnerCollision()` function is no longer symmetrical. It tests whether circle 1 is inside circle 2, not the reverse. Also, notice that you now need to look for the larger root. As long as one circle is inside the other, the velocity vector always collides once in front of the circle and once behind.

```
function circleCircleInnerCollision(cir1, cir2)
  set w to cir1.pos-cir2.pos
  set r to cir2.radius-cir1.radius
  set ww to dotProduct(w,w)
  if ww>r*r then
    set rr to cir2.radius+cir2.radius
    if ww<rr*rr then return "embedded"
    return "outside"
  end if
  set v to cir1.displacement-cir2.displacement
  set a to dotProduct(v,v)
  set b to dotProduct(w,v)
  set c to ww-r*r
  set root to b*b-a*c
  set t to (-b+sqrt(root))/a
  if t>1 then return "none"
  return t
end function
```

The Point of Contact

Much can be said about the point at which a circle meets another object. First, you can quickly calculate the position of the point directly from the results of the functions discussed in the previous sections. Each of the functions returns a variable t that represents the fraction of the period of time that elapses before the objects collide. If you know the velocities of the objects (as you always do in these cases), then you can calculate `shape.pos+shape.displacement * t` to get the new position.

Second, you can determine the actual point of contact, although this will vary according to the situation. As shown in Figure 8.1, when a circle hits a wall, the point (P) of contact is along the perpendicular from the center of the circle to the wall. On the other hand, as shown in Figure 8.4, when two circles collide, the point of contact lies along the line joining the centers. This is \overrightarrow{OQ} in Figure 8.2. Whatever the position, the tangent of the circle at that point is perpendicular to the radius. This fact becomes important later on.

When Squares Collide

Having looked at a smooth shape, you can examine what happens with shapes that have corners. The most common such shapes are squares and rectangles. Like circles, these shapes have a number of useful properties and symmetries that make them reasonably easy to work with. Rectilinear collision detection is at the basis of many complex collision-detection routines.

Rectangles and Squares

Although rectangles and squares have been examined in previous chapters, it is helpful at this point to review a few details. A rectangle is a two-dimensional shape with straight sides. Given this definition, it is often called a *polygon*. A rectangle has four vertices, making it a *quadrilateral*, and each of the four vertices is a right angle. Since both pairs of opposite sides of a rectangle are equal and parallel, it a *parallelogram*. For a rectangle ABCD, where the vertices are labeled clockwise, $\overline{AB} = \overline{DC}$ and $\overline{BC} = \overline{AD}$, and \overline{AB} is perpendicular to \overline{BC} . A square is a special type of rectangle where the lengths AB and BC are equal and thus all the sides are equal.

The *diagonals* \overline{AC} and \overline{BD} meet at the center of the rectangle ABCD. In a square, these diagonals are perpendicular. The diagonal \overline{AC} is equal to $\overline{AB} + \overline{BC}$, and the diagonal \overline{BD} is equal to $\overline{AD} - \overline{AB}$. This is true for any parallelogram, which is why the rule for addition of vectors is often called the *parallelogram rule*.

Note

To extend the classification of quadrilaterals, a *rhombus* is a parallelogram with all sides equal (a diamond shape). A *trapezium* is a quadrilateral with two sides parallel. An *isosceles trapezium* is a trapezium with its non-parallel sides equal in length. A *kite* is a quadrilateral with two pairs of equal adjacent sides. A square is a kind of rectangle and a kind of rhombus, both of which are parallelograms, and a rhombus is also a kite. A parallelogram is a trapezium. All of these are quadrilaterals. All are also convex quadrilaterals.

You employ basic notation to define a rectangle. For example, you say that the rectangle $R(\mathbf{u}, \mathbf{v}, \mathbf{w})$ is the rectangle centered on the point with position vector \mathbf{u} whose sides are given by the perpendicular vectors $2\mathbf{v}$ and $2\mathbf{w}$, where $|\mathbf{v}| > |\mathbf{w}|$. Although there are simpler approaches (see the previous note), you use this approach to defining a rectangle because it is simple and more symmetrical. As a side benefit, if you allow \mathbf{w} to take any value, not just a vector perpendicular to \mathbf{v} , you get a generic parallelogram, which means that many observations about rectangles generalize to arbitrary parallelograms.

Strictly speaking, the description just presented of a rectangle includes more information than you need. Since you can use the fact that two sides are perpendicular to determine the direction of the second side, you can just as well give only the vector for one side and the length (any positive scalar) for the other. With this approach, there is no potential for error. Every possible combination of three values yields a valid rectangle. Only in a degenerate case, where one side is zero, do you arrive at a straight line segment rather than a rectangle. As already mentioned, however, the first approach is simpler and more symmetrical.

With reference to the rectangle $R(\mathbf{u}, \mathbf{v}, \mathbf{w})$, you can determine the vertices of R relatively effortlessly. They are $\mathbf{u} + (\mathbf{v} + \mathbf{w})$, $\mathbf{u} + (\mathbf{v} - \mathbf{w})$, $\mathbf{u} - (\mathbf{v} + \mathbf{w})$, $\mathbf{u} - (\mathbf{v} - \mathbf{w})$. Note that this ordering of the vertices is to be used in this chapter and those that follow. You also say that a rectangle is *oriented* in the direction of its long side, so R is oriented along the vector \mathbf{v} .

As illustrated by Figure 8.6, to develop a function that tests if a point P is inside the rectangle R , you find the components of the vector \overrightarrow{UP} in the directions \mathbf{v} and \mathbf{w} . If the component in the direction \mathbf{v} is shorter than $|\mathbf{v}|$, and the component in the direction \mathbf{w} is shorter than $|\mathbf{w}|$, then P is inside ABCD.

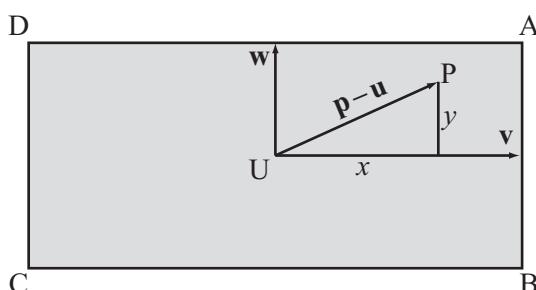


Figure 8.6

Determining if a point is inside a rectangle.

The `pointInsideRectangle()` function applies this algorithm:

```
function pointInsideRectangle(pt, rectCenter, side1, side2)
    set vect to pt-rectCenter
    set c1 to abs(component(vect, side1))
    set c2 to abs(component(vect, side2))
    if c1 > magnitude(side1) then return false
    if c2 > magnitude(side2) then return false
    return true
end function
```

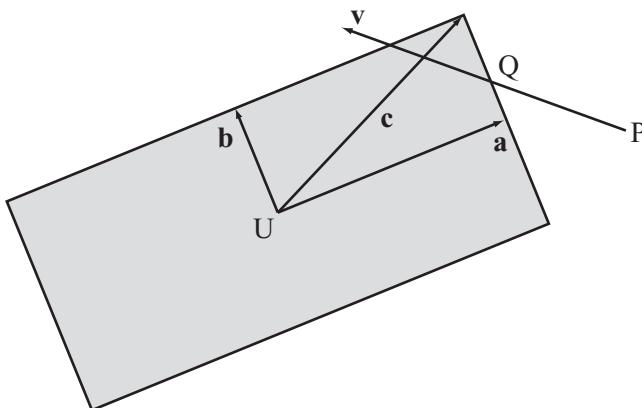
In the next chapter, you'll look at another, more general method for testing if a point is inside a polygon. For now, notice that the `pointInsideRectangle()` function counts a point on the perimeter as "inside" the rectangle. If you don't want to allow this, replace the `>` sign with `>=` signs.

To define a further function, you can also say that for any point P on the perimeter of R , if you take the vector \overrightarrow{UP} , then either its component in the direction \mathbf{v} has magnitude $|\mathbf{v}|$ or its component in the direction \mathbf{w} has magnitude $|\mathbf{w}|$. If both of these are true, then P is a vertex of R . Conversely, a point P is on the perimeter of the rectangle if this is true and also the point is "inside" R as defined by the definition previously given in this section of a rectangle. The `pointOnRectangle()` function applies these observations for any parallelogram:

```
function pointOnRectangle(pt, rectCenter, side1, side2)
    set vect to pt-rectCenter
    set c1 to abs(component(vect, side1))
    set c2 to abs(component(vect, side2))
    set s1 to magnitude(side1)
    set s2 to magnitude(side2)
    if c1 > s1 then return false
    if c2 > s2 then return false
    if c1=s1 or c2=s2 then return true
    // NB: for a safer test, use e.g. abs(c1-s1)<0.001
    return false
end function
```

A Stationary Rectangle and a Moving Point

As an extension of the previous section, consider the problem of detecting the collision between a stationary rectangle and a moving point. To solve this problem, you begin with the simplest case of a particle at P with velocity \mathbf{v} passing through a plane that contains a rectangle $R(\mathbf{u}, \mathbf{a}, \mathbf{b})$, as is illustrated in Figure 8.7.

**Figure 8.7**

A stationary rectangle and a moving point.

What you're looking for is a point Q such that Q is on the trajectory of the particle, and Q is on the border of R. To solve this, you test for intersections with each of the four sides. At the first such intersection, the particle collides. The `pointRectangleIntersection()` function shows how such a function is implemented:

```
function pointRectangleIntersection(pt, rec)
  set c to rec.side1+rec.side2
  set t to 2 // start with a high value of t
  // then repeat over the four sides and look for the first collision
  repeat for v = rec.side1, rec.side2
    repeat for m = 1,-1
      set t1 to intersectionV(pt.pos,
                               pt.displacement,
                               rect.pos-m * c,
                               m * v * rec.axis*2)
      if t1 = "none" then next repeat
      set t to min(t,t1[1])
    end repeat
  end repeat
  if t=2 then return "none"
  return t
end
```

The method used in the `pointRectangleIntersection()` function to loop through the four sides involves creating a vector. You create a vector $\mathbf{c} = \mathbf{a} + \mathbf{b}$, which is the vector from the center of the R to one vertex. The implication of this construction is that $-\mathbf{c}$ is the vector to the opposite vertex. From the first of these vertices, two sides point in the direction $-\mathbf{a}$ and $-\mathbf{b}$, and from the second, the other two sides point in the direction \mathbf{a} and \mathbf{b} .

The `pointRectangleIntersection()` function calls the `intersectionV()` function, which is a variation of the `intersection()` function used before. The `intersectionV()` function takes position-displacement information instead of four end points. It then returns a value of t for the intersection time if the two line segments intersect.

Because there is no smooth mathematical function describing all the points on a rectangle, many approaches to detecting collisions involving rectangles must be developed. Further, you often end up checking several possible options. This makes it necessary to be careful about potential problems at the vertices. With the `pointRectangleIntersection()` function example, a question arises concerning what happens if P lies somewhere along the line extended from one of the rectangle sides and moves parallel to that side. Does the function catch such a movement? Exercise 8.1 asks you to consider this type of problem.

Two Rectangles at the Same Angle

The next problem beyond detection of collisions between particles and rectangles concerns collision between two rectangles. To start with the simplest case, as illustrated by Figure 8.8, a and b, consider a situation in which the rectangles are aligned along the same axis.

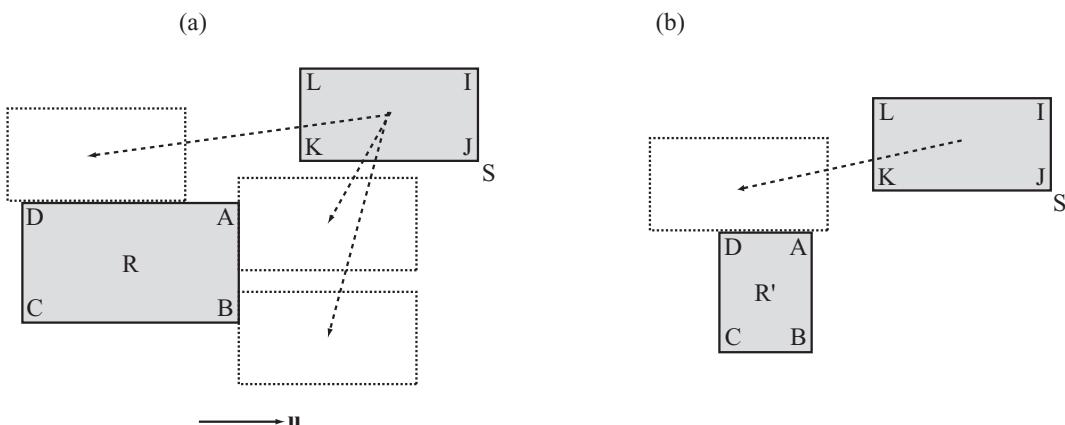


Figure 8.8

Two rectangles at the same angle (a) similar shapes and (b) differing shapes.

Figure 8.8 shows two rectangles, R and S, both of which are aligned along the axis \mathbf{u} . In Figure 8.8a, depending on its velocity vector, rectangle S can collide with rectangle R in a number of different ways. However, they all boil down to six possibilities. In this example, at the point of collision, at least one of the three vertices J, K, L of S must be touching a side of R. The side that is touched is always either the side AB or the side AD of R.

Further, while K can be touching either AB or AD, J must be on AD (or both AB and AD) and L must be on AB (or both AB or AD). The other three possibilities arise from cases such as the one Figure 8.8b illustrates. Here the rectangle R' is smaller along the colliding side than S, so at the point of collision none of the vertices of S is colliding with R'. However, the vertices of R' are colliding with S, which means that you can perform a similar set of calculations the other way around. In the other direction, you need only check collisions for vertices B and D.

Since it is an adaptation of the `pointRectangleCollision()` function, which tests for intersection with all four sides, the `rectangleRectangleCollisionStraight()` function doesn't take advantage of most of the optimizations just mentioned. You might want to see if you can come up with a faster implementation of the same test. One approach is to change the `pointRectangleCollision()` function as well. Still, here is one approach to dealing with collisions involving two rectangles:

```
function rectangleRectangleCollisionStraight(rec1, rec2)
    set t1 to rrVertexCollisionStraight(rec1, rec2)
    set t2 to rrVertexCollisionStraight(rec2, rec1)
    if t1="none" then return t2
    if t2="none" then return t1
    return min(t1,t2)
end function
```

Supporting the `rectangleRectangleCollisionStraight()` you make two calls to the `rrVertexCollisionStraight()` function, which as discussed previously, attends to detecting collisions:

```

// now test each of these for intersection with the second rectangle
set s1 to rec2.side1*xvector
set s2 to rec2.side2*yvector
set t to 2 // you're trying to find a value less than 1 for t
repeat for each pt in points
    set t2 to pointRectangleIntersection(pt, rec)
    if t2="none" then next repeat
    set t to min(t, t2)
end repeat
if t=2 then return "none"
return t
end function

```

As mentioned in the previous discussion, you check for position and displacement using the `pointsToCheck()` function:

```

function pointsToCheck(r1, r2, displacement)
    set points to an empty array
    set c1 to component(displacement, r1)
    set c2 to component(displacement, r2)
    if c1>0 then
        add r1+r2 to points
        add r1-r2 to points
    otherwise
        add -r1+r2 to points
        add -r1-r2 to points
    end if
    if c2>0 then
        if c1>0 then add -r1+r2 to points
        otherwise add r1+r2 to points
    otherwise
        if c1>0 then add -r1-r2 to points
        otherwise add r1-r2 to points
    end if
end function

```

Although developing three functions for detecting rectangular collisions at the same angle is somewhat complicated, it is not as bad as it looks. The complications arise only from having to check so many different cases. As points and rectangles, the same calculation can be performed with aligned parallelograms. With these, however, as differing from the approach used in the `rrVertexCollisionStraight()` function, you must specify the value of the `yvector` variable rather than calculating it from the normal given by the `xvector` variable.

A much simpler version of this process can be used if the rectangles are axis-aligned. Axis-aligned rectangles are aligned in the direction of the two basis vectors. This case is significantly easier to deal with, and you'll return to it in a couple of chapters.

Two Rectangles at Different Angles

When rectangles are not aligned along the same axis, while you might think the problem gets a lot harder, it's actually much the same. In fact, conceptually it is even easier than dealing with rectangles at the same angles. This is so because the point of contact will always be one of the eight vertices. As shown in Figure 8.9, you can narrow down the number of vertices to check to six, three from each rectangle.

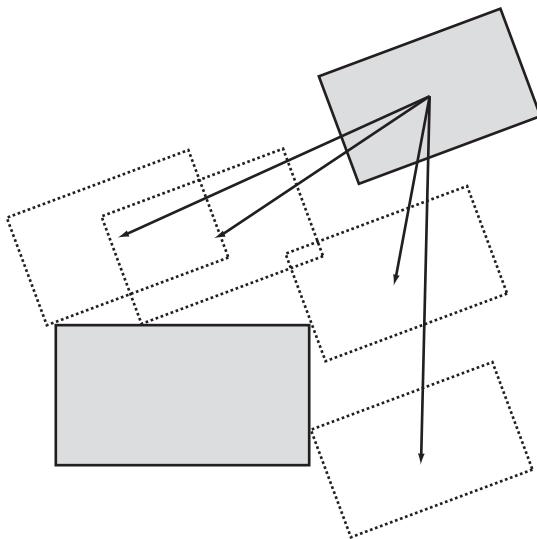


Figure 8.9

Two rectangles at different angles.

As is shown in the `rectangleRectangleAngledCollision()` function, to work out which vertices to check, you can compare the displacement vector with the diagonals of the rectangles. Since they are all different, you give the values of the sides of the rectangle as vectors. This approach allows you to take advantage of work you have already completed:

```

function rectangleRectangleAngledCollision (rec1, rec2)
  set t1 to rrVertexCollisionAngled(rec1, rec2)
  set t2 to rrVertexCollisionAngled(rec2, rec1)
  if t1="none" then return t2
  if t2="none" then return t1
  return min(t1,t2)
end function

```

The `rrVertexCollisionAngled()` is called twice in the `rectangleRectangleAngledCollision()` function, and its responsibilities involve first arriving at the points to test and then testing for the points of interaction:

```

function rrVertexCollisionAngled(rec1, rec2)
  // calculate the points to test
  set axis to rec1.axis
  set points to pointsToCheck(rec1.side1* axis,
                               rec1.side2*normalVector(axis),
                               displacement)
  // now test each of these for intersection with the second rectangle
  set t to 2
  repeat for each pt in points
    set t2 to pointRectangleIntersection(pt, rec2)
    if t2="none" then next repeat
    set t to min(t, t2)
  end repeat
  return t
end function

```

As the code in this and the previous section illustrates, no significant differences arise between calculations of aligned and non-aligned rectangles. One of the only points of concern is with axis-aligned boxes, which is the topic of a subsequent section.

The Point of Contact

A major difference between working with smooth objects like circles and polygons, as with rectangles, is in the point of contact. When two smooth objects collide, there is always a precise normal at the point of contact, and this defines the collision precisely. With polygons, this is no longer the case. Either you have a collision between two edges of the polygon, as in Figure 8.8, or between an edge and a vertex, as in Figure 8.9.

Note

It is theoretically possible to get a collision between two vertices as well, but this is so unlikely that for now it will not be explained. As becomes evident in other discussions, you can use what is called a *perturbation* to deal with it.

If a collision occurs between two edges, detecting the collision involves recognizing that the normal is simply the normal of the meeting edge. However, a vertex doesn't have a normal. When a vertex and edge of a polygon meet, you might expect almost anything to happen. Actually, it's not that bad. When the vertex meets an edge, there is always at least one well-defined normal.

Otherwise, things are reasonably straightforward. It is simple to adapt the given functions to return the point of contact of the rectangles and other useful information.

Likewise, detecting collisions between a rectangle and a wall is almost identical to detecting collisions between two edges. A wall is a rectangle that is much bigger than the rectangle colliding with it. Since you do not need to test for collisions between the wall vertices and the rectangle vertices, you cut the problem in half. You can also perform collisions with a line segment. A line segment is essentially a rectangle with one pair of sides having zero length.

When Ellipses Collide

An ellipse is a flattened oval, and an oval is an object that is more or less the shape of an egg. Ellipses are almost as simple as circles, but because they are not completely symmetrical, complications arise when dealing with them. Where a circle has an infinite number of axes of symmetry, an ellipse has only two. The result of this difference is that you must approach detecting collisions involving ellipses with methods that differ from those used with circles.

Ellipses

An ellipse is to a circle what a rectangle is to a square. It is a circle stretched in one direction. One way to imagine drawing an ellipse is to take a length of string and attach it to two drawing pins on a piece of paper at points A and B. As shown in Figure 8.10, putting the tip of the pencil to the paper and pulling the pencil around the inside of this string results in an ellipse. This shape is distinguished by the fact that for any point P on the perimeter, the sum of the distances AP and BP is constant. The length of the string does not change.

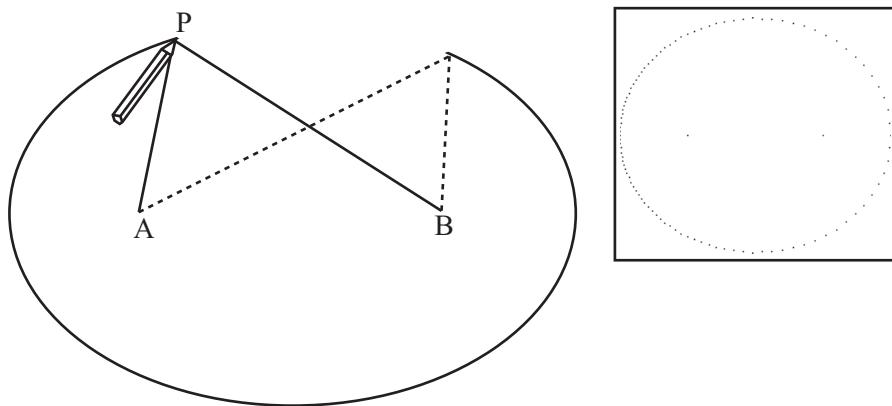


Figure 8.10

Drawing an ellipse. (Inset: a computer-generated ellipse.)

Note

The *perimeter* of a shape is another word for its edge. The perimeter of a circle, for example, is its circumference. Note, likewise, that *foci* is the plural of *focus*.

The two points A and B are called the foci of the ellipse, and the length $AP + BP$ is the internal diameter. Notice that if A and B are the same point, then the ellipse becomes a circle, and the internal diameter is equal to the diameter of the circle. “Internal diameter” is not a formal term used with ellipses, but it conveys the idea that an ellipse differs from a circle by the addition of these two lines.

Developing a function to draw an ellipse can take any number of paths. The `drawEllipseByFoci()` function represents one approach. It generates an ellipse given the foci and internal diameter. Likewise, note that a function discussed in Chapter 5, the `angleBetween()` function, is used in its implementation.

```

function drawEllipseByFoci(focus1, focus2, diameter)
  set resolution to 100
  // increase this number to draw a more detailed ellipse
  set angle to 2*pi/resolution
  set angleOfAxis to angleBetween(focus2-focus1, array(1,0))
  if angleOfAxis="error" then set angleOfAxis to 0
  set d to magnitude(focus1-focus2)
  set tp to diameter*diameter-d*d
  repeat for i=1 to resolution
    set a to angle*i // the angle made at focus1 with the major axis
    set k to tp/(2*(diameter-d*cos(a)))
    set ha to a+angleOfAxis
    set p to k*array(cos(ha),sin(ha))
    draw point p
  end repeat
end function

```

The `drawEllipseByFoci()` function does not provide a purely mathematical approach to drawing an ellipse, but you might want to see if you can use a mathematical approach. To accomplish this, the key is to use the cosine rule on the triangle PAB. The drawback to a purely mathematical approach, however, is that it draws more points around one focus than the other. On the right of Figure 8.10 is an ellipse drawn by using a mathematical approach. The dots on the left half of the perimeter are more dense than those on the right.

Describing an Ellipse Using Coordinates

Rather than the approach presented in the previous section, you can describe an ellipse using coordinate values. To understand how this is so, remember that every point on a circle centered on $(0,0)$ with radius r has coordinates $(r \cos\theta, r \sin\theta)$ for some θ . A similar description holds true of an ellipse. The shape traced by the points with coordinates $(a \cos\theta, b \sin\theta)$ is an ellipse with internal diameter $\frac{b^2}{2a}$ and foci at $\left(\frac{b^2-a^2}{2a}, 0\right)$ and $\left(\frac{a^2-b^2}{2a}, 0\right)$. Assuming that $a > b$, the lengths a and b are called the *semi-major* and *semi-minor* axes, respectively.

However, this approach draws only an ellipse aligned along the x -axis. In this case, you say that the ellipse is aligned along the vector of its major axis. The major axis is the vector between the two foci. If you want this ellipse to be rotated so that its axes are not aligned with the major axis, drawing the ellipse becomes complicated. You must adjust

each point by rotating it by a constant angle around the center of the ellipse. With the assistance of a function that adjusts points, the `drawEllipsesByAxes()` function plots an ellipse using coordinate values:

```
function drawEllipseByAxes(center, a, b, alpha)
  set resolution to 100 // increase to draw more accurately
  set ang to 2*pi/resolution
  repeat for i=1 to resolution
    set angle to ang*i
    set p to rotateVector(array(a*cos(angle), b*sin(angle), alpha))
    draw point center+p
  end repeat
end function
```

The `rotateVector()` function takes care of the work of rotating the ellipse. In addition to making the code easier to understand, refactoring the code into two functions makes it possible to use the `rotateVector()` function in other contexts discussed later on.

```
function rotateVector(v, alpha)
  set x to v[1]
  set y to v[2]
  set l to sqrt(x*x + y*y)
  set x1 to l*cos(alpha-atan(y,x))
  set y1 to l*sin(alpha-atan(y,x))
  return array(x1,y1)
end function
```

Translation

Although the two approaches to plotting an ellipse, shown in the previous two sections, are largely equivalent, in different circumstances, each has certain advantages. Still, the most useful is the second, for it appeals to the idea of an ellipse as a stretched circle. To state this precisely, any ellipse centered on the origin can be created from a unit circle. To do so, first scale the space by a factor of a in one direction and a factor of b in the other. Then rotate the ellipse to point in the correct direction. You can describe this with a standard transformation matrix \mathbf{T} , made up of a scale followed by a rotation. After you've done this, you can translate the ellipse to the correct position. You use the notation $E(\mathbf{c}, \mathbf{T})$ to describe such an ellipse. The `drawEllipseFromMatrix()` function warps the previously discussed `drawEllipseByAxes()` function to accomplish this task:

```

function drawEllipseFromMatrix(pos, mat)
    set v to mat.column
    set n to mat.column
    drawEllipseByAxes(magnitude(v), magnitude(n), atan(v[1],v[2]))
end function

```

One common term used with ellipses is the value $\sqrt{1 - \frac{b^2}{a^2}}$, known as the *eccentricity* of the ellipse. The higher the eccentricity, the more “pointy” an ellipse becomes. An eccentricity of 1 gives a circle, while an eccentricity of infinity gives a straight line.

A final useful fact about an ellipse is that the tangent to the curve at P makes the same angle with each of the lines AP and BP. If you have an elliptical pool table with a ball at one focus and a pocket at the other, hitting the ball in any direction results in the ball going in the pocket. On an ellipse aligned along the x -axis, at the point $(a \cos\theta, b \sin\theta)$, the tangent lies along the vector $(-a \sin\theta, b \cos\theta)^T$.

A Stationary Ellipse and a Moving Point

Suppose that the particle at P is moving with velocity v in space occupied by the ellipse $E(c, T)$. A question that arises is whether the point enters the ellipse. To answer this question, consider first translating to the ellipse’s frame of reference. You accomplish this by subtracting c . Having done this, you end up with a vector equation:

$$Tu = p - c + tv$$

If you invert the matrix T using the methods discussed in Chapter 5, you end up with this equation:

$$u = T^{-1}(p - c + tv) = T^{-1}(p - c) + tT^{-1}(v)$$

Since u is a unit vector, you know that its dot product with itself must be 1. As a result, you reach this equation:

$$(T^{-1}(p - c) + tT^{-1}(v)) \cdot (T^{-1}(p - c) + tT^{-1}(v)) = 1$$

This version of the equation might appear familiar. You are looking at the intersection of a particle with a unit circle, a problem dealt with earlier in this chapter.

As an approach to developing a function to detect collisions of ellipses and particles, it is helpful to think about it geometrically. Consider Figure 8.11. In (a) you have the generalized ellipse. In (b) you have rotated the ellipse to a new frame of reference, with the axes

aligned with the basis. In a final step, not shown in the figure, you can compress the space down to turn the ellipse into a circle. Although everything is moving about the displacement vector accordingly, the relative distances along the lines don't change even if the absolute length does. Here you can really see the power of the relativity principle.

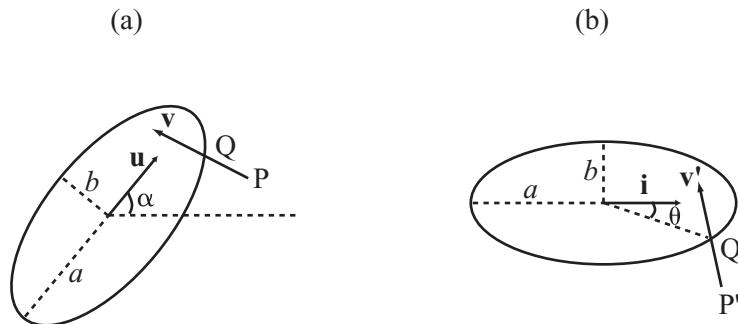


Figure 8.11

A stationary ellipse and a moving point (a) in the plane and (b) rotated to the ellipse's frame of refer-

The `particleEllipseCollision()` function implements an algorithm based on the previous discussion. Its arguments are the ellipse and the point, and as with other functions examined in this chapter, it draws from functions introduced in Chapter 5.

```
function particleEllipseCollision(pt, ell)
  set t to ell.transformationMatrix
  set inv to inverseMatrix(t)
  set p to pt.pos-ell.pos
  set w to matrixMultiply(inv,p)
  set ww to dotProduct(w,w)
  if ww<1 then return "inside"
  set v to matrixMultiply(inv,pt.displacement-ell.displacement)
  set a to dotProduct(v,v)
  set b to dotProduct(w,v)
  set c to ww-1
  set root to b*b-a*c
  if root<0 then return "none"
  set t to (-b-sqrt(root))/a
  if t>1 or t<0 then return "none"
  return t
end function
```

Two Ellipses

If two ellipses that happen to be aligned in the same direction meet head on, their collision is an easy analogue of the circular case. Such luxuries are sadly not to be had in the more general case where two ellipses $E(\mathbf{p}, \mathbf{S})$ and $F(\mathbf{q}, \mathbf{T})$ meet with arbitrary velocities. As in previous examples, you can apply the relativity principle to bring the problem down to a simpler case, a stationary unit circle meeting the ellipse $F'(\mathbf{q} - \mathbf{p}, \mathbf{S}^{-1} \mathbf{T})$. Unfortunately, this time the principle does not help you as much as it might. Ultimately, you end up with a highly complicated pair of nonlinear simultaneous equations that are not easily untangled.

There is one additional trick that can help you. You can use the fact that at the point of collision, the normals on the two surfaces are parallel. This is the key to the whole issue, but you don't yet have the tools and vocabulary to deal with it. For this reason, you'll return to the question in Chapter 19.

The Point of Contact

To determine the point of contact of an ellipse with its partner in collision, you can use the same approach you used with circles, but a bit more caution is needed. From the value of t in each of the functions dealt with previously, it is trivial to calculate the collision position of the moving bodies in your simulation. But to determine exactly where on the shapes they collide, you must transform back into the correct reference frame. This requires you to find the point of contact in the transformed frame, and then transform this point again with \mathbf{T} .

When Things Collide

As a final area of discussion, it is useful to consider mixed collisions between different kinds of shapes. Chief among these are circles and rectangles.

Collisions Between Circles and Rectangles

It's all very well to have these nice neat calculations for objects of the same kind, but what happens if you want to deal with more natural situations? Fortunately, it turns out that calculations with similar shapes can be easily adapted to such mixed events. In Figure 8.12 you can see a circle $C(\mathbf{p}, r)$ heading toward the rectangle $R(\mathbf{u}, \mathbf{a}, \mathbf{b})$ with velocity \mathbf{v} . Notice how, as \mathbf{v} varies, the situation falls into three possible forms:

- C hits a vertex of R .
- C hits an edge of R .
- There is no collision.

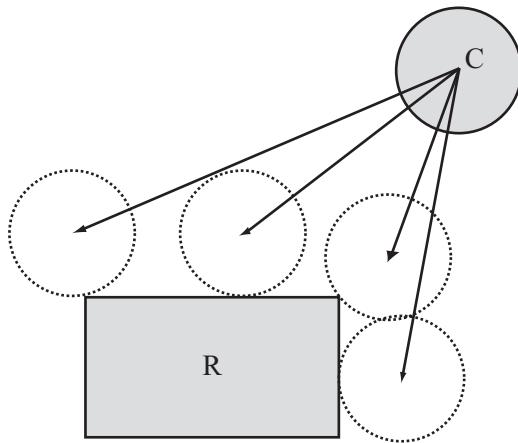


Figure 8.12

A collision between a circle and a rectangle.

In the first case, you can calculate the point of contact by considering the vertices of R and checking for intersection with C using the `pointCircleCollision()` function. In the second, you can think of the edge of R as a line segment or a wall, and use the `circleWallCollision()` function to handle it. Alternatively, you can expand the rectangle by the circle's radius in all directions and then apply the `pointRectangleCollision()` function. In both cases, you can work out in advance which of the vertices or edges are potentially involved in the collision just as you did with rectangle-rectangle collision. In the third case, no function developed so far helps or for that matter is needed.

The Point of Contact

Collisions between circles and rectangles are much the same as collisions between rectangles and rectangles in terms of the point of contact. Whether the collision is with an edge or a vertex, the normal is always the normal to the circle at that point. However, when the collisions involves an edge, it's easier to calculate the normal to the edge.

Exercises

EXERCISE 8.1

Write a function named `pointParallelogramCollision()`. As parameters, use `pt`, `displacement`, `parrPos`, `side1`, `side2`. As a model, use the `pointRectangleCollision()` function discussed in this chapter. Your function should return a value between 0 and 1 or the string "no intersection".

To write this function, take advantage of the fact that a parallelogram can be transformed into a rectangle by applying a skew transformation to the plane. You can do this within a complete function or call the `pointRectangleCollision()` function.

EXERCISE 8.2

Write functions named `rectangleRectangleInnerCollision()`, `circleRectangleInnerCollision()` and `rectangleCircleInnerCollision()` with appropriate parameters in each case to test for collisions inside shapes.

As with the case of a circle colliding with a circle, these inner collisions are very similar to the external ones covered in this chapter, but they each have subtle complications of their own. Use the code in the chapter to help you where possible.

Summary

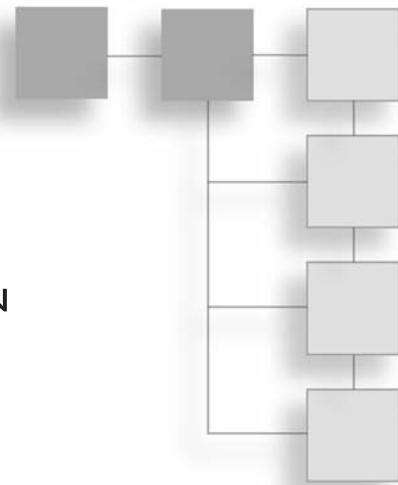
If you have been waiting for real programming tips, this chapter has offered them. Your exploration of collisions is only just beginning, however. In the next chapter, you will look at what happens after two objects have collided. After that, you will proceed to more general methods for collision detection that involve more complex shapes.

You Should Now Know

- How to describe circles, rectangles, ellipses, and lines on the plane in a form useful to collision detection
- How to detect collisions between different combinations of these shapes in various forms, including, for each of them:
 - How to detect collision with an infinitely small particle
 - How to detect collision with an infinite wall
 - How to detect collisions with objects of the same kind
- That ellipses are more difficult to work with and that more work must be performed before you learn how to calculate collisions between them in general cases

CHAPTER 9

COLLISION RESOLUTION



In This Chapter

- Overview
- Resolving a Single Collision
- Multiple Collisions

Overview

Before you look at the complex collisions that can occur between irregular shapes, it is helpful to explore problems involving generalized collisions that do not require extensive discussions of physics or mathematics. In this chapter, one such problem is how to determine what happens *after* two objects have collided. This is usually referred to as *resolving* a collision.

You have already done most of the hard work relating to resolving collisions in the previous chapter. While more complex collisions involve rotational physics (covered in Chapter 13), simple collisions of this type do not require that you account for the shape of the colliding objects. Instead, you concentrate on the mass and velocity of the colliding objects and the normal at the point at which they collide. In anticipation of this, in the last chapter, attention was given to the question of how to find the point of contact and the normal or tangent at the point of contact. This chapter puts this knowledge to work.

Resolving a Single Collision

The fundamental principles in resolving collisions are conservation of energy and momentum. The easiest collisions to deal with are called *elastic* collisions. An elastic collision is an abstraction. It does not exist in the real world. It describes a theoretical situation. In such a situation, a collision occurs in which all kinetic energy before the collision is conserved as kinetic energy after the collision.

Preservation of all the kinetic energy after the collision is not possible in reality because all objects release energy when they collide. If a collision occurs in the atmosphere, for example, as the molecules in the objects that collide start to move more rapidly, the energy is released in the form of heat, which in turn energizes the surrounding air. Often, part of this energy is perceptible as the sound of the collision. Even in space, where there is no air and no one to hear the crash, collisions between astronomical bodies are not totally elastic. Some energy is released.

Despite its limitations in dealing with the precise results of real-world events, due to its simplicity, elasticity is a useful starting point for understanding how collisions are resolved. Given this starting point, the stage is set for considering real-world events.

A Ball Hitting a Wall

Figure 9.1 illustrates a situation in which a ball traveling with velocity v hits a stationary, fixed wall with normal n . Since it is especially simple, this is a good example to use to explore elasticity. It is simple because you can deal with it without having to account for specific details of the ball or the wall.

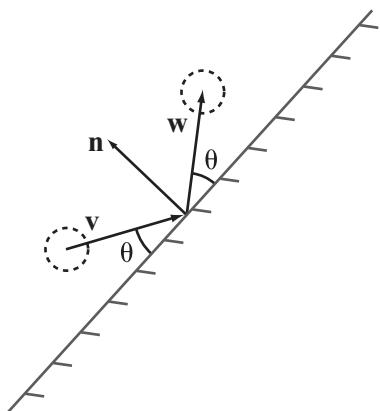


Figure 9.1

A ball hitting a wall.

The simplest way to deal with the event depicted by Figure 9.1 is to split the vector \mathbf{v} into two components, one in the direction \mathbf{n} (the normal component) and the other in the direction of the wall (the tangential component). Because no forces act on the ball in the tangential direction, the velocity of the ball in that direction remains unchanged. In the other direction, the velocity is reversed. When looked at in this way, the energy of the ball, the only moving object in the collision, remains unchanged. The end result of this is that you can find the new velocity by simply subtracting twice the normal component. To accomplish this, consider the `resolveFixedCollision()` function:

```
function resolveFixedCollision(obj, n)
    set c to componentVector(obj.velocity, n)
    set obj.velocity to v-2*c
end function
```

The `resolveFixedCollision()` function is both simple and applies to any collision in which one of the objects is fixed in place. It makes use of `componentVector()` function, explained in Chapter 5. Using this function, you obtain the normal at the point of contact. With this information, you can show that the colliding object will always follow the same path. With respect to the specific workings of the function, you can try the algebra yourself. If \mathbf{c} is the component of \mathbf{v} in one direction, then the component in the perpendicular direction is $\mathbf{v} - \mathbf{c}$.

Note

A mathematical purist might observe that the `resolveFixedCollision()` function does not return a value. To answer this objection, observe that `obj` argument of the object assumes that an object has been passed to the function. This object is assumed to have a `velocity` property. The function implicitly returns a value by altering the value assigned to `velocity`.

As is shown in Figure 9.1, a useful geometrical side effect of using the approach given by the `resolveFixedCollision()` function is that the angle between the original velocity vector and the wall (the angle of incidence) is equal to the angle between the final velocity vector and the wall (the angle of reflection).

As mentioned at the opening of this chapter in the discussion of elastic collisions, the calculations performed by the `resolveFixedCollision()` function center on the question of the ball's motion before and after the collision. It does not address what is going on *during* the collision. Dealing with this problem requires a much deeper level of detail, which is dealt with in Chapter 12.

A Ball Hitting a Movable Ball

Things become more complicated when the objects are not fixed in place. If both objects can move, then you need to determine the velocity of both objects after they collide. If you consider this problem at length, one of the first things that become apparent is that you need more information.

A ball bearing hitting a cannonball is going to make a lot less difference than a wrecking ball hitting the same cannonball at the same velocity. To determine the correct solution, you need to know the masses of the two objects. More importantly, you need to know the energy and momentum of the two objects. When neither of the two objects is fixed, you have a situation in which both of the laws relating to conservation of energy apply. First, the total momentum of the two objects after a collision should be the same as the total momentum before. Second, the total energy should be equal before and after the collision.

To explore how this is so, it is easiest to begin by considering a case in which the velocity of one object before the collision is zero. As you'll see in a moment, due to the principle of relativity, you can transform any other case to this one by subtracting one velocity from the other. Suppose you have a ball B (the *incident ball*) with velocity \mathbf{u} and mass m hitting a stationary, movable ball C (the *object ball*) with mass p , at a normal of \mathbf{n} . After the collision, as Figure 9.2 illustrates, B has velocity \mathbf{v} and C has velocity \mathbf{w} , both of which are unknown.

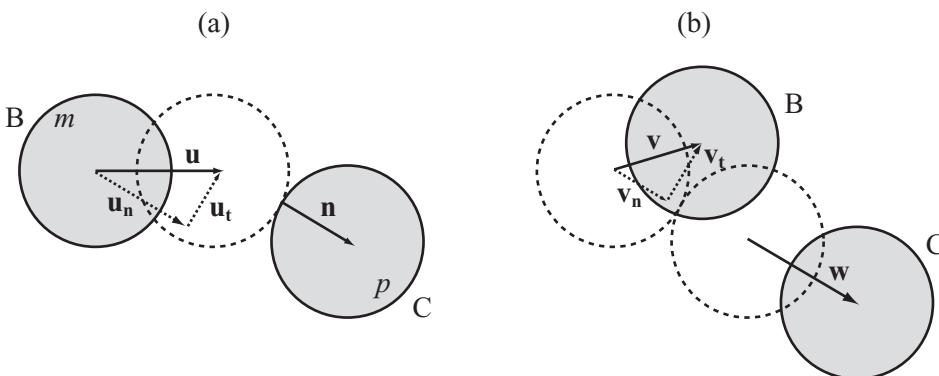


Figure 9.2

A collision with a stationary, movable object.

As before, you can split \mathbf{u} into two components with magnitude u_n (normal) and u_t (tangential). The same can be done for \mathbf{v} and \mathbf{w} . Likewise, since the forces do not act tangentially, the velocity of each ball in the tangential direction remains unchanged. As a result, $v_t = u_t$ and $w_t = 0$. In the other direction, conservation of momentum tells you that

$$mu_n = mv_n + pw_n$$

To make calculations easier to perform later on, divide through by p to get a ratio $r = \frac{m}{p}$. Using this approach, you arrive at $ru_n = rv_n + w_n$.

Meanwhile, conservation of energy tells you that $\frac{1}{2}mu^2 = \frac{1}{2}mv^2 + \frac{1}{2}pw^2$. This equation splits up to become

$$r(u_n^2 + u_t^2) = r(v_n^2 + v_t^2) + (w_n^2 + w_t^2)$$

Using your knowledge of the tangential direction, this becomes

$$r(u_n^2 + u_t^2) = r(v_n^2 + v_t^2) + w_n^2$$

You now have simultaneous equations in the unknowns v_n and w_n , which can be solved by substitution. From the momentum equation you have

$$w_n = r(u_n - v_n)$$

Substituting this value into the energy equation, you get

$$\begin{aligned} r(u_n^2 + u_t^2) &= r(v_n^2 + u_t^2) + r^2(u_n - v_n)^2 \\ u_n^2 + u_t^2 &= v_n^2 + u_t^2 + ru_n^2 - 2ru_nv_n + rv_n^2 \\ (r + 1)v_n^2 + (r - 1)u_n^2 - 2ru_nv_n &= 0 \end{aligned}$$

Given this step, you end up with a quadratic equation in v_n , which factorizes as follows:

$$(v_n - u_n)((r + 1)v_n + (r - 1)u_n) = 0$$

This gives you two roots of the equation: $v_n = u_n$ and $v_n = \frac{r-1}{r+1}u_n$. The first of these roots is expected. It is the initial situation. The other root must represent the situation after the collision. Notice that it depends on the ratio of the masses of the balls, not on their absolute values, which makes sense physically. Also notice that the value of u_t drops out of the calculation. This, again, is to be expected. You want the two components to be completely independent.

Note

Since elastic collisions are time-reversible, it is to be expected that the initial situation provides a valid solution to the equations. In other words, if you reverse all velocities after the collision and then run the simulation again, you get back to your initial position.

Given the progress so far, you can now quickly determine w_n by substituting back into the momentum equation:

$$\begin{aligned} w_n &= r \left(u_n - \frac{r-1}{r+1} u_n \right) \\ &= \frac{2ru_n}{r+1} \end{aligned}$$

Having come this far, you are in a position to develop a function that addresses moving objects. This is the `resolveCollisionFree1()` function. As with the previous function, it makes use of the `componentVector()` function from Chapter 5. In this case, two objects are passed to the function, with the velocity of each altered.

```
function resolveCollisionFree1(obj1, obj2, n)
    set r to obj1.mass/obj2.mass
    set un to componentVector(obj1.velocity,n)
    set ut to obj1.velocity-un
    set vn to un*(r-1)/(r+1)
    set wn to un*2*r/(r+1)
    set obj1.velocity to ut+vn
    set obj2.velocity to wn
end function
```

Two Moving Balls Colliding

Previous examples restricted you to using the velocity of one object. Adding a velocity to the second object is straightforward. To accomplish this, subtract the velocity from all objects prior to performing calculations. In effect, this makes it so that you are working with the same algorithm as before. After you subtract the velocity, remember to add it back on afterward. The `resolveCollisionFree()` function provides an example of this approach:

```

function resolveCollisionFree(obj1, obj2, n)
    set r to obj1.mass/obj2.mass
    set u to obj1.velocity-obj2.velocity
    set un to componentVector(u,n)
    set ut to u-un
    set vn to un*(r-1)/(r+1)
    set wn to un*2*r/(r+1)
    set obj1.velocity to ut+vn+u2
    set obj2.velocity to wn+u2
end function

```

With the `resolveCollisionFree()` function, except when calculating the collision normal, the shapes and sizes of the objects do not matter. This one function can handle every possible elastic collision.

There are further applications. Consider, for example, that if $r = 1$, the two masses are equal. If this is so, then you have $v_n = 0$ and $w_n = u_n$. This is the “Newton’s Cradle” effect, where the velocity of one ball is transferred entirely to the other. The result is that the original ball is left stationary (normal to the collision). The `resolveCollisionEqualMass()` function addresses this event:

```

function resolveCollisionEqualMass (obj1, obj2, n)
    set u to obj1.velocity-obj2.velocity
    set un to componentVector(u,n)
    set ut to u-un
    set obj1.velocity to ut+obj2.velocity
    set obj2.velocity to un+obj2.velocity
end function

```

With the `resolveCollisionEqualMass()` function, r tends to 0, then $v_n \rightarrow u_n$ and $w_n \rightarrow 0$. In other words, the mass p tends to infinity relative to m . This is the phenomenon encountered previously with a fixed wall. When the mass of the object ball is infinite, it cannot move, so this becomes a fixed-wall collision. Conversely, when an object is fixed, it can be considered to have an infinite mass.

Inelastic Collisions

As you might expect given the previous discussion, with an inelastic collision, not all the initial kinetic energy existing prior to the collision is transferred to kinetic energy after the collision. A certain proportion, for example, is lost to heat or sound.

The simplest way to simulate energy loss is to establish a fixed proportion of energy that will be lost when two objects collide. This proportion is called the *efficiency*. Efficiency can apply to one or the other object, or to both. If the objects in your simulation are similar, you can use a global efficiency. If the objects are not similar, you are best off giving one efficiency value to each object and combining them for each collision.

For example, consider an event in which ball B, with energy, transfers 95% of its energy at each collision. Another object, ball C, transfers 90% of its energy at each collision. In a collision between the two, where B has energy E_b and C has energy E_c , the total energy after the collision is $0.95 \times 0.9 \times (E_b + E_c)$.

Like elastic collisions, this approach to inelastic collisions represents a simplification. Real collisions do not work so linearly. Generally, faster collisions are more efficient than slower ones, and efficiency is affected by factors like the ambient temperature of the surrounding air. Still, the approach remains useful, for the errors are small, especially when working within normal ranges of speed and mass.

With an inelastic collision, as mentioned previously, the equations get a bit more complicated. Fortunately, you still know that motion tangential to the collision must be unaffected. In the normal direction, conservation of momentum remains as it was before. In fact, barring external forces, this law is always true. On the other hand, conservation of energy must be revised to take efficiency into account, and you end up with this equation:

$$\frac{1}{2} emu^2 = \frac{1}{2} mv^2 + \frac{1}{2} pw^2$$

Here, the value e is the product of the efficiencies of the two objects, expressed as a fraction. You can now go through exactly the same process as before. First, putting to use your knowledge that tangential motion is unaffected, you obtain

$$er(u_n^2 + u_t^2) = r(v_n^2 + u_t^2) + w_n^2$$

(As before, note that $r = \frac{m}{p}$).

Substituting for w , you arrive at

$$(r + 1) v_n^2 + 2ru_nv_n + (r - e) u_n^2 + (1 - e) u_t^2$$

Notice that this time, unless $e = 1$, the value of u_t does not drop out of the equation. If $e = 1$, perfect efficiency, then the equation reduces to the elastic example. The tangential speed does have an effect on the end result, because the more energy the system has, the more it will lose.

This equation again has two roots, which can be calculated from the quadratic formula:

$$v_n = \frac{-ru_n \pm \sqrt{r^2u_n^2 - (r+1)((r-e)u_n^2 + (1-e)u_t^2)}}{r+1}$$

The value u_n is no longer a root of the equation (unless $e = 1$). This is so because inelastic collisions are not time-reversible. However, by generalizing from the elastic case, you know that the particular root you are looking for is the result of taking the negative square root and substituting back for w . The result is that you can arrive at the `resolveInelasticCollisionFree()` function:

```
function resolveInelasticCollisionFree(obj1, obj2, n)
    set r to obj1.mass/obj2.mass
    set u to obj1.velocity-obj2.velocity
    set e to obj1.efficiency*obj2.efficiency
    set un to component (u,n)
    set ut to mag(u-un*n)
    set sq to r*r*un*un-(r+1)*((r-e)*un*un+(1-e)*ut*ut)
    set vn to n*(sqrt(sq)-r*un)/(r+1)
    set wn to r*(n*un-vn)
    set obj1.velocity to ut+vn+ obj2.velocity
    set obj2.velocity to wn+ obj2.velocity
end function
```

When $e = 1$, the `resolveInelasticCollisionFree()` function describes an elastic collision. Likewise, you can see that in this case $sq = un * un$, which leads to the same final values as before.

Just for completeness, the `resolveInelasticCollisionFixed()` function addresses fixed inelastic collisions. To reach this result, you start with the previous function and set $r = 0$. If you compare the `resolveInelasticCollisionFixed()` function to the elastic versions, you can see that they are similar.

```
function resolveInelasticCollisionFixed (obj1, obj2, n)
    set e to obj1.efficiency*obj2.efficiency
    set un to component (obj1.velocity,n)
    set ut to mag(obj1.velocity -un*n)
    set sq to (e*un*un+(e-1)*ut*ut))
    set vn to n*sqrt(sq)
    set obj1.velocity to ut+vn
end function
```

As before, the `resolveInelasticCollisionFixed()` function is independent of the mass of either object, which makes it much simpler to implement.

Multiple Collisions

Strictly speaking, this section should be part of collision detection rather than resolution, but it is related to both. How do you deal with the situation where more than one object is moving around on your screen? What happens if more than one object is colliding at the same time?

Collision Is a Recursive Function

The key to complex collision detection is to understand the process involved. With respect to a computer simulation, you can break down the process into six steps:

1. You have a set of objects O_1, O_2, \dots, O_n , all of which are moving with some velocity, have some mass and efficiency, and possess other characteristics.
2. Your simulation is divided into time-steps. Usually, these are variable and depend on the speed of the computer. You calculate the size of each time-step by comparison to the last known time-step. At this stage you have a time-step of s time units.
3. In each time-step, you need to search through the objects and find the first pair that collide. The collision detection function returns a value t between 0 and 1 and the collision normal.
4. The time taken to reach this collision is ts . As a result, you can move all the objects by a distance ts times their current velocity.
5. Resolve the collision. Each of the colliding objects now has a new velocity.
6. In this time-step, since there are still $(1 - t)s$ time units remaining, you must return to step 3 and repeat your calculations until there are no more collisions.

The `checkCollision()` function implements this procedure. In the scope of this book, this is a long function and might be best presented as several functions. The longer approach is used to make the steps explicit. The function also includes calls to functions that depend on your computing environment. This is discussed in detail later on. Also, notice that you split the list of objects into fixed and movable, which allows you to streamline the algorithm a little.

```
function checkCollision(time, movableObjects, fixedObjects)
    // check for the earliest collision
    set mn to 2
    set ob1 to 0
    set ob2 to 0
    repeat for i=the number of movableObjects down to 1
        set obj1 to movableObjects[i]
        // find the displacement vector and other
        // relevant facts about this object
        set l to parameters(obj1, t)
        // search for collisions with other movableObjects
        // (don't bother with those already checked)
        repeat for j=i-1 down to 1
            set obj2 to movableObjects[j]
            set l2 to parameters(obj2, t)
            set c to detectCollision(l,l2)
            if c is not a collision array then next repeat
            set tm to c[1] // the time of collision
            set m to min(mn,tm)
            if m<mn then
                set mn to m
                set n to c[2] // the normal vector
                set ob1 to obj1
                set ob2 to obj2
                set lf1 to l
                set lf2 to l2
            end if
        end repeat
        // now search for collisions with fixed objects
        repeat for j=the number of fixedObjects down to 1
            set obj2 to fixedObjects[j]
            set c to detectCollision(l, l2)
            if c is not a collision array then next repeat
            set tm to c[1] // the time of collision
            set m to min(mn,tm)
            if m<mn then
                set mn to m
                set n to c[2] // the normal vector
                set ob1 to obj1
                set ob2 to obj2
                set lf1 to l
                set lf2 to l2
```

```
    end if
    end repeat
end repeat
if mn=2 then set tmove to 1
otherwise set tmove to mn*t
repeat for each obj in movableObjects
    moveObject(obj, tmove)
end repeat
// if there is no collision you are finished
if mn=2 then return
// otherwise, you can resolve the collision here
set res to resolveCollision(lf1, lf2)
setNewVelocity(ob1, res[1])
setNewVelocity(ob2, res[2])
// and now recurse for the rest of the time-step
checkCollision(t*(1-mn), movableObjects, fixedObjects)
end function
```

Details in the `checkCollision()` function have been left unfinished, because they depend on the programming environment you are working in. In an object-oriented world, each moving shape is likely to be represented by an object, probably with some kind of subclassing corresponding to different shapes. In a procedural language, you usually use an array of values for each shape. Ultimately, however, the process is the same. With respect to the functions called, while these have been dealt with previously, the specific work they perform in this context is not discussed. The implication is that specific work relative to your computing environment might be required to refine them. The goal here has been to trace the general steps.

Another point is that the function can be made much more efficient by using processes of culling. Culling involves determining beforehand which shapes are likely to collide before making calculations. In addition to culling, a further efficiency measure is to ensure that calculations that have already been performed once are not needlessly performed again. For example, in a sequence of time-step calculations, if objects A and B do not collide during one time-step, then they won't collide in the next time-step unless struck by an object that has changed its velocity. Conversely, two objects that are about to collide during one time-step are probably going to collide at another time-step unless struck by an object that has changed its path. Generally, then, unless one of the objects that collided in the last calculation hits something, all your other calculations will remain valid. Later chapters provide further discussion of how such observations allow you to optimize your code.

Simultaneous Collisions

Up to now, you have considered single collisions between two objects. What happens when three or more objects collide simultaneously? Consider, for example, the situation Figure 9.3 illustrates—a first ball strikes two others.

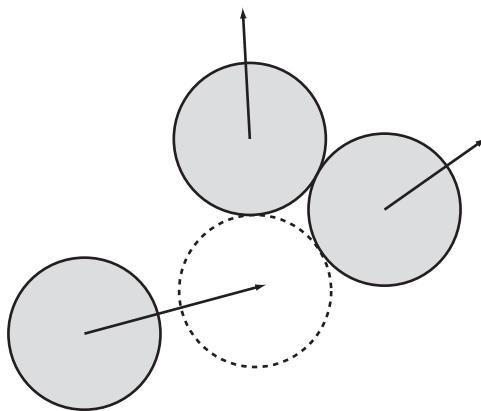


Figure 9.3

Three balls colliding simultaneously.

If you consider the likelihood that one object can collide with two others at precisely the same instant, you might object that this situation is extremely unlikely to occur. The implication is that the procedures for detecting single collisions, dealt with earlier in this chapter, suffice for all collisions. However, as it is, computers are not infinitely precise, and at some stage it might happen that you must deal with an exactly simultaneous collision.

The easiest way to resolve this problem is to cheat. By using a perturbation, you can alter the parameters of the simulation very slightly, forcing one of the collisions to happen first. In fact, this is implicitly the case in the existing function. It always chooses the first minimal collision encountered to be the one to resolve.

But you still need to be careful. After resolving the collision, a second will be encountered immediately, with a value of $t = 0$. This means that the collision time will not diminish, creating a potentially infinite loop in the function. Fortunately, in almost all situations this will not be an issue. If it were to occur, it might arise when a ball is sliding down a fixed tunnel of its exact width. Such situations can be easily avoided by not setting up your collision world so that such situations are allowed.

As shown in Figure 9.4, another example of a simultaneous collision is the Newton's Cradle. In this instance, as shown, three balls are arranged in a row touching one another. Struck by another ball at one end with velocity v , the momentum of the incident ball passes through the group, causing the last ball to move off at the same velocity. It turns out that the same process that deals with simultaneous collisions also covers this case. B_1 is moving with velocity v , but strikes ball B_2 instantly ($t = 0$). The same process passes down the chain until the final ball experiences just one collision and can move off freely.

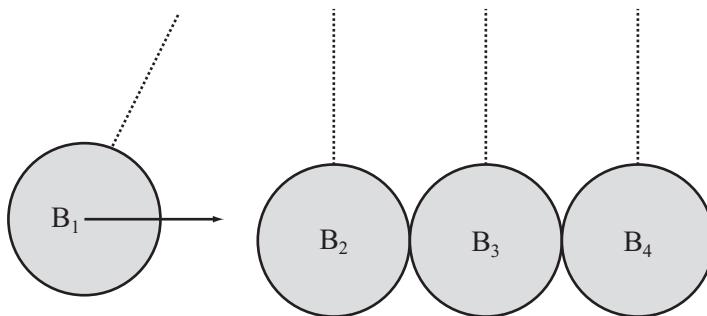


Figure 9.4
Newton's Cradle.

Exercises

EXERCISE 9.1

Complete the `checkCollision()` function by filling in the missing details, particularly the two functions `detectCollision()` and `resolveCollision()`.

This is mostly a programming exercise rather than a mathematical one. Try to write these functions in a general way that allows you to create new collision types and resolve them correctly.

EXERCISE 9.2

Create a simplified Newton's Cradle. To accomplish this, make a simulation with two fixed walls at either end, a number of touching balls in between, and one or more other balls between the group and the walls, all in a straight line. Any of the balls should be allowed to have an initial velocity along the line. See if you can use the resolution functions to make the simulation behave like a real Newton's Cradle.

Summary

After the hard work of collision detection, you should now be getting a clear idea of the power of the mathematical techniques introduced in the first part of the book. A key to success in this respect is seeing how the techniques can be applied to the physical world. In the next chapter, you'll go back to the question of collision detection and learn how to deal with more complex shapes.

You Should Now Know

- The meaning of the terms elastic, inelastic, and efficiency
- How to use conservation of momentum and energy to resolve an arbitrary elastic collision between two moving objects or one moving object and one fixed object
- How to use an efficiency coefficient to simulate an inelastic collision
- How to write a simple algorithm that will detect an unlimited number of collisions between fixed and movable objects
- How to deal with multiple simultaneous collisions

This page intentionally left blank

CHAPTER 10

DETECTING COLLISIONS BETWEEN COMPLEX SHAPES



In This Chapter

- Overview
- Problems with Complex Shapes
- Some Reasonable Problems
- Built-In Solutions

Overview

In life, you don't see many perfect circles or rectangles colliding. Most objects have irregular shapes, and as you have seen in previous chapters, collisions between even regular shapes, such as ellipses, can be very difficult to compute. Consider, for example, what happens when a ball bounces on rough terrain. Even if the ball is smooth, the terrain is not, and as a result, your calculations must include general ways to determine points of contact.

In this chapter, you will look at ways to calculate the solutions to general collision problems involving irregularly shaped objects. As involved as these tend to be, there are shortcuts you can employ to make the problems more tractable. In the process, you will also look at some ways to generate and store details of rough terrain, including how to calculate normal and tangent values.

Problems with Complex Shapes

What is it about complex shapes that make calculations concerning their collisions so difficult to perform? To start with, consider what is meant by a complex shape. For example, how can you describe a rough terrain? Suppose you are working with the collision of a ball as it bounces over a rough terrain or comes into contact with a human being's foot, as might occur in a football game. Or consider what happens when one irregular shape collides with another, as when a football player's foot comes down on the ground. In this section, you explore collision detections that involve such complexities.

Bitmaps and Vector Shapes

Collision detection begins with how computers depict images on monitors. They do so in two ways. The simplest is called a *bitmap*. A bitmap is an array of values identifying the position and color of each *pixel* (short for *picture element*) in a picture. A bitmap specifies the color and position of each pixel to a particular degree of precision. In many cases, because the complexity of the image requires that it be represented point by point, using a bitmap is the only sensible way to store an image. Consider, for example, how you might store a photograph of a painting by Seurat or any other great painter. Seurat created paintings using many thousands of points of color. If a digital photograph is made of a Seurat painting, a numerical value must be assigned to each point of color. The numerical values are then stored in an array. The array identifies the value and location of each point of color.

The alternative to using individual points of color stored in an array is to use an algorithm that generates a shape. Use of this approach can generate extremely complex images, but at the same time, it is not likely to lead to success if the objective is to recreate a painting by Seurat. Still, for shapes that are not reproductions of photographs or paintings, use of algorithms is in many ways preferable. One common approach is to use *vector* shapes. Vector shapes are defined using selected pieces of information. A circle, for instance, is a center and a radius. A rectangle is identified as four vertices.

Defining shapes as vectors presents advantages and disadvantages when compared to bitmaps. A bitmap of a 500×500 pixels square requires 250,000 pieces of information. In contrast, a vector shape of the same square requires 10 or so pieces of information. These identify the four vertices, the fact that four straight lines connect them, and the color of the lines and vertices. The amount of information required in this instance to generate a bitmap shape is 25,000 times that needed to generate a vector shape. On the other hand, producing a bitmap shape requires only that the computer run through the

array and write its contents to the monitor. In contrast, a vector shape requires calculations—repeated calls to the processor. For this reason, vector shapes can slow the computer as it performs calculations to draw images.

The same observations apply in collision detection. You can store information about a collision either as a vector or as an array of data. As an array of data, the information consists of a precise list of pixels. When a collision is involved, this is referred to as a *collision map*. As a general principle, if a great deal of information is required to describe a collision using vectors, then the complexity of describing the collision can make using vectors inefficient. Usually, the more complex a shape, the more likely it is that a plain pixel-by-pixel description will be the best. However, sometimes the best method is to combine the two approaches.

Defining a Complex Shape

To explore what happens when you define shapes using either bitmaps or vectors, consider that complex shapes come in different forms. For example, some shapes are irregular but smooth. Others are irregular and rough. An example of an irregular smooth shape is a polygon. As shown in Figure 10.1, a polygon consists of a set of vertices connected by a closed loop of straight line segments. The polygon illustrated by Figure 10.1 represents a random *convex octagon*. *Convex* here means that its sides form a figure that closes on itself somewhat like a circle. An *octagon* is an eight-sided polygon. A polygon is said to have eight straight line segments connecting the vertices v_1, v_2, \dots, v_8 in the order given. Position vectors describe the line segments. For the sake of simplicity, assume that the origin from which these position vectors are calculated is at the center of mass of the octagon.

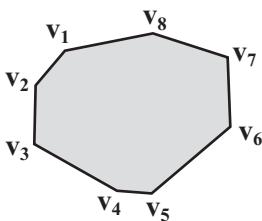


Figure 10.1

A generic convex polygon.

Note

You use the notation $P(v_1, v_2, \dots, v_8)$ to describe a generic *n*-gon. An *n*-gon is a polygon with *n* sides.

As mentioned previously, in contrast to using vertices, another way to describe a complex shape involves using a collision map. A collision map is like a bitmap of the shape. Black pixels represent the inside of the shape, and white pixels represent the outside. Certain factors play into how a collision map works. One key factor is the density of the points you use to draw the image—its resolution. In Figure 10.2, you can see collision maps of different resolutions for an amoeba-like creature. Although collision maps are expensive to calculate, sometimes they are the best solution. You can often save on calculations by using a low-resolution collision map. In Figure 10.2, the collision map is only 30×30 pixels in size.



Figure 10.2

A collision map for a blob.

A Collision Map Function

If you are using collision maps, it is generally best to calculate them in advance, either by storing them with your images, or if absolutely necessary, calculating them when the program starts. If you calculate them when the program starts, while you will save on file size, the start of your program will be slowed. The `collisionMap()` function illustrates how you can create a collision map. Smaller details are left out. For small details, you must depend on the particular language you are using. Additionally, the specific approach you use depends on the kind of image you are processing.

```
function collisionMap(image, resolution, sensitivity)
  // resolution should be an integer representing
  // the number of pixels of the original per pixel
  // of the collision map.
  // sensitivity should be a float between 0 (no
  // fuzzy edges) and 1 (the whole thing is ignored)
```

```

set map to an empty 2-dimensional array
set w to ceil((the width of the image)/resolution)
set h to ceil((the height of the image)/resolution)
repeat for x=1 to w
  set xstart to (x-1)*resolution
  repeat for y=1 to h
    set ystart to (y-1)*resolution
    set tot to 0
    repeat for i=1 to resolution
      repeat for j=1 to resolution
        add the color of the pixel at (xstart+i, ystart+j) to tot
      end repeat
    end repeat
    divide tot by resolution*resolution
    if tot<(the color of white)*sensitivity then
      set map[x][y] to 1
    otherwise
      set map[x][y] to 0
    end if
  end repeat
end repeat
return map
end

```

Among the details this function leaves vague is the code that addresses how you transform a color to a number. How you accomplish this depends on the bit depth of your bitmap. For now, think of bit depth as a number from 0 (black) to N (white).

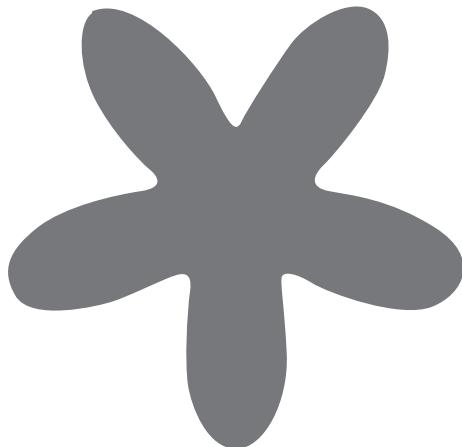
Parametric Functions

There is one other approach you can use to define a shape, and to a certain extent, you have already been using it. This approach is to use a functional description. An example of this is when you describe a circle. Just knowing that the shape is a circle means that you already know a great many things about it. In particular, you know that a point is inside a circle if and only if its distance from the center is less than the radius.

While most complex shapes do not have such a simple functional description, some do. Those that do, have a symmetrical quality about them. One of the most commonly seen is the shape shown in Figure 10.3, which traces the contours of a starfish. One appealing aspect of this shape is that describing it requires a fairly simple *parametric* equation:

$$x = r(\sin(5\alpha) + 2)\cos(\alpha)$$

$$y = r(\sin(5\alpha) + 2)\sin(\alpha)$$

**Figure 10.3**

A starfish shape drawn with a simple formula.

The starfish shape shown in Figure 10.3 is a parametric equation because both x and y are described in terms of the parameter α (and the constant r). You have encountered several such equations already. Essentially, this shape is like a circle, but in this case, the radius varies as you move around the origin.

Functional descriptions such as the one used to describe the starfish shape are useful for two reasons. First, they allow you accurately to describe a smooth shape and draw it at an arbitrarily high resolution. This also means that you can determine the normal at the point of contact. Second, they are often relatively simple to calculate. To determine if a point is on the perimeter, for example, you plug the values of x and y into the function and see if it fits. With the starfish shape, you can determine whether a point P is inside the shape by finding the angle α that OP makes with the x -axis. To accomplish this, you find the value $x = r(\sin(5\alpha) + 2)$ and compare its value to the length of OP . However, although determining whether a point is inside a shape is clearly a vital first step toward collision detection, it isn't quite enough in itself to make the collision detection efficient. Efficiency requires consideration to how the algorithms that attend to calculations are performed.

Bezier Curves and Splines

Another common type of parameterized curve is the system used in most drawing packages, called the *Bezier curve*. A Bezier curve is a vector-based description of a curved line that is defined by two sets of information. The first set of information is a list v_1, v_2, \dots, v_n of nodes, or salient points on the line. The second set of information is a list of control points for the nodes, usually described as $c_{12}, c_{21}, c_{22}, c_{31}, c_{32}, c_{41}, \dots, c_{(n-1)2}, c_{n1}$. For a curve that is open, except for the first and last nodes, two control points apply to each node. The nodes of a closed curve are uniformly associated with two nodes. Figure 10.4 shows a short open Bezier curve, with its nodes and control points marked.

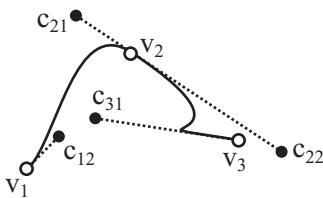


Figure 10.4

A three-node Bezier curve.

As you see in Figure 10.4, the relationship between the control points and the curve is complex. It can be summarized in two ways. First, the direction of the control point from its node gives the tangent to the curve. Second, while the distance of the control point tells you the curvature, the farther away the control point is from the curvature, the closer the curve is to the tangent.

Since longer curves are divided into sections, one between each successive pair of nodes, the simplest way to look at curves involves only two nodes. In this respect, you end with an equation such as $n_1 = (x_1, y_1)$, $n_2 = (x_2, y_2)$, $c_{12} = (p_1, q_1)$, $c_{21} = (p_2, q_2)$.

Like the starfish, a Bezier curve is parametric. This means that it is described in terms of a variable t , which varies from 0 to 1. At $t = 0$, the function evaluates to the first node. At $t = 1$, it evaluates to the second node. Between the two nodes, you get a smooth curve that follows this cubic parameterization

$$x(t) = a_x t^3 + b_x t^2 + c_x t + x_1$$

where

$$\begin{aligned}c_x &= 3(p_1 - x_1) \\b_x &= 3(p_2 - p_1) - c_x \\a_x &= x_2 - x_1 - 3(p_2 - p_1)\end{aligned}$$

and similarly for $y(t)$.

If you plug the values 0 and 1 for t into this formula, you'll see that it yields the values x_1 and x_2 , as required. Also, if you differentiate once with respect to t and again plug in the value $t = 0$, you'll see that you get the value c_2 . The magnitude of the tangent with respect to x at that point is proportional to the distance between the control point and the node.

Catmull-Rom Curves

A Bezier curve is not the only way to create a parameterized cubic function. Another method is the *Catmull-Rom spline*. While a Catmull-Rom spline is often used as a general term for a parameterized curve, instead of working with control points, it employs a method for interpolating that uses four points that lie on the curve itself. Figure 10.5 illustrates how the Catmull-Rom spline is generated. The four nodes n_0, n_1, n_2, n_3 precisely define the curve segment between n_1 and n_2 . By creating a chain of $n + 2$ such points, you can define n segments. Since each of the segments has the same tangents at the end points, you arrive at a smooth curve. Notice that since the first and last control points don't precisely define a segment, you can't draw in a curve at the ends.

As with a Bezier curve, a value between 0 and 1 parametrically defines the Catmull-Rom curve. The function is used to generate the Catmull-Rom spline is as follows:

$$p(t) = \frac{1}{2}(2n_1 + (-n_0 + n_2)t + (2n_0 - 5n_1 + 4n_2 - n_3)t^2 + (-n_0 + 3n_1 - 3n_2 + n_3)t^3)$$

If you plug in the value $t = 0$, you'll see that you get the answer n_1 , and if you put in $t = 1$, you get n_2 . While Catmull-Rom splines are simpler than Bezier curves, they have a disadvantage in that they can't be used to define sharp corners at individual nodes.

Note

In Chapter 21, attention is given to a much more complex kind of spline that can overcome such limitations. This is the non-uniform relational B-spline (NURBS).

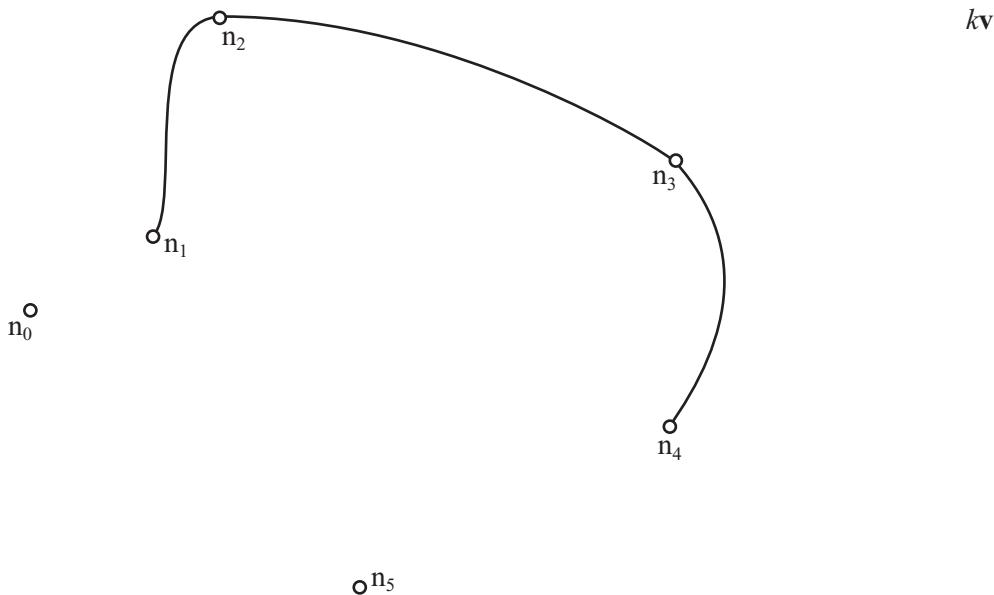


Figure 10.5
A Catmull-Rom spline.

Because a cubic polynomial is a fairly simple function, it's relatively easy to calculate collisions against these splines. In particular, it's possible to find the intersection of a straight line with a Bezier or Catmull-Rom spline. You're in the usual territory here. The goal is to look for a point lying on both lines. To accomplish this, using the notation applied to Bezier curves, you solve two simultaneous equations, as follows:

$$u_1 + sv_1 = a_x t^3 + b_x t^2 + c_x t + x_1$$

$$u_2 + sv_2 = a_y t^3 + b_y t^2 + c_y t + y_1$$

Substituting for s , you find that

$$u_1 + \frac{v_1}{v_2} (a_y t^3 + b_y t^2 + c_y t + y_1 - u_2) = a_x t^3 + b_x t^2 + c_x t + x_1$$

$$(v_1 a_1 - v_2 a_x - v_1 a_y) t^3 + (v_2 b_x - v_1 b_y) t^2 + (v_2 c_x - v_1 c_y) t + (v_2 (x_1 - u_1) - v_1 (y_1 - u_2)) = 0$$

This is just another cubic equation. All of the coefficients can be calculated. Using the method detailed in Chapter 3, you can solve this cubic. From then, you can find the values of t and s .

Movable Splines

Another solvable problem is dealing with collisions between a moving line and a stationary spline or vice versa. Solving this requires that you employ a useful property of parametric curves presented in Chapter 6. This property establishes that the gradient of the curve at a particular value of t is given by the quotient of the derivatives with respect to t . Here is how the property is expressed:

$$\frac{\frac{dy}{dt}}{\frac{dx}{dt}} = \frac{dy}{dx}$$

To find the gradient of a spline (and hence a collision normal), you use the following approach:

$$\frac{dy}{dx} = \frac{3a_y t^2 + 2b_y t + c_y}{3a_x t^2 + 2b_x t + c_x}$$

When calculating collisions between a line and a cubic parameterization, you can take advantage of the fact that at the moment of impact, assuming the point of collision is not at a vertex or endpoint of the line segment, the line and the curve meet along a tangent to the curve. As a result, the gradient of the curve must be equal to the gradient of the line. This gives you the following line of reasoning:

$$\frac{v_2}{v_1} = \frac{3a_y t^2 + 2b_y t + c_y}{3a_x t^2 + 2b_x t + c_x}$$

$$3(v_2 a_x - v_1 a_y) t^2 + 2(v_2 b_x - v_1 b_y) t + (v_2 c_x - v_1 c_y) = 0$$

What results is a simple quadratic that is easy to solve and yields at most two points in a particular section of the curve that could collide with the line, depending on its direction of travel.

Some additional calculations are required to get the complete picture. These tasks will not be covered here, but it is important to be aware of them. For example, you have to find out if either of the points lies within the line segment during its motion. Likewise, for a Bezier curve with sharp corners, you also calculate point-line collisions with the nodes of the curve and point-spline collisions with the ends of the line segment. As you proceed, you eventually find that you can readily calculate collisions between a cubic spline and any polygon.

You might conclude that if you take the current line of reasoning far enough, you will be able to calculate almost any collision. As it is, this is not possible. For situations where a single object is described with a spline, or where other objects are described with polygons, these functions can be enough. Your approximation methods can help. However, detecting collisions between splines and circles, ellipses, or even other splines presents a different prospect. While from the start they can't be done anywhere near as easily, you eventually end up with equations of the fifth or even the sixth order that cannot be solved algebraically.

Convex and Concave

Shapes can be broadly divided into two kinds: convex and concave. You've encountered them before, and there is a fairly simple way to define them. As Figure 10.6 illustrates, a shape is concave if there are three points P_1, P_2, P_3 on the perimeter such that the interior angle P_1, P_2, P_3 is greater than 180° . If no such points exist, the shape is convex. The definition works equally for polygons and smooth shapes, and the three points can be any points at all on the perimeter.

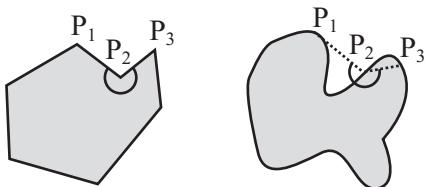


Figure 10.6
Two concave shapes.

As a general rule, convex shapes are relatively straightforward. Any line intersects them at most at two points. If in doubt, you can describe a convex shape arbitrarily accurately with a convex polygon. Then collision detection is just the same as with a rectangle, only with more line segments to check. The `pointPolygonCollision()` function provides a generic approach to collision detection involving convex shapes:

```
function pointPolygonCollision(pt, displacement, poly)
    // here poly is an array of vertex points in order
    set t to 2
    set c to the number of points in poly
    repeat for i=1 to c
        set p1 to poly[i]
        set p2 to poly[(i mod c)+1]
        set t1 to intersectionV(pt,displacement,p1,p2-p1)
        if t1="none" then next repeat
        set t to min(t,t1)
    end repeat
    if t=2 then return "none"
    return t
end function
```

As becomes apparent after brief inspection, the `pointPolygonCollision()` function closely resembles the functions presented for detecting collisions involving rectangles. The primary difference is that more sides are defined. Among possible improvements, the speed of the function can be improved if you find the leading edges of the shape. This topic is covered later in the chapter.

While the `pointPolygonCollision()` function works with convex shapes, concave shapes are a different matter. A generalized concave shape can be as convoluted as you want to make it. Consider, for example, a maze of caves. Any line might intersect the shape any number of times, and with extremely complex shapes, you can't even reliably approximate them using a polygon. In fact, if you are working with a shape generated as a fractal, it is difficult to determine even if a point is inside a generalized concave shape. However, for shapes such as those discussed in this chapter, approximating polygons works well.

Determining If a Point Is Inside a Shape

Once you determine an intersection between a moving point or a line segment and a shape, you can use this information to answer a fundamental question: is a particular point inside the shape?

As you have seen, in many cases, especially when the shape has some functional description, this question can be resolved reasonably simply. In other cases, you need a more general solution. For this you can use *raycasting*. Raycasting is a technique that comes into its own in 3-D programming. A ray is an infinitely long line in one direction. It is analogous to a beam of light from a flashlight. Using your point/shape collision routines, you can determine the intersection of this ray with the shapes you are interested in.

If you have a point P inside a shape S , what happens when you cast a ray from that point to infinity? As Figure 10.7 illustrates, since S is a closed line, at some stage it has to intersect with S . If S is concave, then the ray can intersect again somewhere else. If this happens, then the ray is once again inside the shape, and if it is inside the shape, then you know that somewhere down the line it must intersect the shape yet again. A couple of additional observations arise. First, if it starts from within the shape, ultimately the ray must intersect an odd number of times. Conversely, a ray cast from outside the shape must intersect with the shape either 0 or an even number of times.

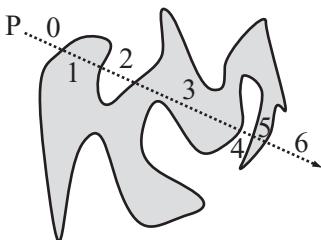


Figure 10.7

Casting a ray to determine if a point is inside a shape.

As clear as the division between even and odd collisions might be, there is also the possibility that a ray might touch S at a tangent. In this case, the ray meets the shape but does not intersect. This is potentially a major problem, but as long as S is a polygon, you can deal with the special case. If you do not deal with the special case, to be absolutely sure you need to cast two or more rays. If two rays with a small angle between them agree on the disposition of the point, then you can accept their answer with reasonable confidence for a sufficiently simple shape. If they disagree, you can cast a third ray to confirm the answer.

The `pointInsidePolygonIncomplete()` function tests for intersections of lines with polygons. It does not test for the special case of tangential rays:

```
function pointInsidePolygonIncomplete(pt,poly)
  // choose an arbitrary point outside the polygon
  set mx to the maximum x-value in poly
  set outpoint to (mx+10,0)
  // now count the intersections along the ray from pt to outpoint
  set intersections to 0
  set c to the number of points in poly
```

```

repeat for i=1 to c
  set p1 to poly[i]
  set p2 to poly[(i mod c)+1]
  set t to intersection(p1,p2,pt,outpoint)
  if t="none" then next repeat
  add 1 to intersections
end repeat
if (intersections mod 2)=1 then return true
otherwise return false
end function

```

What about those rays that touch the perimeter without passing through it? First of all, consider the case of a ray that meets the perimeter along one of the sides. If it does not pass through the side, it must be parallel to it. If it is parallel, it also passes through a vertex. In fact, it passes through two vertices. Given this information, all you need to consider is the case where the ray meets a vertex.

You might have noticed a small change between `pointInsidePolygonIncomplete()` and the earlier versions of the collision detection function. If the ray intersects at a vertex, the routine ignores it and doesn't count it. This is clearly wrong, but it can be fixed at the same time that you fix the tangent problem. With the `pointInsidePolygon()`, if the ray intersects at a vertex, you abandon it and try a slightly different ray. Because the shape dealt with is a polygon and so has a finite number of vertices, this technique is guaranteed to work.

```

function pointInsidePolygon(pt, poly, outpoint)
  if outpoint is not defined then
    // choose an arbitrary point outside the polygon
    set mx to the maximum x-value in poly
    set outpoint to (mx+10,0)
  end if
  // now count the intersections along the ray from pt to outpoint
  set intersections to 0
  set c to the number of points in poly
  repeat for i=1 to c
    set p1 to poly[i]
    set p2 to poly[(i mod c)+1]
    set t to intersection(p1,p2,pt,outpoint)
    if t="none" then next repeat

```

```

if t=0 or t=1 then
    // try a different ray
    return pointInsidePolygon(pt, poly, outpoint+(0,100))
end if
add 1 to intersections
end repeat
if (intersections mod 2)=1 then return true
otherwise return false
end function

```

There are other ways to determine whether the ray meets the vertex tangentially. The simplest is to check for an intersection with the line segment joining the two neighbors of the vertex and the ray. As illustrated by Figure 10.8, if the intersection occurs within the line segment, then the ray meets the vertex tangentially. However, unless the polygon has a large number of vertices, the method given is fairly simple and unlikely to need more than two runs.

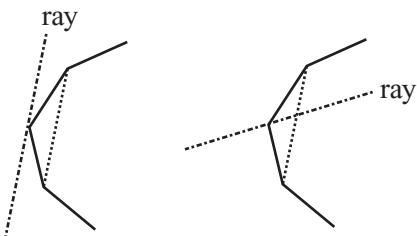


Figure 10.8

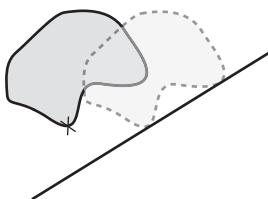
Determining the type of intersection with a vertex.

Some Reasonable Problems

Collision detection between arbitrary shapes is always going to be computationally expensive. In light of this, it is helpful to review a few of the standard approaches. They provide benchmarks for improvements you can explore on your own.

Finding the Leading Edge of a Complex Shape

If a shape S is moving along like a glob of jelly on a constant vector toward a solid wall, you can save a lot of time by pre-calculating its leading point relative to the wall. In other words, as shown by Figure 10.9, you pre-calculate the first point on the perimeter of S that will hit the wall. There may be more than one leading point, each of which strikes the wall at the same time, but you only need to find one.

**Figure 10.9**

The leading point of a complex shape relative to a wall.

For an image known only by a bitmap description, the simplest way to calculate the leading point is checking all the points and finding the first one in a particular direction. For a polygon, you can accomplish this task with little effort. What applies to polygons also applies to functionally described objects generally.

Toward this end, suppose first that S is a polygon. Using the same technique used with circles, you can find the distance from each vertex to the wall in the direction of motion. In this case, the leading point is simply the closest one. The `leadingPointOfPolygon()` function applies this approach:

```
function leadingPointOfPolygon(poly, vel, wallPt, wallVect)
    set min to -1
    set minpt to 0
    // calculate the normal to the wall
    set n to norm(normal(wallVect))
    if dotProduct(n,vel)<0 then set n to -n

    repeat for each pt in poly
        set c to component(wallPt-pt, n)
        if c<0 then return "past"
        if min=-1 or c<min then
            set min to c
            set minpt to pt
        end if
    end repeat
    return pt
end function
```

Although the `leadingPointOfPolygon()` function has many uses, it can be made more helpful if it is changed so that it finds the leading edge first. The leading edge is the set of points on the perimeter that can strike any wall or any other static object. Recall how this type of collision was calculated between rectangles. For a particular direction of motion, at least one point on the rectangle existed that could never collide with another.

The leading edge is somewhat more complicated than the leading point. For one thing, it is no longer just vertices that you are interested in. Instead, you are interested in whole edges. To picture this, suppose that the shape is going to collide with a particle rather than a wall. If this is the case, then the particle could just as easily collide anywhere along an edge without ever meeting a vertex. One result is that sides might conceivably hit.

To explore the possibilities this scenario introduces, start with a convention. Suppose that all shapes have their vertices listed in a counter-clockwise direction, as shown in Figure 10.10. This allows you to conveniently avoid worrying about which side is inside and which side is outside. As long as the vertices are always in counter-clockwise order, the inside is always on the counter-clockwise side of the edge between two consecutive vertices. As Figure 10.10 illustrates, you can then always take the clockwise normal of the edge to get a normal pointing outward.

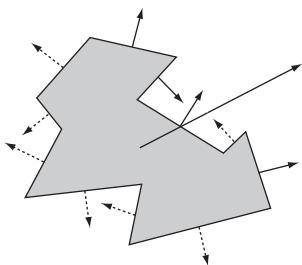


Figure 10.10
Finding the leading edge of a polygon.

To move this discussion one step farther, suppose that the polygon is traveling along the vector \mathbf{v} . The edges that could collide are those where the inside is pointing away from the vector \mathbf{v} . In other words, the edges are those where the dot product of the normal with \mathbf{v} is positive. In Figure 10.10, these are the solid arrows. The `leadingEdgeOfPolygon()` function applies this notion:

```

function leadingEdgeOfPolygon(poly, vel)
    set edges to an empty array
    set c to the number of points in poly
    repeat with i=1 to c
        set v to poly[(i mod c)+1]-poly[i]
        if dotProduct(clockwiseNormal(v),vel)<0 then
            append array(poly[i],v) to edges
        end if
    end repeat
    return edges
end function

```

Note

As used by the `leadingEdgeOfPolygon()` function, the `clockwiseNormal()` function is presented generically. Its definition is a task taken up in Chapter 13.

After you have the set of leading edges, collision detection is simplified a great deal. Essentially, it is reduced by roughly half. To detect a collision with S , you need to calculate only the first collision with any one of the target line segments.

Given that you have knowledge of the target line segments, you can return to leading points, particularly the leading points of functional shapes. As with polygons, the leading point of any shape with respect to a wall is nothing more than the closest point to the wall. Finding this is a matter of calculus, and while in the general case it can be very complicated, if you restrict what you are looking for, it can be relatively simple.

Since most shapes described in functional terms can be expressed in terms of a parametric function, each point on the shape is equal to $(x(t), y(t))$ for some functions x and y and some value of t . For example, in a circle the functions might be $x(t) = r\cos(t)$, $y(t) = r\sin(t)$. To determine the nearest such point to a wall, you must find the minimum distance to it. As before, you can do this by first determining the normal $(n_1 \ n_2)^T$ to the wall in the appropriate direction. Next, you find the normal component of an arbitrary perimeter point to some reference point $(p_1 \ p_2)^T$ on the wall. Here is one approach to accomplishing this task:

$$D(t) = n_1 (p_1 - x(t)) + n_2 (p_2 - y(t))$$

To find the minimum value of D , you differentiate with respect to t . This gives you

$$D(t) = -n_1 \dot{x}(t) - n_2 \dot{y}(t)$$

Notice that the constants representing the reference point disappear. That this occurs can be helpful because the leading point shouldn't be affected by the distance to the wall.

Any minimum value of the function $D(t)$ is to be found at a zero value of $\dot{D}(t)$. Finding such zero values, of course, depends on the functions x and y . Suppose, for example, that x and y are for a circle. Then you have these equations:

$$\begin{aligned}\dot{x}(t) &= -r \sin(t) \\ \dot{y}(t) &= -r \cos(t)\end{aligned}$$

As a result, $\dot{D}(t) = n_2 r \cos(t) = n_1 r \sin(t)$.

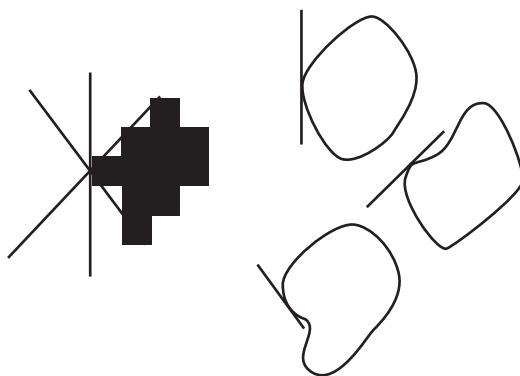
Given that \mathbf{n} is a unit vector, it is also equal to $(\sin(s), \cos(s))$ for some value of s . This means that $\dot{D}(t) = 0$ if and only if $\sin(s) \cos(t) - \cos(s) \sin(t) = 0$ (eliminating the constant factor r). Using trigonometric identities, this is the same as $\sin(s - t) = 0$. The result of this is that either $s - t = 0$ or $s - t = \pi$. (It can also be other integer multiples of π .) The outcome is that the leading point could either be the point with the same angle as the normal vector or the one diametrically opposed.

Using a Collision Map

In many cases, nothing but a bitmap description of an object will do. Collision maps thus come into play. Collision maps come in two forms. One is a full collision map. The other is a *height map*. A height map is a collision map in one dimension. Height maps are most suitable for terrains, which as a general rule are single lines at varying heights. Likewise, they do not have “overhangs.” When maps do not have overhangs, they are produced by single-valued functions, such as $y(x)$.

As a general rule, you do not want a simple height map or collision map. These are useful for telling you where the edge of a shape lies, but they don't give you the all-important information about the normal at that point. After you have calculated a collision or height map, you must create a second map that stores the normal information for each point on the perimeter. More usefully, rather than taking a redundant approach, you can encode this information for each edge point into a single collision map.

To gather this information, however, is a surprisingly tricky problem. Because your information is pixel-by-pixel rather than vector-based, it is impossible to know the exact tangent at any point. Consider Figure 10.11, for example, which shows a scaled-up view of a particular collision map for a smooth object. As you can see, any of the lines marked could be the correct tangent at a particular point of impact. In each case, you see three curves with the appropriate tangents.

**Figure 10.11**

Problems with finding the tangent to a collision map.

The solution is to make an educated guess, as is done in the `calculateNormals()` function. If you find the gradients between adjacent pixels on each side of a point P , you can average them out to guess the correct gradient at P . However, this approach also depends on the shape being relatively smooth. Bumps and curves must be relatively flat in comparison to the size of anything you want to collide with them.

You can begin with height maps, as these are reasonably simple:

```
function calculateNormals(heightMap)
    set normals to an empty array
    repeat with i=1 to the number of elements of heightMap-1
        set hd to heightMap[i+1]-heightMap[i]
        set v to norm(-hd, 1)
        if i=1 then
            set thisNormal to v
        otherwise
            set thisNormal to (v+lastNormal)/2
        end if
        append thisNormal to normals
        set lastNormal to v
    end repeat
    append lastNormal to normals
    return normals
end function
```

For a more general collision map, detection becomes a little trickier. In more general applications, you assume that your collision map is in the form of a black-and-white image. You need to start by tracing the edge of this image. This can be done by allowing each pixel to have three possible values. The values consist of black for the interior, white for the exterior, and gray for an edge. An edge pixel is a black pixel with a white pixel as an immediate neighbor. The `findEdges()` function provides an outline of how this is accomplished:

```
function findEdges(bwImage)
    set newImage to a copy of bwImage
    repeat for each point in bwImage
        if the point is black then
            if at least one neighbor of the point is white then
                set the equivalent point in newImage to gray
            end if
        end if
    end repeat
    return newImage
end function
```

Use of the `findEdges()` function furnishes you with an image outlined in gray.

After you possess such a map, the technique for finding the normals is much the same as demonstrated previously. A useful shortcut is to save the normal information in the collision map image itself. Specifically, instead of using just black, white, and gray, allow the amount of gray to vary according to the angle of the normal. If the image has 8 bits per pixel (a grayscale image), you have 254 possible values for each edge pixel, as well as 255 for an empty pixel and 0 for an interior pixel. Thus if the pixel has a value of, say, 200, a normal angle of $200 \times \frac{2\pi}{254}$ would be represented. The `calculateNormals()` function shows how this is accomplished.

```
function calculateNormals(collisionMap)
    set newImage to a copy of collisionMap
    repeat for each point in collisionMap
        if the color of the point is white or black then next repeat
        set clist to an empty array
        set neigh to 0
        repeat for neighbor in (0,1), (1,0), (0,-1), (-1,0)
            if the color of point+neighbor is white then
                append 0 to clist
```

```

        otherwise
            append 1 to clist
            add 1 to neigh
        end if
    end repeat
    if neigh=0 then set v to 0
    if neigh=1 then set v to -(the neighbor vector)
    if neigh=2 then
        if clist[1]=1 and clist[3]=1 then set v to (1,0)
        if clist[2]=1 and clist[4]=1 then set v to (0,1)
        otherwise set v to -(the average of the neighbor vectors)
    if neigh=3 then set v to the non-neighbor vector
    set v to norm(v)
    set the color of the corresponding
        point of newImage to writeColor(v)
    end repeat
    return newImage
end function

```

To process color values, you call the `writeColor()` function, which is defined as follows:

```

function writeColor(v)
    set a to atan(v[2],v[1])
    return integer(a*127/pi)
end function

```

The action that corresponds to the writing color values is reading color values, and the `readColor()` function attends to this. It is defined as follows:

```

function readColor(c)
    set a to c*pi/127
    return vector(cos(a),sin(a))
end function

```

Note

The `calculateNormals()` function is fairly crude. The only possible normal vectors are at multiples of 45°. In Exercise 10.2, you are asked to find a way to smooth out the normals.

After you have a collision map complete with normals, it is a fairly reasonable problem to find the intersection of the map with a moving particle. Since height maps are simpler to work with, a good starting point is with them. Suppose that the particle at P is moving with displacement s near a ground with height map H , which you will assume begins at $x = 1$. The particle travels with its x -coordinate varying from p_1 to $p_1 + s_1$. If at any of these positions of x , the y -coordinate is greater than or equal to the height map coordinate as you measure downward, the particle will collide. The `particleHmapCollision()` function puts this approach to work:

```
function particleHmapCollision(p, s, h)
  if s[1]=0 then
    // vertical motion: only one check
    if h[p[1]]<=p[2]+s[2] then
      return (h[p[1]]-p[2])/s[2]
    otherwise
      return "no collision"
    end if
  otherwise
    set grad to s[2]/s[1]
    repeat for i=0 to s[1]
      set x to p[1]+i
      set y to p[2]+grad*i
      if h[x]<=y then
        return s[1]/i
      end if
    end repeat
    return "no collision"
  end if
end function
```

Observe that the `particleHmapCollision()` function does not return the normal. However, to do so can be accomplished with the help of the accompanying normal map.

An essentially identical process can be used for the general collision map. In this case, instead of only testing for a height's being greater, you also check whether the particle at any time is inside the shape. Since you marked the interior points black, this is not difficult. In the `pointCmapIntersection()` function, you have a shape at position Q with collision map C :

```

function pointCmapIntersection(p, s, q, c)
  set d to magnitude(s)
  if d=0 then return "no intersection"
  set sn to s/d
  set st to p-q
  repeat with i=1 to integer(d)
    set pos to st+i*sn
    set col to the color of the pixel in c at pos
    if col is not black then return d/i
  end repeat
end function

```

Although this function works well, it can be improved in a number of ways. For one thing, it is slow. Likewise, it requires revision if the normal is to be found. In this respect, consider that if the color of the pixel is black, then the particle has overshot the edge. However, since you are only checking pixel-length sub-steps of the motion, the edge pixel is, at most, one pixel away. It would also be sensible here to use integer-only calculations. The main avenue to reach this goal is Bresenham's Algorithm, which is detailed in Chapter 22. Another optimization would be to avoid checking the full range of motion. Instead, check only the end point. You need to calculate the precise intersection point only if the end point lies inside the shape. But this approach is reliable only if the particle's displacement is small relative to the size and irregularity of the shape.

Another way to speed things up is to use what might be called a *collision halo*. Instead of making the outside of the shape plain white, you can use shades of gray to mark the distance to the nearest edge point of the shape. By testing this value, it is possible to calculate whether a collision is possible from any particular position.

If the colliding object is larger than a point particle, calculating collisions becomes more tedious. For any potentially irregular shape, there is no substitute for testing collision against the entire leading edge of the moving object. Depending on how much you can rely on your colliding objects being reasonably regular, you can take some shortcuts, but for truly complex worlds, you just have to bite the bullet.

Finding Bounding Shapes

One technique that can greatly speed up collision detection is to use a *bounding shape* or *armature*. You have been doing this implicitly with the collision maps above. A bounding shape is a simple shape that entirely encloses a colliding object. For example, if you are testing for collision with a person, you might enclose the person in a rectangle. If you are colliding with your starfish shape, this might be enclosed in a circle.

When you use a bounding shape, before going down to the more detailed level and checking for precise pixel-level collision, you can pre-calculate whether there is any chance that two objects are going to collide. To use a bounding shape effectively, it is important to find a shape that is both simple and encloses as much of your object as possible. A bounding circle around a lamp post would stretch a long way beyond its actual collision range. A thin rectangle would not.

As a general rule, it is best to decide in advance what kind of bounding shape is most appropriate. It might be a rectangle or circle. It might be a triangle or any other polygon. After you have decided which bounding shape to use, you need to calculate the dimensions of the bounding shape. Especially with respect to polygons, circles provide a ready option. If your polygon has vertices $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n$, then the center of the circle can be placed at the average of the vertices, $\frac{1}{n} \sum_{i=1}^n \mathbf{p}_i$. Observe that this is also the center of mass of the shape. As the construction of the `boundingCircle()` function shows, given this basis of understanding, all you need to do is find the maximum distance of the vertices from the center, which is the radius of the circle:

```
function boundingCircle(poly)
  set c to the number of vertices in poly
  set s to (0,0)
  repeat for each v in poly
    add v to s
  end repeat
  set center to v/c
  set mx to 0
  repeat for each v in poly
    set d to magnitude(v-center)
    if d>mx then set mx to d
  end repeat
  return array(center, mx)
end
```

The `boundingCircle()` function won't find the absolutely smallest possible circle, but it works well enough for a reasonably regular shape. Better algorithms are required to find the smallest possible bounding circle. This applies, likewise, to other shapes.

Bounding boxes are a little more difficult. It's not hard to find a bounding box along a particular axis, but not all axes are as good as one another. Two common options are the *axis-aligned bounding box* (AABB) and the *object-aligned bounding box* (OABB). In the first of these, the box is aligned along the principal axes of your simulation. These are usually the *x*- and *y*-axes. If you are working in three dimensions, this includes the *z*-axis.

If all the objects in the simulation have bounding boxes aligned along the same axes in this way, collision detection between the bounding boxes is greatly simplified. You can test for collisions by looking at *x* and *y* components. You then remove many of the dot product calculations in more generalized algorithms. On the other hand, if your objects are not shaped in such a way that an AABB is a close fit, or if they are rotating so the AABB changes over time, then you can lose many of the advantages of simplification.

An OABB is often a better option than the AABB. In this case, the axes are chosen to fit as closely as possible to the shape, without worrying about their orientation. Although this complicates collision detection, it is much more general, and when you use it, you don't have to recalculate the bounding box whenever the object rotates.

Figure 10.12 illustrates examples of both methods. In the image on the left, the shape has been enclosed by an AABB oriented along the *x*- and *y*-axes. This rectangle is much too large and does not represent the shape well. In the second image, on the right, you choose a better pair of axes, forming an OABB that snugly encloses the shape.

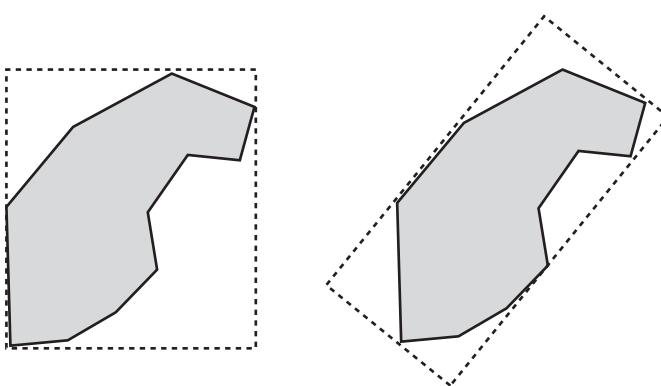


Figure 10.12

An AABB and an OABB for the same shape.

Finding the best OABB is a tricky mathematical problem, essentially the same as *factor analysis*. You must find the axis that *minimizes* the *maximum* distance to the vertices, which is called a *line of best fit*. Generally, you calculate this using the *least-squares method*. When you employ the least-squares method, your objective is to find a line such that the squared perpendicular distance of all vertices from the line is minimized. Although minimizing the absolute value would generally be better, it is harder to do analytically. This becomes the long axis of your rectangle. The short axis is perpendicular to it, with half-length equal to the maximum distance. You can use a similar technique in three dimensions, too.

The least squares method is detailed but straightforward. You start by assuming that you have a list of points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. You must then calculate the mean of the x and y variables, which you will call \bar{x} and \bar{y} , respectively. You also calculate the variance of the variables, which is a measure of how spread out they are from the mean. The variance is defined generally as the square of the standard deviation. The variance of a set of data is given by the following formula:

$$v_x = \frac{1}{n} \sum_{i=1}^n (x_i^2) = \bar{x}^2$$

Note

The formula for variance makes use of *sigma notation*. The large Greek letter sigma, with “ $i = 1$ ” at the bottom and n at the top, signifies the *sum* of the subsequent values. The index i is to be replaced with each of the values from 1 to n .

Having calculated the variance, you calculate a quantity that can be called S , which is equal to

$$S = \frac{n(v_x v_y)}{\sum_{i=1}^n x_i y_i - n \bar{x} \bar{y}}$$

The denominator here is a kind of two-dimensional version of the variance, which can be called v_{xy} .

Given these preliminaries, the line of best fit is given by the equation $y = ax + b$, where $a = -2S \pm \sqrt{4S^2 - 1}$ and $b = \bar{y} - a\bar{x}$.

Using this approach, you can define the `lineOfBestFit()` function, which makes use of two other functions that attend to mean and variance values:

```
function lineOfBestFit(dataPoints)
    set xlist to arrayOfValues(dataPoints,1)
    set ylist to arrayOfValues(dataPoints,2)
    set n to the number of elements in dataPoints
    set mx to mean(xlist)
    set my to mean(ylist)
    set sx to variance(xlist)
    set sy to variance(ylist)
    set sxy to variance(xlist,ylist)
    if sxy=0 then return "vertical"
    set s to n*(sx-sy)/sxy
    set a to -2*s+sqrt(4*s*s-1)
    set b to my-a*mx
    return array(a,b)
end function
```

As anticipated by the previous discussion, the `mean()` function takes the following form:

```
function mean(list)
    set s to 0
    set n to the number of elements in list
    repeat for i=1 to n
        add list[i] to s
    end repeat
    return s/n
end function
```

Likewise, the `variance()` function makes use of the `mean()` function:

```
function variance(list,list2)
    if list2 is undefined then set list2 to list
    set m1 to mean(list)
    set m2 to mean(list2)
    set s to 0
    set n to the number of elements in list
    repeat for i=1 to n
        add list[i]*list2[i] to s
    end repeat
    return s/n-m1*m2
end function
```

Having found the line of best fit, you can convert it into vector form, giving you the direction and position of the principal axis of the shape. The perpendicular vector gives you the minor axis. You can then calculate the maximum distance of the vertices from each of these axes, giving you the length of the OABB's sides.

After placing your bounding shape, of whatever kind, the technique used for collision detection is the same for all. As far as the physics is concerned, you treat the underlying shape as if it is identical to the bounding shape. When there is a potential collision, you can switch to full collision detection, or depending on how closely the shape matches its bounding shape, you might decide to skip the full collision detection altogether. If something is nearly a circle, you can just treat it as a circle. To the naked eye, it will look like a circle.

Built-In Solutions

This section stands on its own because it's not strictly mathematical. You might find that it is possible to use built-in functions for dealing with arbitrary shapes in your language. Some languages, for example, might offer an `intersects()` function. Although these functions have drawbacks, there are times when using them is the best approach.

It's important to remember that at heart, these functions represent versions of the functions discussed in this chapter. If you are working with an application that uses bitmaps, such functions work with a collision map system. If you are working with an application that uses vectors, the functions employ a vector-based engine. Vector functions are likely to work with vector methods such as polygon and spline functions. Whatever their advantages, however, you are likely to find that they have the same drawbacks as the functions discussed in this chapter. Principally, the more complex the shapes, the harder work for the computer.

You can make life easier for the processor by saving it some work. One way to do this is to use a collision proxy. A collision proxy is a shape that offers a version of the colliding object that is simplified and often scaled down. At the start of this chapter, you used a collision proxy when creating collision maps. To a lesser extent, you also did this when creating bounding shapes. Generally, collision proxies offer a useful technique when developing customized collision detection. In this respect, you can *slave* your real colliding objects to the proxies. In other words, you make it so that the actual physics happens to the proxy objects somewhere unseen while the real objects follow them blindly. All that the engine needs to do is calculate collisions for the proxies.

Exercises

EXERCISE 10.1

Write a function named `splitPolygon(poly)` that takes an arbitrary polygon with its vertices numbered clockwise and split it into triangles.

This is harder than it sounds. Use a recursive function that finds three adjacent corners and makes a triangle from them, leaving a polygon with one less vertex. If it is to work for polygons generally, you must make sure that the triangle you make does not intersect with any other sides.

EXERCISE 10.2

Write a function named `smoothNormals(collisionMap)` that takes a collision map generated by the `calculateNormals()` function and generates a new map with more realistic normals.

The function should use a system that is similar to the one used with height maps, averaging out the normal vectors between neighboring edge points. You might also be interested in trying to implement the collision halo suggestion mentioned in the chapter.

Summary

Although this chapter presents many of its concepts generally, it provides you with a good overall sense of how collision detection techniques can be applied. This is not the last discussion this book offers of collision detection. In fact, there is a great deal left to go. At this point, however, be assured that you have covered a lot of ground and laid the foundation for what is to come.

In the next chapter, you will apply the techniques you have learned so far to create a game.

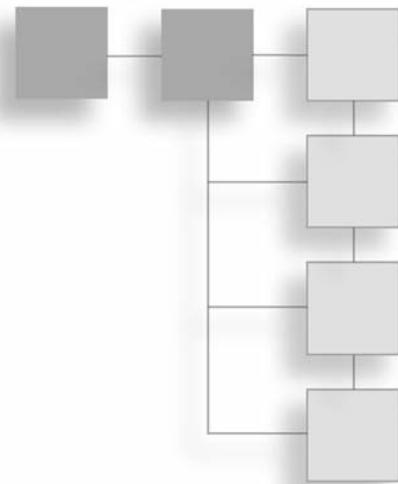
You Should Now Know

- How to describe a general shape using either a bitmap or vector description
- How to define a curve using a Bezier or Catmull-Rom spline
- The meaning of the terms *convex* and *concave* and how to recognize a concave polygon
- How to detect collisions with an arbitrary polygon
- How to calculate the leading point of any shape when meeting a wall, and the leading edge of a polygon
- How to calculate a collision map or height map, including edge normals, and calculate collisions between a particle and either of these
- How to calculate a bounding circle or 2-D object-aligned bounding box (OABB)
- How to use proxies to speed up collision detection

This page intentionally left blank

CHAPTER 11

A SIMPLE POOL GAME



In This Chapter

- Overview
- Primary Elements of a Simulation
- Taking a Shot

Overview

Having explored the techniques presented in the previous chapters, you have opened the door to a huge variety of simulations and games based on real-life physics. Using the basic collision methods, you can create games like pool, pinball, marbles, pong, and breakout. Further, the path you have covered so far has carried you far toward being able to work with more complex collision events, such as those involved in Lemmings or Worms.

In this chapter, you put some of the techniques investigated so far to work to create a standard pool game. Working with a game is not like working with individual functions. While functions allow you to isolate mathematical operations and scope out algorithms in pseudocode, constructing a game requires that you attend to specific details of the programming environment in which you are working. In this respect, while this chapter represents an advance beyond the previous chapters, it still imposes limitations. The code samples, once again, are generic, and to fully implement them, you must adapt them to the environment in which you are working. Similarly, there are a number of details of game logic, such as determining fouls, tracking players, or judging whether a game has

been won, and these are not covered here. Like previous chapters, then, this chapter continues to concentrate on the mathematical elements, but at the same time, the context is expanded so that you see a common and useful context of application.

Primary Elements of a Simulation

A pool game falls into the general category of *realistic simulation*. To develop a game that involves realistic simulation, one of your first tasks is to establish the framework of the simulation. This involves identifying the context in which the simulation is to occur and the limitations you wish to impose on the simulation. To reach such objectives, it is important to consider, first, what you have to work with in relation to what you wish to create. In many instances, your tools restrict what you can hope to arrive at. In other cases, the tools you use allow you to do much more than you might initially think possible. In this instance, the progression of activities starts with the pool table. How you define the pool table establishes how the balls on the table can behave, for how the balls behave is defined using physical parameters that relate the balls and the table.

Defining the Table

With a pool game involving the functions investigated thus far, a number of restrictions become apparent. For example, in the discussion so far, 2-D collisions have been dealt with. To accommodate this restriction, consider first how you can depict a pool table. A horizontal table viewed from a top angle is the best. As shown in Figure 11.1, the table is a rectangle, and while a number of ways exist for defining a table, the simplest is to split it into six primary elements, each of which is a line segment. The pockets of the table are the edges of the rectangle not covered by the line segments.



Figure 11.1
Defining the pool table.

As for the pool balls, they collide with the walls of the table or with each other, and given the discussion so far, their collisions can be regarded as elastic. Likewise, because the table is horizontal, gravity is not an issue. Other than friction, there is no acceleration on the balls except when struck by the cue.

With respect to how the balls fall into the pockets, consider first that at the mouth of each pocket, as illustrated by Figure 11.2, each of the walls curves around in a quarter-circle. The quarter circles are represented by the perimeters of the unfilled circles shown in the figure. Since the walls connect with the insides of the two circles relative to the running surface of the pool table, most of the circumferences of the two circles remain invisible to the collision-detection activity. Given the way the walls are positioned, after a ball (represented by the filled circle) passes these circles, it can be assessed as having dropped into the pocket. Since there are six lines forming the wall of the table and 12 circles forming the “jaws” of the pockets, there are 18 possible collisions between a ball and the table.

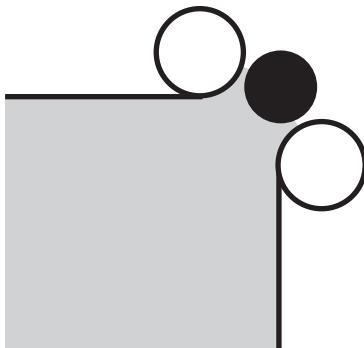


Figure 11.2

Detail of the pocket mouth.

When considering the dimensions of the table and the sizes of the pockets, four topics stand out. First is the size of the table. Second is the jaws as they curve at the pockets. Third is the gap between the jaws. Fourth is the size of the ball. With respect to dimensions, the table is a rectangle, and since it is twice as long as it is wide, it is essentially two squares. With respect to openings between the jaws at the pockets, while these must be a little wider than the radius of the ball, it is important not to make them too much wider. To fine-tune this adjustment, consider that the circles of the jaws can be increased or decreased in radius. In the version introduced here, the radius of the jaws is set to half the

radius of the ball. As for the mouths of the pockets, a good start is to set them to 1.7 times the width of the ball. This means that the gap through which the ball can pass is 1.2 times the width of the ball, which is a little wider than those found on real pool tables.

As much work as you might put into depicting the table visually, when defining the function you apply to it, it is important that all properties pertaining to the pool table, the pockets, the wall, and the size of the balls be represented by generic variables. Given this approach, among other things, if you want to vary the parameters, making the game more or less difficult, you don't have to go through the code to change hard-coded values.

The `defineTable()` function sets the primary properties of the pool table as defined so far. The function takes as arguments the sizes of the balls, the table, the pockets, and the jaws of the pockets. As used in a game, this function is likely to be a little more complicated than shown. For example, to draw the table walls, you might want to offset the table by a small amount. In this version, likewise, you define the top left corner of the table as the point $(0,0)$.

```
function defineTable(ballRadius, tableSize, pocketSize, jawSize)
  set rs to ballRadius*pocketSize
  set rd to ballRadius*sqrt(2.0)*pocketSize // just for convenience
  set walls to an empty array
  append these arrays to walls:
    ((rd,0), (tableSize-rd*2,0)) // top
    ((0,rd), (0,tableSize-rd-rs)) // left top
    ((0,tableSize+rs), (0,tableSize-rd-rs)) // left bottom
    ((rd,tableSize*2), (tableSize-rd*2,0)) // bottom
    ((tableSize,rd), (0,tableSize-rd-rs)) // right top
    ((tableSize,tableSize+rs), (0,tableSize-rd-rs)) // right bottom
  set pw to ballRadius*jawSize
  set jaws to an empty array
  // you now use walls as a guide to draw jaws
  repeat for wall in walls
    if wall[2][1]=0 then // this is a vertical wall
      if wall[1][1]>psize/2 then // it's on the right
        append (wall[1]+(pw,0)) to jaws
        append (wall[1]+wall[2]+(pw,0)) to jaws
      else // it's on the left
        append (wall[1]-(pw,0)) to jaws
        append (wall[1]+wall[2]-(pw,0)) to jaws
    end if
```

```

else // this is a horizontal wall
  if wl[1][2]>psize then // it's on the bottom
    append (wall[1]+(0,pw)) to jaws
    append (wall[1]+wall[2]+(0,pw)) to jaws
  else // it's on the top
    append (wall[1]-(0,pw)) to jaws
    append (wall[1]+wall[2]-(0,pw)) to jaws
  end if
end if
end repeat
return array(walls,jaws)
end function

```

Defining the Balls

The surface of a pool table is divided by baulk lines. In a real pool game, when preparing to break the racked balls, a player may place the cue ball anywhere along the baulk line at the end opposite the racked balls. This baulk line is $\frac{1}{5}$ of the way down the table. In the version of the game depicted here, for simplicity, you always place the cue ball on the same spot on the baulk line.

The balls other than the cue ball are racked in a triangle as shown in Figure 11.3, with the front ball $\frac{3}{4}$ of the way down the table. The figure has been supplemented with a right triangle representing distances using ratios of the radius of the ball to show how the configuration might be recreated mathematically.

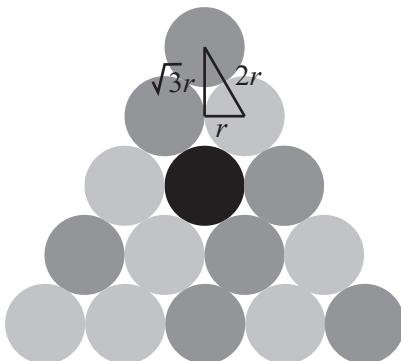


Figure 11.3

The initial configuration of balls.

In a pool game, you use one cue ball and 15 racked balls. To represent the balls, you employ an array with 16 elements. Each element of the array contains information about each ball. For now, this information is comprised of four pieces of data: the ball's position, direction, speed, and color. Since doing so is a useful practice, you keep the direction and speed of the balls separate. Likewise, to represent the direction of the ball, you employ a unit vector. When the speed is zero, the direction is arbitrary.

When placing the information about the balls in the array, you use the information about the cue ball in the first element. The other balls are placed in order after that, with the black last. The order of the colors is "cue," "red," "yellow," or "black." The cue ball is usually a white ball.

When setting up the balls in the rack, it is important to marginally separate the balls from each other. If they touch exactly, you might find that rounding performed by the computer will make it appear to the computer that they overlap. The `createBalls()` function puts the previous discussion to work to set up the balls on the pool table. Only two parameters are needed, the size of the table and the radius of the balls.

```
function createBalls(tableSize, ballRadius)
  set r to ballRadius*1.02
  set cuestart to (tableSize/2,2*tableSize/5)
  set y to sqrt(3.0)*r
  set tristart to (tableSize/2,tableSize*3/2)
  set balls to an empty array
  repeat for i=1 to 16
    if i=1 then
      set col to "cue"
    else if i=16 then
      set col to "black"
    else if (i mod 2)=0 then
      set col to "red"
    else
      set col to "yellow"
    end if
    if i is
      1: set p to cuestart
      2: set p to tristart
      3: set p to tristart +(-r,y)
      4: set p to tristart +(r,y)
      5: set p to tristart +(2*r,2*y)
      6: set p to tristart +(-2*r,2*y)
      7: set p to tristart +(-3*r,3*y)
```

```

8: set p to tristart +(-r,3*y)
9: set p to tristart +(r,3*y)
10: set p to tristart +(3*r,3*y)
11: set p to tristart +(4*r,4*y)
12: set p to tristart +(0,4*y)
13: set p to tristart +(2*r,4*y)
14: set p to tristart +(-4*r,4*y)
15: set p to tristart +(-2*r,4*y)
16: set p to tristart +(0,2*y)
end if
append array (p, 0,(1,0), col) to balls
end repeat
return balls
end function

```

While the `createBalls()` function serves well for a starter, eventually you require a few more properties for each ball. One is whether each ball has been pocketed. Another, possibly more useful for bookkeeping, is whether each ball is currently moving.

Note

Object-oriented programming makes organizing this information much easier. Consequently, in a real-life version of this game, if the programming language you are working with allows you to do so, it is far more sensible to create a class for the ball. Each ball could then keep track of its own state and appropriately respond to queries. However, in this context, to make the development of the primary algorithms clearer, a procedural approach is illustrated.

Defining the Physical Parameters

The final aspect of the simulation is to define the physical parameters that remain constant throughout the game. Many of these need to be determined through a certain amount of trial and error, but you can still make some approximations at the start.

Because the physics of pool is relatively simple, it turns out that many parameters can be factored out right from the start. For one thing, all the balls, including the cue ball, have the same mass, and they collide almost entirely elastically. As a result, you can use the `resolveCollisionEqualMass()` function from Chapter 9. Recall that this function ignores mass and efficiency.

The walls of the pool table have cushions, and this affects the way the balls collide with them. For one thing, collisions of balls with cushions are somewhat less efficient than collisions between balls. One outcome is that you cannot adequately resolve collisions between balls and walls elastically.

Apart from situations in which balls collide with cushions, another area where energy is lost is in friction. Realistically dealing with friction is complicated and has a significant effect on the play of the game. For one thing, without friction, balls will not roll. To cut down on the complexity that accompanies a comprehensive physics engine for friction, you can use a model involving “game friction.” There are various ways this can be done, yielding more or less realistic motion:

- Reduce the energy of the simulation by a constant amount each second.
- Decrease the energy by a constant factor each second.
- Decrease the energy by a factor that varies according to the current energy.

You might want to think about which of these is most likely to work. The first yields a speed that decreases linearly (constant deceleration). The second decreases exponentially (deceleration varying according to speed). The third is somewhere between the first two. While the last is going to appear the most realistic, in practice, it’s hard to distinguish from the first. The worst is the second option. With exponential decrease, the ball quickly slows down but then takes a long time to come to a complete rest. In reality, an object moving slowly comes quickly to a halt.

If you employ the third approach, involving variable speed, one final parameter might be useful. This parameter has no corresponding physical reality, but it is convenient to work with. This parameter is for “deactivation,” and it establishes a minimum speed below which a ball is deemed to have stopped. With constant deceleration (given by the first two options), this parameter is not needed, for all balls are guaranteed to reach zero speed in reasonable time.

Given the discussion thus far, a complete set of parameters for the game can be presented as follows:

- Table width
- Ball radius
- Pocket size
- Deceleration due to friction
- Efficiency of collision with cushions

The values of these parameters depend on how you want the game to play. Any value for them will yield a game that feels realistic to some degree. In reality, the value of the pocket size is determined precisely by the width of the table, but to make the game more or less difficult, as mentioned previously, you can change this value. The physical parameters have more to do with the game “feel” than difficulty or realism. They affect, for example, how long players will have to wait between shots and how fast the action is. Also, the efficiency of collisions with cushions affects the difficulty of getting out of a snooker. A *snooker* is a situation in which, to make a legal move, a player must bounce the cue ball off of a cushion.

Taking a Shot

Having set up the framework of the game, you can start things moving. In this section, you look at how the user can start the game. The first action is to initiate the movement of the cue ball. After that, you must consider the main game loop and how a turn begins. Finally, it is essential to attend to simulating a single shot.

Creating the Cue

How the user interacts with the game has little to do with the physics or mathematics of the game. For this reason, in this context of discussion, your primary concern begins with the cue ball and its initial momentum. At the start of each turn, the cue appears at the position of the cue ball. In a common scenario, the cue is seen with its point resting on the outside of the ball. It rotates to follow the mouse cursor so that it lies along the line from the mouse cursor to the center of the ball.

When the mouse button is pressed, the cue begins to pull back. When the mouse button is released, the cue strikes the cue ball. The initial momentum of the cue ball is proportional to the length of time the mouse button is held down. If the mouse button is held longer than a set time, the cue automatically strikes the cue ball at full power.

The `cueRotation()` function follows this scenario. To create this function, you begin with the assumptions that you know the position of the cue ball and that you can read off the position of the mouse cursor at any particular time. Given this information, you can determine the vector between the cue ball and the mouse cursor. Converting this vector into an angle allows you to set the rotation of the cue. The definition of the function also depends on the default rotation of the cue, which is assumed to be horizontal, with its point to the right. The function then determines the rotation at any one moment. Here is how the `cueRotation()` function is constructed:

```
function cueRotation(ballPos)
    set v to ballPos - (the current mouse position)
    if magnitude(v)>0 then
        set ang to atan(v[2],v[1])
        return ang*180/pi
    otherwise
        return "error"
    end if
end function
```

Note

The precise details of how the cue is drawn depend on your development platform. In some instances, this might involve setting the rotation property of the cue sprite.

With the cueRotation() function, you have a potential problem when the mouse is exactly over the ball position. There are various ways to resolve this. One is to set the rotation to zero at that moment. A problem with this approach is that the cue can appear to flicker as it passes over the ball. To remedy this, you can remember the last rotation and leave it unchanged at that instant. When the mouse is over the cue ball at the instant the cue first appears, there is no previous rotation, and at this point, you can then set the value to zero by default.

When the mouse button is pressed, you can fix the current vector. For simplicity, you normalize it so that you always have a unit vector for the direction. This direction translates directly to the initial direction of the ball. Now all that is left is to determine the speed.

To determine the speed, pulling the cue back is simple enough. You keep track of the time when the mouse button is pressed. Then, at each time-step, you know how long it has been down. To give the new position of the cue relative to the cue ball, this value can be multiplied by the reverse of the cue vector (along with a constant scale factor). If the time is greater than the maximum time allowed, you strike the ball automatically; otherwise, you go on to the next time-step.

When the mouse is released or the maximum time has elapsed, you set the speed of the cue ball proportionally. The simplest way to do this is to fix a maximum initial speed as mentioned previously. Then the speed of the cue ball is arrived at by the proportion of the cueing time to the maximum time and multiplying this value by the maximum speed. Visually, while you can animate the cue moving in to strike the ball, in practice it works just as well to place the cue immediately at the ball position again. You then leave it there for a few moments before hiding it and showing the result.

The Main Game Loop

With the cue ball on its way, you now start to run your physics simulation. With all the work you have done already, this is surprisingly simple. It involves only plugging the appropriate values into the functions already created in previous chapters. The following list provides a summary of the calculations you perform using these functions for each time-step:

1. **Determine time elapsed.** Calculate the time elapsed since the last time-step.
2. **Apply friction.** For each moving ball, apply friction by decreasing the speed of the ball by the fixed friction value. If the speed reaches zero, mark the ball as no longer moving.
3. **Attend to potential collisions.** For each still-moving ball, loop through all the other balls, as well as the cushions and pocket corners, to see whether there is a potential collision during this time period.
4. **Attend to no collisions.** If there is no collision, then set all the balls to their new position and finish.
5. **Attend to the first collision.** Find the first collision. While detecting collisions, keep track of the time at which it happens (as a proportion of the total time), as well as the collision normal. These values are returned by the collision-detection function.
6. **Ascertain collision time.** To get the collision time, multiply the time proportion by the total time. By multiplying the collision time by the product of their speed and direction vector and adding it to their current position, you set all the balls to their positions at this time.
7. **Resolve collisions.** Using the `resolveInelasticCollisionFixed()` function, resolve collisions between balls and cushions. For collisions between two balls, use the `resolveCollisionEqualMass()` function. Set the new velocity of the colliding ball(s) accordingly.
8. **Repeat.** Decrease the total time by the collision time to get the remaining time for this time-step. Repeat from step 3.

The `moveBalls()` function provides an example of how to implement the actions given in steps 1 through 8. It is the longest function in this book. While in practice you are likely to refactor it into several functions, as with other code examples given in this book, it is left in one piece to make its actions, relative to description, easier to follow. While it is nearly identical to the `checkCollision()` function, it has been optimized in a few places.

```
function moveBalls(t, table, cushions, pockets, balls, r, f, e)
  // apply friction and deactivate where appropriate
  set mv to 0
  repeat for each b in balls
    if b is moving then
      set the speed of b to max(the speed of b-f,0)
      if the speed of b is 0 then
        set b to not moving
      otherwise
        set mv to 1 // there is some ball moving
      end if
    end if
  end repeat
  if mv is 0 then return "stopped"

  // check for collisions
  repeat while t>0
    set mn to 2
    // mn is going to be the minimum time proportion
    set ob1 to 0
    set ob2 to 0
    repeat with i=1 to the number of balls
      set b1 to balls[i]
      set pos1 to the position of b1
      set v to t*the velocity of b1
      // its speed * its direction vector
      // check for collisions between balls
      repeat with j=i+1 to the number of balls
        set b2 to balls[j]
        if b1 or b2 is moving then
          set pos2 to the position of b2
          set u to t*the velocity of b2
          if possibleCollision(pos1, v, pos2, u) then
            set c to
              circleCircleCollision(pos1, v, r, pos2, u, r)
            if c is not a collision then next repeat
            set tm to the time of c
            set m to min(mn,tm)
```

```
if m<mn then
    set mn to m
    set n to the normal of c
    set ob1 to b1
    set ob2 to b2
    set tp to "ball"
    end if
end if
end if
end repeat
// check for collisions with cushions
if b1 is moving then
    repeat for each w in table
        // table is an array of two-element arrays,
        // as defined in defineTable()
        set c to circleLineCollision(r, pos1, v, w[1], w[2])
        if c is not a collision then next repeat
        || check if it is minimal just as before;
        || if so, set n, tm and ob1, and set tp to "wall"

    end repeat
    repeat with p in cushions[1]
        // cushions is a two-element array representing the
        // pocket entrances, the first element being
        // a list of circle centers, the second being
        // the radius of the circles.

        if possiblecollision(pos1, v, p, (0,0)) then
            set c to
            circleCircleCollision(pos1, v, r, p, (0,0), cushions[2])
            if c is not a collision then next repeat
            || again, check if minimal, and if so
            || set n, tm and ob1, and tp= "wall"
        end if

        end repeat
    end if
end repeat
```

```
if mn=2 then exit repeat // no collision

// otherwise there is a collision

// move balls to collision position
repeat for each b in balls
    if b is moving set its position to
        its position + mn*t*its velocity
end repeat
// resolve collision
if tp=#wall then
    set u to the direction of ob1
    set the direction of ob1 to
        resolveInelasticCollisionFixed(u, e, e, n)
otherwise
    set u1 to the velocity of ob1
    set u2 to the velocity of ob2
    set res to resolveCollisionEqualMass(u1, u2, n)
    set the speed of ob1 to magnitude(res[1])
    if this speed>0 then
        set the direction of ob1 to norm(res[1])
        and set ob1 to be moving
    otherwise set ob1 to be not moving
    || repeat for ob2

end if

// decrease time and repeat
set t to t*(1-mn)
end repeat

// move balls for the last section (no collisions)
repeat for each b in balls
    if b is moving set its position to its position + t*its velocity
end repeat
end function
```

One element that has been left out of the `moveBalls()` function is determining if a ball has been pocketed. There are a number of ways to achieve this. Exercise 11.1 challenges you to give some thought to this.

Basic Culling

Culling is a general term for removing or ignoring elements that are not needed in a particular set of circumstances. For example, in a 3-D model, you use it to describe the process by which you determine which parts of the model can be seen and which can't. This allows you to draw the visible faces and so save time.

In a physics simulation like this, you can use the word “culling” to mean the process of pre-checking collisions before actually testing them. The result is that you cut down the number of collision checks you perform. While you will look at some of these methods now, an initial warning is necessary. Culling is not always worth the effort. If your culling checks take longer to perform than the collision detection itself, it's best to ignore them. In this case, however, you are dealing with colliding circles and straight lines. These are fairly straightforward to check, especially since there aren't so many of them. As it is, when there are few collisions to check, the situation is improved. When large numbers of balls are involved, no amount of culling will help.

One of the most common culling techniques involves partitioning the game world into subspaces. For example, suppose that the current table state in your pool game is as shown in Figure 11.4. Here, the table has been subdivided into eight squares. Ball A is moving toward balls B, C and D, but because A and B are in non-contiguous squares, they cannot possibly collide within a short time period. This means that for ball A, you need only to check for collisions with balls C and D. If a ball is in a particular square, it can only collide with balls within its square and the three squares that its velocity vector could conceivably intersect.

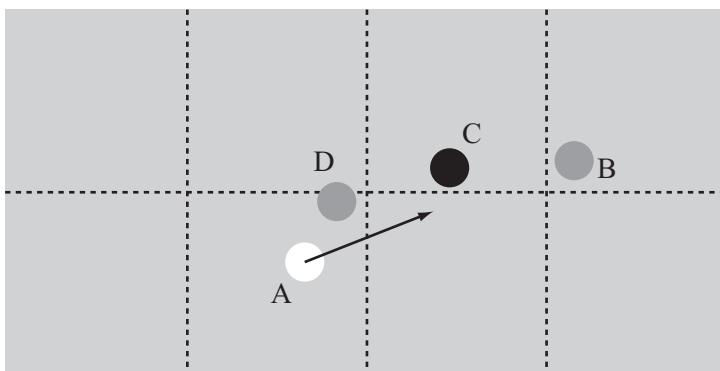


Figure 11.4

Partitioning the table into contiguous subspaces.

Another partitioning technique entails using overlapping subspaces, as shown in Figure 11.5. In this system, a ball is always within several squares at once, which means that whatever its velocity, you can usually find a subspace that the ball remains within for the whole of its motion. Then you need only to check for collisions within the space. This method is essentially equivalent with the first; it moves the same calculations to different parts of the process. Which approach you use is a matter of personal preference.

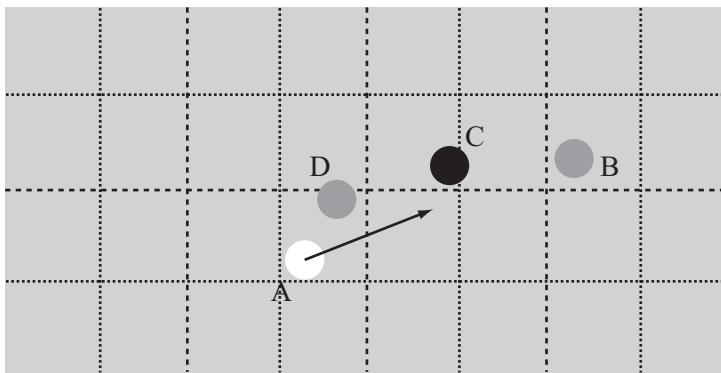


Figure 11.5

Partitioning the table into overlapping subspaces.

It is important in both of the culling approaches shown previously to ensure that the squares are large enough that no ball will pass completely through a square in one turn. This can be done by setting a maximum speed, but you can also set a minimum time-step for the simulation. This is necessary because, for various reasons, any simulation might be interrupted while playing, leading to an extremely long gap between collision checks. If you want to have a maximum speed of m pixels/sec and a square size of s pixels, then the maximum time-step you can allow is $\frac{s}{m}$. Actually, even this isn't quite safe because the balls have a non-zero radius. A ball in one square can collide with a ball in a neighboring square. Given this possibility, you should limit the time-steps to $\frac{(s-2r)}{m}$, where r is the radius of a ball.

Game Logic

The final stage of creating the game is to add the game logic. Game logic, among other things, involves determining fouls, the order of play, and whether a player has won or lost. While most such elements of logic are irrelevant to the mathematics, it's worth looking at the two main changes that need to be made to the mathematical system to turn it into a game.

1. **Determine if the correct ball (if any) has been hit in a particular shot.** Several foul shots are dependent on the first ball hit during a turn. The `moveBalls()` function can be adapted to return the color of the first ball struck in that time period, which can be used to determine the first ball struck since cueing. If the color is incorrect, or if no ball is struck, this is a foul.
2. **Deal with pocketed balls.** A pocketed ball is removed from play. The `moveBalls()` function can take this into account by ignoring pocketed balls in collision checks. At the end of each turn, newly pocketed balls need to be checked for their color to see if it was a legal pocket, resulting in a second shot for the current player, or if an illegal one, resulting in a foul. If the black was pocketed, this is always either a win or a loss for the current player.

Exercises

EXERCISE 11.1

Amend the `moveBalls()` function to include a check for whether a ball has been pocketed. There are several ways to develop this test. No one stands out as the best.

EXERCISE 11.2

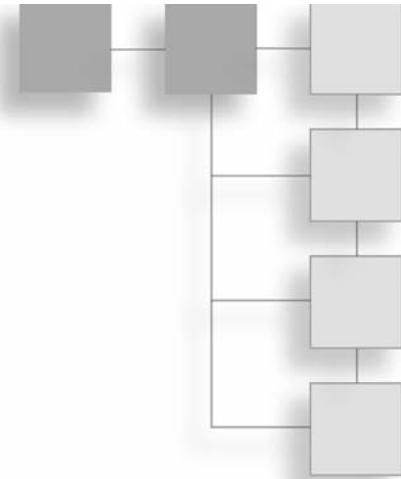
Add a “preview” function that determines which ball will be struck first and in which direction it will move. Make the output display where the cue is being aimed.

Summary

With this chapter, you have had a chance to see how the functions developed in previous chapters can be combined to accomplish tasks in any number of contexts. The pool game is just one of many possibilities. You will return to this game several times later in the book. In Part III, you will look at more complex physics that can be added to the pool game. This includes friction and spin.

You Should Now Know

- How to use the collision-detection and resolution functions developed in previous chapters in a real-life game situation
- How to simplify the physics to deal with the specific needs of a game
- The basics of culling, particularly how to partition a space to reduce calculations



PART III

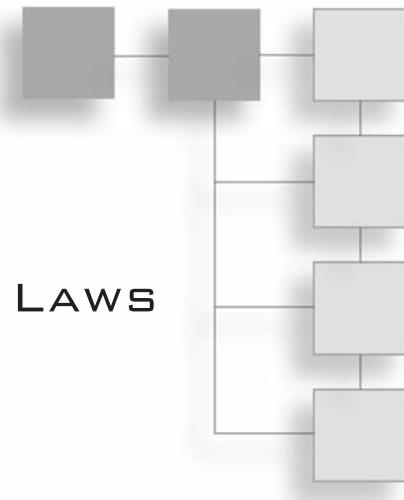
COMPLEX MOTION

In Part III of the book, you will continue with mechanics, looking at more difficult concepts like angular motion, springs and pendulums, and orbits. You will begin by formally looking at the concept of force, which you have skirted around up to now. Then you will move through Newtonian physics, simple harmonic motion, friction, and other topics, before finishing with a quick excursion into the slightly complex mathematics of Bezier curves.

This page intentionally left blank

CHAPTER 12

FORCE AND NEWTON'S LAWS



In This Chapter

- Overview
- Force
- Gravity
- Rockets and Satellites

Overview

Although the word *force* has been used quite a few times in the preceding pages, as of yet, it has not been precisely defined. Force is a term that is precisely defined in physics, and the study of physics begins with the laws of motion developed by Isaac Newton (1643–1727). Newton's laws respecting motion and gravitation were presented as universal scientific statements. They have remained valid for hundreds of years, and while it is generally known since the time of Albert Einstein (1879–1955) that they are limited according to the scales of space and time in which they are applied, in everyday use, they are still remarkably accurate.

Even if it provides precise definitions of force, in other respects, this chapter is something of a recapitulation of previous chapters. Among other things, it re-examines terms previously present in the context afforded by the mathematics alone. In the current contexts, you see mathematical techniques put to work by the science of physics.

Force

The term *force* is used commonly in everyday speech. People refer to the force of an argument, the force of nature, or the force of law. All such uses are both effective for the contexts of their use and highly metaphorical. Such uses suggest an effort or power, and the words “energy” and “work” are used almost interchangeably. In a scientific context, force is a technical term. Newton defined the concept of force precisely with his three Laws of Motion. Force is measured with a unit named after Newton. This is the *newton*. A newton is defined as one kilogram meter per second per second.

The First Law

How things move has not always been understood the same way. Before Newton, it was thought that every moving object required some impetus to keep it moving along its path. Unless something is being pushed, it was contended, it will slow down. Among many others, this was a notion introduced by Aristotle (383 B.C.E.–321 B.C.E.). This is not an unreasonable assumption. After all, an automobile stops moving when the engine stops. If you move a toy automobile with your hand, it stops moving if you remove your hand. Even if the automobile has been speeded up or pushed to a high speed, due to friction, it slows and stops after a time. It is only in a setting without friction that Newton's observations become clear. One such setting is outer space, where there is very little friction.

Having given consideration to how objects move in outer space and elsewhere, Newton formulated his first law. According to this law, *an object continues moving at the same velocity unless acted on by a force*. This movement includes a velocity of zero. An object that is standing still will remain standing still unless a force is applied to it to set it in motion.

What form this force takes is not stated in Newton's first law, but for the moment you can think of it as some kind of *influence*. A particle will continue to travel in the same way unless influenced by some other particle.

As illustrated by Figure 12.1, a body with no forces acting on it is said to be in *equilibrium*. The sum of the forces acting on it is zero. In this respect, Newton's first law can be stated as “a body in equilibrium remains at a constant velocity.”

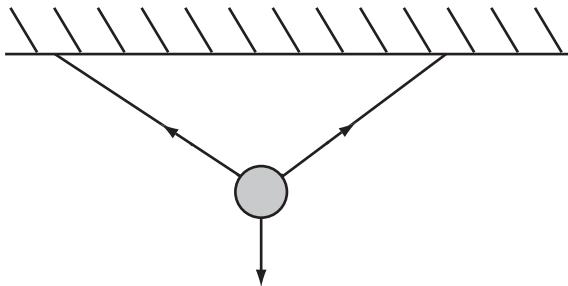


Figure 12.1

A body in equilibrium with several forces acting on it.

In essence, the Newton's first law is an early version of a later law of physics relating to conservation of momentum. You implicitly used this latter law when you assumed that the pool balls in the previous chapter would continue to move with the same velocity unless acted on by some friction or colliding with another ball.

The Second Law

Newton presented a second law that is more mathematically involved than the first. While Newton stated the second law in complex terms, stripped to its essentials, it says, *When a body is acted on by a force, it experiences an acceleration that is proportional to the force and inversely proportional to the mass of the body*. Using mathematics, you can formulate this as follows:

$$\text{Force on body} = \text{mass} \times \text{acceleration}$$

An alternative way to look at Newton's second law is to say that the force on a body is equal to the rate of change of the body's momentum. This is more useful when dealing with forces or masses that change continuously over time, such as centripetal force or the motion of a rocket. Both of these are covered later in this chapter.

Newton's second law defines the unit of force. It specifies that one unit of force is the force required to accelerate a kilogram mass by 1 ms^{-2} in one second. As mentioned previously, the newton is the name applied to this force. The abbreviation for one newton is 1N. As you have seen in previous chapters, the acceleration due to gravity at sea level is approximately 10 ms^{-2} . This means that the force of gravity at sea level experienced by a 1-kg mass is approximately 10N. When you feel this force on your body, you feel your *weight*.

Unlike the equations of motion examined in previous chapters, force is intimately tied in with mass. If you want to throw a 10-kg cannonball, it will take twice the force it would take to throw a 5-kg cannonball. This is where force is related to momentum and the concept of mass can be understood as *inertia*. Inertia is how much a body resists the action of any force.

The Third Law

How is it that things stand still? By Newton's second law, if you're experiencing a constant downward force, you should be accelerating downward. If this does not happen, the reason is that you are in equilibrium. Although there is a force acting down on you, there is another force acting upward. For example, if you are sitting in a chair, then force is acting through the chair you are sitting in to prevent you from going farther down. In turn, the chair is experiencing a downward force from you, but it doesn't move because it is experiencing an upward force from the floor. That this is so is implied by the first two laws Newton presented. It is formalized in Newton's third law, which is the most misunderstood of the three. *If a body is experiencing a force from some other object, then it exerts an equal and opposite force on that object.*

The third law has to be carefully stated because most people get it wrong. They say "every action has an equal and opposite reaction," but they assume that the two actions are on the same body, which is clearly false. What Newton's third law actually says is that force is symmetrical. One object can't exert a force without experiencing a force. Earth pulls you toward itself by the force of gravity, but simultaneously you are pulling Earth slightly toward you.

You don't notice these effects while on Earth's surface because, with the huge discrepancy between your mass and Earth's, the acceleration of Earth due to your gravitational pull is infinitesimally small. What is more, there are many of these infinitesimal forces acting on Earth in all directions, so this tends to cancel their effects. Even the moon, which is much larger than a person, only barely pulls Earth off its orbit about the sun, although its gravitational pull on Earth's oceans does have a noticeable effect in the tides.

If Newton's first law is a statement of conservation of momentum for bodies in equilibrium, the third is the equivalent for bodies that are colliding. In fact, aside from conversion of energy into other forms such as heat, it also codifies the law relating to the conservation of energy. Ultimately, while Newton's first and third laws are different ways of viewing the methods you have already been using to deal with ballistics and collisions, Newton's second law is a little different. It is a mathematical result that allows you to make calculations about motion. While you will be using the first and third laws from time to time, it is the second law that will concern you most in this and later chapters.

Impulse

You might have recognized by this point that Newton's second law is going to encounter problems when dealing with the rigid body collisions you looked at in Part II. With these collisions, you consider that the velocities of colliding bodies change instantaneously at the moment of collision. This implies an infinite acceleration, which requires an infinite force. However, if you have infinite force and acceleration, the velocity at the end of it becomes impossible to calculate in any especially useful way in the context of collision detection.

To deal with this problem, you introduce the concept of an *impulse*, which is defined as a force acting over a particular time period. It is equal to *force* \times *time*. Newton's second law tells you that this impulse is going to be equal to the change in momentum. In a rigid-body collision, an infinitely large force acts over an infinitely short period of time, and these two infinities cancel out to give a finite impulse. As you saw in Chapter 9, using the laws of conservation of energy and momentum, such an impulse is readily calculable.

Note

Recall that there is not really any such thing as a rigid body. Colliding bodies deform as they collide. They also rebound from each other. Since deformation and rebounding require some amount of time to take place in, real collisions always last for some measurable time.

Using these generalizations about impulses, you can rewrite your collision resolution calculations. When two bodies collide, they experience some impulse J in the direction of the collision normal \mathbf{n} . This causes a change of momentum that is some multiple of \mathbf{n} . You can summarize the event as follows:

$$m\mathbf{v} = m\mathbf{u} + J\mathbf{n}$$

The impulse acts equally and oppositely, so one body experiences J and the other $-J$. Combining these momentum equations with the energy equations, you can calculate the value of J , which turns out to be

$$J = \frac{-m_1 m_2 \mathbf{u} \cdot \mathbf{n}}{(m_1 + m_2)}$$

The resulting equation assumes \mathbf{n} is unit length, where \mathbf{u} is the relative velocity of the two bodies before collision. You might want to see if you can derive this result and make it match with the results of Chapter 9.

Gravity

In this section, you will look at the force of gravity and how it can be used to explain planetary motion. This story begins with Newton. In addition to his three laws of motion, another discovery of Newton's was the concept of gravity. While other astronomers, particularly Johannes Kepler (1571–1630), had observed and recorded data that described the motions of the planets, it was Newton who realized that the same principle that causes an apple to fall to the ground could be used to calculate the motions of the planets. It was a revolutionary concept. Accordingly, as Newton understood it, all the planets in the solar system, including Earth, are constantly “falling” toward the sun and each other.

The Law of Gravitation

The force of gravity is a universal phenomenon that affects all bodies from the smallest to the largest. Unlike the other “fundamental forces,” gravity is only attractive. Every physical body in the universe attracts every other physical body in the universe. This contrasts with other forces. For example, magnetic forces can attract or repel. In the current state of the universe, however, gravity is one way only.

Gravity works as an *inverse-square* relationship. The strength of the gravitational attraction between two objects is inversely proportional to the square of the distance between them. The phrase “the distance between them” means the distance between their centers of mass. Expressed as a formula, the relationship is presented as follows:

$$F = \frac{Gm_1m_2}{d^2}$$

Here, m_1 and m_2 are the masses of the two objects, d is the distance between them, and G is a constant, known as the *gravitational constant*, the value of which turns out to be approximately $6.673000 \times 10^{-11} \text{ m}^3 \text{ kg}^{-1} \text{ s}^{-2}$.

Given that gravity works at all scales, you might wonder why you don't have to take into account the gravitational attraction between all the molecules in each body as well. Fortunately, it turns out that these effects cancel out for a spherical body, and as mentioned earlier, you can treat most bodies as being particles at the center of mass.

The Motion of a Planet Under Gravity

Newton deduced the inverse-square relationship by examining the motion of the planets. As the basis of his research, he made use of data associated with Kepler's laws of planetary motion. By making painstaking and largely accurate observations, Kepler had replaced an understanding of the solar system developed by Nicolaus Copernicus (1473–1543). Copernicus thought the planets revolved around the sun at a constant rate in a circular path. Kepler said that view needed to be modified.

According to Kepler, the motion of a planet relative to the sun is elliptical, with the sun at one focus of the ellipse. Copernicus and others had mistaken the paths of the planets because the eccentricity of the planetary paths are so small. Likewise, Kepler pointed out that speeds of the planets as they travel around the sun are not constant. In fact, the speeds vary quite subtly. The area of the sector of the ellipse swept out during any time period by a planet is constant. Figure 12.2 illustrates how the areas of the two highlighted regions are the same. When the planet is nearer the sun, it moves faster than it does when far away. Figure 12.2 has been drawn in an exaggerated manner to make the effect clear; for planets, the deviation from a circle is minimal.

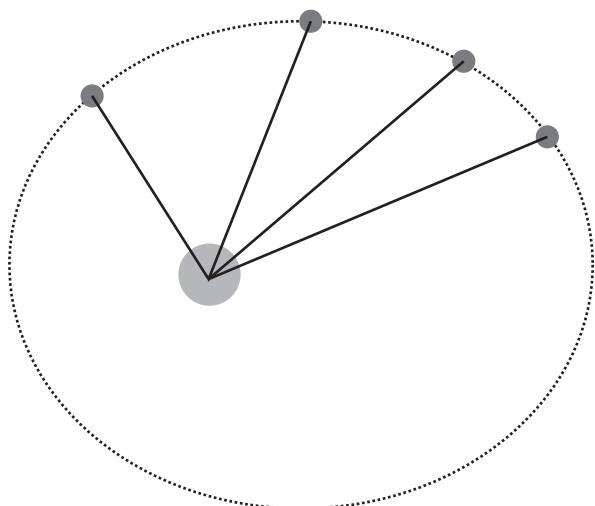


Figure 12.2

The speed of a planet at different stages of its orbit.

Kepler's observations apply to bodies other than planets. In fact, they work for all orbiting bodies, such as moons, comets, and asteroids. Not only that, but they work for meteorites that pass through the solar system without going into orbit. The only qualifier is that rather than moving in an ellipse, they move in a parabola. The law of equal areas continues to apply. Newton worked backward from these observations to realize that they implied an inverse-square law.

Stable Orbits

All the planets, including Earth, are spiraling slowly in toward the sun. Less stable orbits would have disappeared many billions of years ago and are unlikely to have been suitable for sustaining life. In addition to the Earth, the other existing planets are in nearly stable orbits. Again, however, perfect stability is impossible to achieve. Among other things, the gravitational pull of other planets slightly deflects each one from its orbit, destabilizing it.

The most important aspects of a stable orbit are its *period* and its semi-major axis. The period, denoted as T , is the time taken for one complete orbit. When one body has a significantly greater mass than the other, as with a planet in orbit around the sun, it turns out that these values are related as follows:

$$T = 2\pi \sqrt{\frac{a^3}{GM}}$$

where M is the total mass of the two bodies.

The inverse of the period is called the *angular frequency* of the orbit, which is 2π times the angular velocity, or the number of orbits in a given time. This gives you the equation of Kepler's third law:

$$\frac{2\pi}{T} = \sqrt{\frac{GM}{a^3}}$$

The value $\frac{2\pi}{T}$ is also called the *mean motion* n , and the initial equation can be rewritten in a simpler form as $n^2a^3 = GM$.

Another important result is that you can calculate the speed v of a particle when at a particular distance r from the center of the ellipse. This can be worked out by conservation of energy, and gives

$$v = \sqrt{GM\left(\frac{2}{r} - \frac{1}{a}\right)} = \sqrt{GM\left(\frac{2a-r}{ar}\right)}$$

Surprisingly, despite all these results, the question of finding the position of an orbiting body at a particular time t cannot be solved by simple algebra or calculus, although it is possible to find a differential equation that can be solved numerically.

Centrifugal and Centripetal Force

An object rotating in a circle appears to be disobeying Newton's first law. Since the direction of motion is changing, its velocity is constantly changing. However, this correspondence presents no great problem. To achieve the circular motion, there must always be a linear force on the object, directed toward the center of the circle. This force is called the *centripetal force* and can be calculated precisely. The centripetal force required to keep a mass m rotating at a constant speed v around a radius r is given by $\frac{mv^2}{r}$. In terms of the angular velocity ω , the force is $\omega^2 r$. This relationship will be examined more extensively in the next chapter.

Centripetal force should not be confused with *centrifugal force*, which is to some extent a myth. A particle moving in a circle is only experiencing a net force inward. However, by Newton's third law, the particle exerts a force in turn on whatever is causing it to spin. For example, if you whirl an object around on a string, then there is an outward force on the string equal and opposite to the centripetal force. This force on you is called centrifugal force. Similarly, when you whirl a bucket full of water around, the bucket exerts an inward force on the water, which in turn exerts an outward force on the bucket. It is easy to get confused about this phenomenon, and it's best to ignore centrifugal force altogether and concentrate on the centripetal.

The reason you seem to experience an outward force when moving in a circle is due to Newton's first law. Your tendency is to keep going in a straight line. It is this inertia that feels like an outward force. But if you imagine standing on a "wall-of-death" fairground ride, it is by being on the inside of the wall, with the wall exerting a force toward the center, that you are held in place. If you were on the outside, you would fly off the wheel. Further, you would fly off tangentially to the wall, not straight outward.

Rockets and Satellites

When working in Chapter 7 on ballistics problems, you assumed that the effect of gravity was a constant acceleration. Now you see that this is not true, and the force of gravity varies with height. Over small distances relative to the size of Earth, the difference is negligible, but once you start dealing with objects traveling into space, you have to take variable gravity into account.

Geo-Stationary Orbit

In 1945, over a decade before satellites were launched into space, science fiction writer Arthur C. Clarke realized that because the period of a stable orbit varies with distance, at some distance from Earth's surface, the period of an orbiting satellite will be exactly one day. This means that if the satellite orbits parallel to the equator, it will continuously remain over the same point on Earth's surface. Envisioning a network of such satellites used for telecommunications, Clark's speculations turned out to be right on the mark. In part inspired by his ideas, geo-stationary satellites are at the heart of telecommunications, surveillance, and GPS technology.

You can calculate the correct height of a geo-stationary satellite by noting that, as with any orbiting body, it must experience a centripetal force equal to the gravitational force. This gives you the following equation:

$$\frac{mv^2}{r} = \frac{mMG}{r^2}$$

If the satellite has a period of T , then you have $v = \frac{2\pi r}{T}$, so

$$\frac{4\pi^2 r}{T^2} = \frac{MG}{r^2}$$

$$r = \sqrt[3]{\frac{MGT^2}{4\pi^2}}$$

To this equation, you plug in values for Earth. Earth has a mass of about 6×10^{24} kg. The sidereal day for Earth is 86164 seconds. A sidereal day is the time taken for Earth to complete one rotation relative to a distant star. Earth's sidereal day is shorter than a solar day, which is the time to complete a rotation relative to the sun. Since Earth is orbiting around the sun, this adds a little extra relative spin. As a result, you get a value for r of 42168 km.

In current science fiction, one speculative prospect involves a space elevator, which is a long fiber connecting a geo-stationary satellite and its corresponding point on the ground. Once in place, such an elevator would vastly reduce the energy needed to get into space. As interesting as this prospect might be, the technical problems are immense. Among other things, the weight of the fiber alone would be immense, as well as the danger to and from aircraft. Likewise, although the satellite is in geo-stationary orbit, the connecting cable is not, and one result of this is that it acts as a drag on the satellite, tending to pull it out of orbit. Having said all this, many research groups are working on overcoming these obstacles.

A Really Fast Cannonball

The discussion thus far has not provided the tools to deal with rocket travel properly. The key to successful rocket travel is that the mass of the rocket decreases as its fuel is used up, and working comprehensively with an object whose mass changes over time must wait a few chapters. However, at this point, you can make a start. One approach is to consider a very fast cannonball. While such an object moves at great speed into space, its mass remains constant. How does such an object behave?

To investigate the behavior of the cannonball, consider first that the mass of the cannonball is much less than Earth's, so you can assume that Earth does not move significantly as a result of the gravitational pull of the cannonball. Likewise, assume the ball is fired vertically into space from sea level.

You can solve this problem of what happens after the cannonball is fired by means of energy considerations. At a distance x from the center of Earth, the cannonball has a gravitational potential energy of $\frac{GMm}{x}$. If its initial upward speed is u , then you know that $\frac{1}{2}mu^2 = \frac{1}{2}mv^2 + \frac{GMm}{x}$ at every instant of the motion. This gives you a differential equation:

$$\left(\frac{dx}{dt}\right)^2 = \frac{2GM}{x} - u^2$$

Differential equations are not, in general, easy to solve exactly in some algebraic form, and this is no exception. However, it does mean that given a particular initial position and speed for the cannonball, you can calculate its motion over time by incremental steps. The `moveCannonBall()` function is constructed to carry out this operation:

```
function moveCannonBall(currentHeight, initialSpeed, timePeriod, G, M)
    set currentSpeed to sqrt(2*G*M / currentHeight
                           - initialSpeed * initialSpeed)
    return currentSpeed * timePeriod
end function
```

Exercise

EXERCISE 12.1

Write a set of functions that will allow a system of planets to move under gravity.

You can do this a number of different ways, but the mathematics is the same. At each time-step, calculate the total force on each planet due to the gravitational fields of the others. Then convert this to a linear acceleration. Be careful to base your accelerations on positions at the beginning of the time step. Don't use moved positions. See if you can set up a planet to orbit smoothly.

Summary

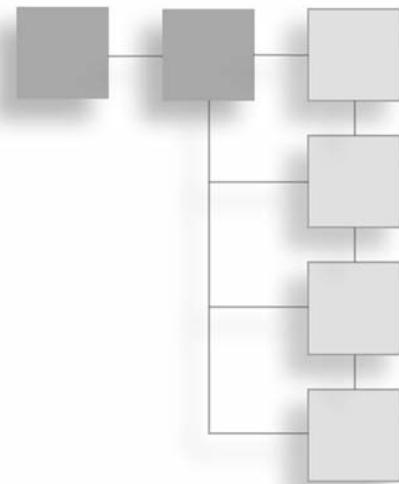
Little code has been presented in this chapter. The reason for the lack of code is that it has been necessary to restate earlier results in a new form. Given this new phase of understanding, however, you can now use the language of forces and Newton's laws much more extensively in your programming efforts. In the next chapter, you will continue the study of orbits by looking at angular motion and particularly angular momentum.

You Should Now Know

- Newton's three laws and how they relate to earlier results on energy and momentum
- The law of gravitation and how it can be used to calculate planetary motion
- How to calculate a geo-stationary orbit
- How to launch a cannonball into space and calculate its trajectory

CHAPTER 13

ANGULAR MOTION



In This Chapter

- Overview
- The Physics of a Lever
- Spin
- Spinning Collisions
- Spin Applied to the Pool Game

Overview

In previous chapters, you have looked at the linear motion of objects, but this is not the only way that something can move. In contrast, while a spinning top has no linear momentum, it is certainly moving. In this chapter, you will look at *angular motion*, or the physics of spin. You'll also see how it can be incorporated into the pool game discussed in Chapter 11.

The Physics of a Lever

Discussions of angular physics often begin with the lever. One of the simplest machines devised by humanity, the lever involves using a rod placed over a fulcrum or pivot. By adjusting the position of the fulcrum, you can lift heavy weights relatively easily. That you can employ a lever to lift weights of hundreds of pounds with little more effort than

is needed to move a small suitcase makes its powers seem endless. Archimedes (280 B.C.E.–211 B.C.E.) was one of the most celebrated mathematicians of ancient Greece and one of the earliest scholars to systematically investigate the lever. He became so impressed by the powers of the lever that he declared, “Give me a lever and a strong place to stand, and I can move the world.”

In many discussions of levers, the strengths of the rod and fulcrum are not taken into consideration. The assumption is that the strengths of these two parts of the lever allow them to support any weights applied to them. In reality, this is not so. Material strengths must be considered in real-world applications. A rod of insufficient strength bends, and a fulcrum lacking the capacity to support great weights crumbles or collapses. However, for purposes of exploration, ignoring the strengths of the materials used for the parts makes performing calculations much easier. In this chapter, the expression *light rod* is used to designate a rod that has no mass and does not bend or break, and unless otherwise indicated, the calculations assume this type of rod.

Torque

As illustrated by Figure 13.1, when a force is exerted on a part of a lever, it causes the lever to rotate. The downward force F on the left of the lever causes the lever to accelerate downward, but the fulcrum exerts a reaction force $-F$ upward. Because these two forces are acting on different parts of the lever, the result is that part of the lever moves downward, and part of it moves upward. This rotational force is called *torque*.

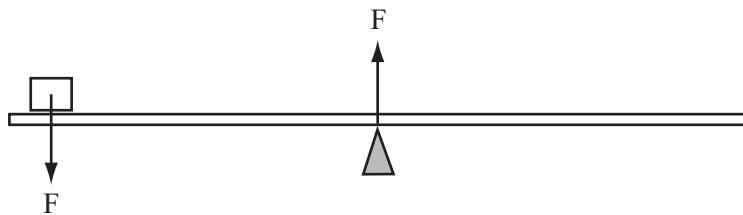


Figure 13.1

Torque exerted on a lever.

To understand how torque can be calculated, consider first that the force F is offset from the fulcrum. The fulcrum can only exert a force at the point of contact, so any force that acts somewhere other than the fulcrum causes rotation. In fact, the amount of rotation depends on how far the applied force is from the fulcrum. The formula for torque can be represented as follows:

$$\text{Torque} = \text{perpendicular force} \times \text{distance from fulcrum}$$

Expressed differently, the formula can be represented this way:

$$\text{Torque} = \text{force} \times \text{perpendicular distance from fulcrum}$$

The difference between these two ways of representing torque can be understood by considering Figure 13.2. While not worked out in this context, the two formulas are equivalent. (Demonstrating this involves using the dot products of the two formulas.)

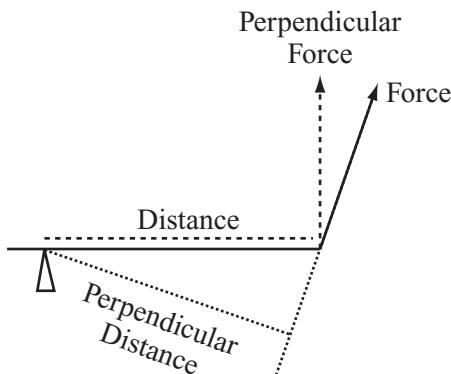


Figure 13.2
Calculating torque.

As shown in Figure 13.2, the distance from the fulcrum to the point at which you apply force is a vector quantity. If the force is applied at the other side of the fulcrum, the torque is reversed. As a result, you can envision this action as a “lever,” which can be described as a plane balanced on a pinpoint. A force can then be applied at any point on the plane. The corresponding torque is a function of the vector from the pinpoint to the position of the applied force. The product of a value with a perpendicular distance is called a *moment*. You can say, generally, that torque is the “moment of force.” Using the same reasoning, you can have a moment of velocity or a moment of momentum.

If two forces are applied to a lever, the corresponding torques can be added together. If the sum is zero, then the lever is in equilibrium and therefore balances. This means that you can balance a large force that is somewhere close to the fulcrum with a smaller force that is farther away from the fulcrum. Using this principle, you can then proceed to do such things as measure the weight of an object placed on the rod.

To calculate the weight of the object, suppose for starters that you have a lever with an object of unknown weight W resting on it, as shown in Figure 13.3. You know the distance x of the object from the fulcrum. If you take a second object of known weight A and move it along the lever until it balances at some distance y , you arrive at the following formula:

$$Wx - Ay = 0$$

$$W = \frac{Ay}{x}$$

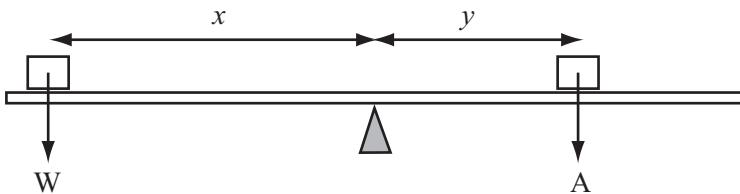


Figure 13.3

Two forces acting on a lever.

There is another point worth noticing here. Because neither object on the lever is moving, you can deduce that the forces on the objects are all in equilibrium. This means that there is a force of magnitude W acting upward on the first object. This force is the result of the torque exerted by the other object. This relationship can be made more explicit if you consider that any object resting on a lever experiences a force that is normal to the lever with its magnitude equal to the sum of torques on the lever caused by all other objects. You then take this value and divide it by the object's distance from the fulcrum. If you investigate the implications of this calculation, you see how a lever can be used as a catapult.

Specifically, suppose you have a lever resting on the ground with a ball placed on it at a distance x along the lever from the fulcrum. The ball has a weight of W . You place a heavy object of weight A on the other side of the lever at distance y . Immediately, the ball experiences a force due to the other object. Assuming the lever is nearly horizontal, the magnitude of the force is approximately $\frac{Ay}{x}$ and acts normal to the lever.

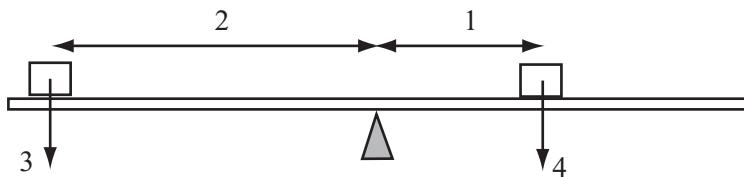
The result is that the ball accelerates perpendicularly to the lever, but simultaneously the lever itself rotates, so the force continues to act until the ball is traveling faster than the rotation of the lever, or until the lever stops moving because its other end hits the ground. At this point the ball flies off the lever.

Generally, as becomes apparent from the ratio of Ay and x , torque decreases as the component of weight tangential to the fulcrum becomes smaller. Other factors also enter into the work of the lever. Consider, for example, as mentioned previously, the weight of the rod and the possibility that it might break. The weight of the rod is not particularly troublesome. To deal with it, you can consider it one more torque, which can be measured exactly, as is done in Figure 13.3. With the possibility that the rod might break, the calculations become more difficult. Breaking depends on the strength of the rod's material when subjected to stress. With respect to levers, this stress is referred to as a *shearing force* or *bending moments*. Only if you are involved in extremely precise simulations do you need to concern yourself with such issues.

Moments of Inertia

The physics of rotation are much the same as for linear motion. Newton's laws and the equations of motion, momentum, and energy have their counterparts in rotational motion. For example, in units of radians (or degrees) per second, you can calculate an object's angular velocity or angular acceleration. For all rotational quantities, you need to specify the center of rotation and the axis around which the rotation occurs. There can be only one axis of rotation. If a body is rotating about the axis A and not moving linearly, then its angular velocity about any other axis is zero.

The angular equivalent of mass is called the *moment of inertia*. You've already encountered this measurement to some extent with the lever. In this case, however, the word "moment" is used differently because it includes a square factor of distance. The moment of inertia of a lever is the sum of the products of all masses with their squared distance from the fulcrum. In Figure 13.4, the moment of inertia of the lever around the fulcrum is $3 \times 4 + 4 \times 1 = 16 \text{ kgm}^2$.

**Figure 13.4**

Calculating a moment of inertia.

When dealing with a real object rather than a light rod, the calculation of the moment of inertia becomes more complex because you must employ integration to sum up infinitesimally small pieces. For example, consider what is necessary to calculate the moment of inertia of a uniform rod of a certain mass m and length $2l$ about its center. Because its mass is uniform, any segment of the rod with a length δ has a mass of $\frac{m\delta}{2l}$. The integral of the moment of inertia is calculated as follows:

$$\begin{aligned}
 I &= \int_{-l}^l x^2 \left(\frac{m}{2l} \right) dx \\
 &= \left(\frac{m}{2l} \right) \left[\frac{x^3}{3} \right]_{-l}^l \\
 &= \frac{ml^2}{3}
 \end{aligned}$$

For another example, consider a circle. You can think of a circle as a succession of infinitely thin concentric rings. Because these rings are uniform and every point on the ring is at the same distance from the center, the moment of inertia of each ring about an axis perpendicular to the circle through the center is $m_x x^2$, where m_x is the mass of the ring with radius x .

Consider a ring with a width of δ and assume that δ is infinitesimally small. At a radius x from the origin, for a circle of mass m and radius r , the mass of the ring is going to be approximately $\frac{2\pi x \delta m}{\pi r^2} = \frac{2x \delta m}{r^2}$. If you integrate over all of these rings, you end up with following equation:

$$\begin{aligned}
 I &= \int_0^r x^2 \times \frac{xm}{r^2} dx \\
 &= \frac{m}{r^2} \left[\frac{x^4}{4} \right]_0^r \\
 &= \frac{mr^2}{4}
 \end{aligned}$$

If the integral seems difficult to understand, for general purposes, keep in mind that in this context the focus is on using formulas rather than understanding how they are derived. In general, you are most interested in the moment of inertia about the center of mass of an object. The center of mass is the point at which, however you slice the object, half of its mass is on each side of the line.

A similar integration calculation can be used to calculate the center of mass. But you can calculate a moment of inertia about any axis. The calculation is not very involved. If the moment of inertia of an object with mass m about some axis A through the center of mass is I , then the moment of inertia about a parallel axis at a distance p from A is $I + mp^2$.

Inertial and Laminar Objects

Another type of object is the *laminar object*. Such an object can be described as an infinitely thin planar object like a disc or square. To calculate the inertia of such an object, as shown in Figure 13.5, consider first that you have two parallel axes in the plane of the lamina. These run through the same point O on the object. Given this start, the moment of inertia around the axis perpendicular to the laminar plane through O is the sum of the moments of inertia around the other two axes.

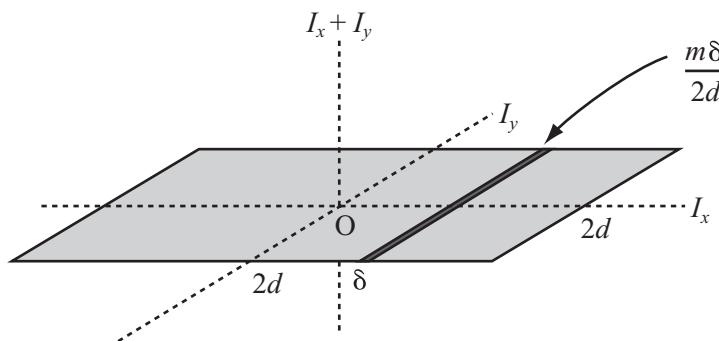


Figure 13.5

The moment of inertia of a square lamina.

You can use this understanding to calculate the moment of inertia of a square with side $2d$ around an axis perpendicular to the square through its center. Assuming the square is centered on the origin, using an argument similar to the one applied to calculations with the rod, the moment of inertia of the square around the y -axis is $\frac{md^2}{3}$. Since symmetry applies to this case, the same is true of its moment of inertia about the x -axis. The result is that its moment of inertia about the perpendicular axis is $\frac{2md^2}{3}$. What applies to laminar objects applies to many other simple shapes.

Just as mass is the measure of how hard it is to push something, the moment of inertia is a measure of how hard it is to spin something. An object with a small moment of inertia—an object whose mass is mostly very near to the axis—is easy to spin. An object whose mass is far from the axis, with a large moment of inertia, is much more difficult to spin, or for that matter to stop spinning. With respect to inertial generally, you can use a variant of Newton's second law to establish the following relation:

$$\text{Torque} = \text{moment of inertia} \times \text{angular acceleration}$$

Spin

In this section, further consideration is given to spin. Specifically, since you have dealt with the concept of a moment of inertia in place, you can now attend to calculating angular motion for spinning or rolling objects, such a flywheels.

Ballet Dancers and Spinning Tops

If you are working in two dimensions, the angular momentum of an object is its angular velocity multiplied by its moment of inertia. In three dimensions, the definition of angular momentum becomes more complex, for it is necessary to use objects called *tensors*. (This topic is dealt with briefly in Chapter 5.) As with linear momentum, angular momentum gives an idea of how difficult it is to stop something that is in motion. An object with a large moment of inertia has a large angular momentum, which means that it takes much more torque to stop it.

The fact that an object possessing large angular momentum requires more torque to be stopped explains the behavior of a flywheel. A flywheel is a large, heavy wheel that has a large moment of inertia and will spin for a long time. Most of the force opposing its motion comes from friction at the axle. Because this force occurs very near the axis of rotation, the corresponding torque is very small. This makes a flywheel an excellent way to store energy.

You can also calculate a moving object's angular momentum around some axis other than the axis of spin. As in other calculations, the best approach is to first consider the object as a particle. A particle traveling linearly in space still has an angular momentum about any axis, which is the moment of its linear momentum. As illustrated by Figure 13.6, to express this value, you can employ the relation $m|\mathbf{v}|d$, where d is the perpendicular distance of the particle's line of motion from the axis.

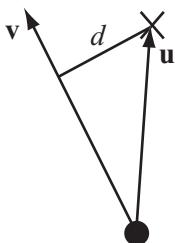


Figure 13.6

The angular momentum of a linear particle.

One complication of the relationship shown in Figure 13.6 is that you must think about whether the value calculated is positive or negative. In other words, it is necessary to consider whether the particle is moving clockwise or counterclockwise about the axis. One approach to a solution is to define the clockwise normal of a vector. You then use the dot product of this clockwise normal of \mathbf{v} with the vector \mathbf{u} from the particle to the axis, which gives $|\mathbf{v}|d$. The value is positive if and only if the particle is moving clockwise around the axis. The `moment()` function employs this approach to return the angular momentum.

```
function moment (position, vector, axisPosition)
    set n to clockwiseNormal(vector)
    return dotProduct(n, axisPosition-position)
end function
```

In addition, you can use the angular velocity to calculate the velocity of a point at a vector \mathbf{r} from the center. The value is equal to $\omega\mathbf{r}_N$, where \mathbf{r}_N is the clockwise normal of \mathbf{r} .

Like linear momentum, angular momentum is conserved, which means that if the moment of inertia of an object changes, its angular velocity must also change to compensate. The classic example of this is the ice skater or ballet dancer. Such dancers extend their arms horizontally, begin to spin, and then raise or lower them to a vertical position, decreasing their moment of inertia about the vertical axis while increasing the angular velocity of their spin.

Three-dimensional objects generally spin around the axis with the smallest moment of inertia, which is usually an axis of symmetry. If no other forces oppose them, they will move to orient this axis along the direction of spin. This is called the *gyroscopic effect*, and can be seen most obviously in spinning tops. Such motion can seem mysterious. Initially, the object can be moving slowly. When it rotates to orient along the axis of symmetry, it suddenly spins much faster. The change in spin is the retention of angular momentum. Since objects don't spontaneously speed up in linear motion, when it happens in angular motion, it's surprising. The gyroscopic effect is also used in guns. Bullets are "rifled" by spinning them through the barrel of the gun, which makes them more stable in flight since they tend to orient along the spin.

Rotational Kinetic Energy

The energy of a rotating object can be found by the same principle as linear kinetic energy. Rotational kinetic energy is equal to $\frac{1}{2} \times \text{moment of inertia} \times \text{angular velocity}^2$. If you check the units of this expression, you will see that they are the same as those for normal kinetic energy: kg ms^{-2} . Note that in this formulation, angular velocity should be measured in radians.

Because rotational kinetic energy is just one of the forms of energy, it can be readily converted to linear kinetic energy or gravitational potential energy. An example of this is the yo-yo, which spins as it falls. By controlling the rate of fall, and thus the momentum, it is possible to perform tricks. Consider when a yo-yo is made to drop to the ground and then spin in place before returning to the hand. Here, the gravitational potential energy at the top of the motion converts to kinetic energy and, as the yo-yo falls, to rotational kinetic energy. At the bottom, both the kinetic energy and the original gravitational potential energy have been converted to rotational kinetic energy—a small flick and the rotational kinetic energy is converted back into kinetic energy, allowing the yo-yo to climb.

It is interesting to note that a yo-yo falls more slowly than a ball falling through the air. Because some of the gravitational potential energy is converted to rotational kinetic energy, the total kinetic energy and thus linear speed must be less than it would be if there were no rotation. Similarly, ignoring the results of friction, a concrete pipe rolling down a slope accelerates more slowly than a block of ice sliding down it. A solid cylinder of the same size, with a larger moment of inertia, will move more slowly still. This gives another reason why you find it hard to believe Galileo's proof that all objects fall at the same speed. Larger objects do fall more slowly if they are free to rotate.

Spinning Collisions

When bodies can rotate, there is a potential for more complex collision behavior. Several factors must be taken into account. Among these are non-rotating bodies that collide at an angle. Such bodies might cause one another to rotate. Likewise, rotating bodies can collide laterally or along the leading edge. Angular momentum affects the results of collisions after the fact.

As with many of the more complicated collision types, angular collisions tend to be difficult to calculate algebraically. As a result, it is often necessary to use approximation methods. While there are far too many possibilities to cover in depth, hopefully the examples shown here will provide a starting point.

Detecting Collisions Between a Rotating Line and a Circle

As a first example of spinning collisions, consider something simple. Figure 13.7 illustrates a line segment pivoted on the point P situated at the end of the line. The line is rotating with an angular velocity of ω around P, starting from an initial angle of θ_0 from the vertical. In its path is a circle radius r , centered on the point Q at a distance d from P and at an angle of α from vertical.

With respect to the actions represented by Figure 13.7a, assume that the circle is far enough within the radius of the line segment that you do not need to consider the end point. You know that at any time t , the line has an angle of $\theta = \theta_0 + t\omega$. You also know that when touching the circle at the collision point R, the line forms a right-angled triangle PRQ, with $QR = r$ and the hypotenuse of length d . If the line is rotating clockwise, $\angle RPQ = \alpha - \theta$. If the line is rotating counterclockwise, $\angle RPQ = \theta - \alpha$. This gives you a single equation to determine the point of contact:

$$\alpha - \theta = k \sin^{-1} \left(\frac{r}{d} \right)$$

$$\theta_0 + t\omega = \alpha - k \sin^{-1} \left(\frac{r}{d} \right)$$

$$t = \frac{1}{\omega} \left(\alpha - \theta_0 - k \sin^{-1} \left(\frac{r}{d} \right) \right)$$

where k is 1 for clockwise motion and -1 otherwise.

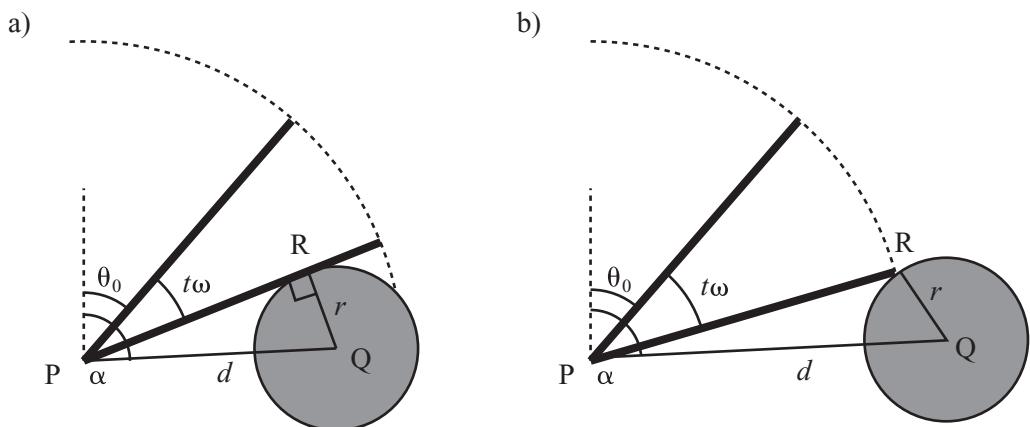


Figure 13.7

A rotating line and a circle (a) colliding on the flat and (b) colliding at the end point.

As mentioned previously, this equation will only work if the circle is close enough to P so that it is struck by the flat of the line, rather than its end point. If the line segment has a length of l , you can say that it collides on the flat if and only if $d^2 \leq l^2 + r^2$. You also know that if the circle is far enough out, it will not be struck by the line at all. This occurs when $d > l + r$.

What about in the mid-range? As illustrated by Figure 13.7b, this time you're looking at a slightly different problem. Because the circle is not struck at a tangent, PRQ is no longer a right-angled triangle. However, you do know the length PR, since it is the length of the line segment l . This means you can use the cosine rule to determine the angle RPQ:

$$\cos(\alpha - \theta) = \frac{l^2 + d^2 - r^2}{2ld}$$

$$\theta = \alpha - \cos^{-1} \left(\frac{l^2 + d^2 - r^2}{2ld} \right)$$

$$t = \frac{1}{\omega} \left(\alpha - \theta_0 - \cos^{-1} \left(\frac{l^2 + d^2 - r^2}{2ld} \right) \right)$$

It is important to remember that angular calculations occur in a closed field, where $\alpha + \beta$ is not always greater than α or β . It's simplest to shift calculations around to make sure that the numbers do not exceed 360° or fall below 0° .

Given this preparation, you can construct the `angularCollisionLineCircle()` function. This function determines all possible collisions between a rotating line and a circle:

```
function angularCollisionLineCircle(theta0, omega, l, r, d, alph)
    if d>l+r then return "no collision"
    if d<r then return "embedded"
    // move into a calculation within the range [0,2pi]
    subtract theta0 from alph
    if omega<0 then
        set omega to -omega
        set alph to -alph
        set k to -1
    otherwise
        set k to 1
    end if
    while alph<0 add 2*pi to alph
    while alph>2*pi subtract 2*pi from alph
    // check if there is a possible collision
    if alph>omega then return "no collision"
    // now perform the appropriate collision check
    if d*d<=l*l+r*r then
        return (alph-k*asin(r/d))/omega
    otherwise
        return (alph-k*acos((l*l+d*d-r*r)/(2*l*d)))/omega
    end if
end function
```

A similar calculation that lacks the option of hitting on the flat allows you to calculate the collision of a rotating point with a circle, as occurs when a player kicks a soccer ball with the toe of his or her boot. Another way to view this is as the vertex of a rotating polygon.

As shown in Figure 13.8, when the line segment is rotating about some point not on the line, as with an edge of a rectangle, things are not much more difficult to reckon with. In this case, you can see a line segment rotating around a point P, at a perpendicular distance k from the line. As before, a circle sits at Q, a distance d from P and angle α around it.

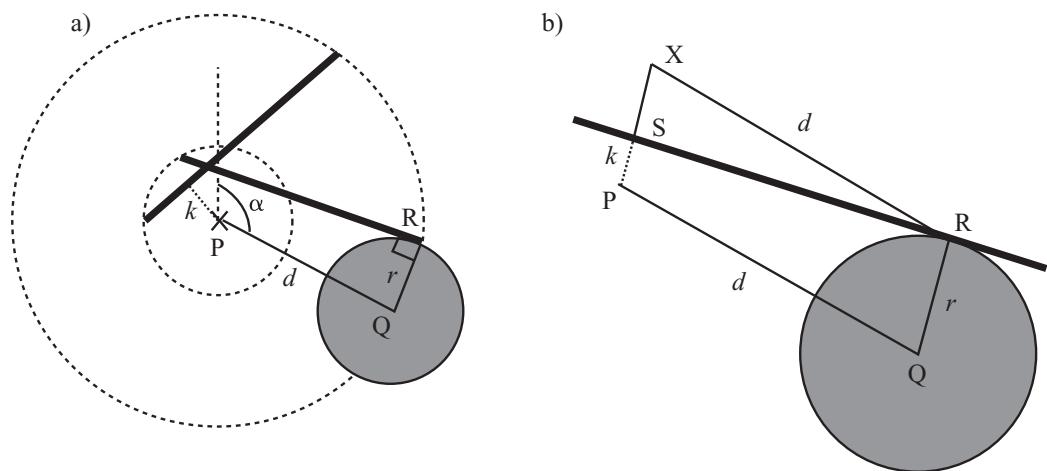


Figure 13.8

A line rotating about an offset point (a) setup and (b) collision detail.

Figure 13.8b illustrates the moment of collision. In this case, point R is where the circle meets the line, and point S is where the perpendicular from P meets the line. Likewise, the line PQ has been drawn. Notice that both PS and QR are perpendicular to the line, so they are parallel. If you draw a new line from R parallel to QP, it meets the extended line SP at a point X, a distance $r - x$ from S, giving you a right-angled triangle XSR, with one side equal to $r - k$ and a hypotenuse of d . From this you can work out the angle $\text{RXS} = \theta - \theta_0$ and the distance RS. With a bit of angle juggling, it turns out that there is only the most minor adjustment to the calculation for collision on the flat:

$$t = \frac{1}{\omega} \left(\alpha - \theta_0 - k \sin^{-1} \left(\frac{r - x}{d} \right) \right)$$

Since the end points move in a circle around P at a distance $\sqrt{l^2 + x^2}$ for each length, this is a different application of the previous method. This particular collision method is important, however, for any rotating polygon is a collection of line segments rotating around some axis.

A Circle and a Moving Line

A problem that is more difficult than those addressed in the previous section concerns a circle that moves when it hits the line. As illustrated by Figure 13.9, the circle moves along the path $\mathbf{q} + t\mathbf{v}$, and the line rotates around the point P, as before. At some time t , the circle and line touch, and two important construction lines have been drawn in.

Figure 13.9 assumes that the line begins rotating at the angle 0 . In order to calculate other situations, you first need to rotate the frame of reference around P. Also in these calculations, you are assuming that any collision occurs in the first quartile of motion—that is, that the angular displacement of the line in this time-step is less than 90° . This simplifies calculations. If you need to deal with larger angles, you must split the calculation into several subproblems. A final point is that it is easiest to assume that P is the point $(0,0)$.

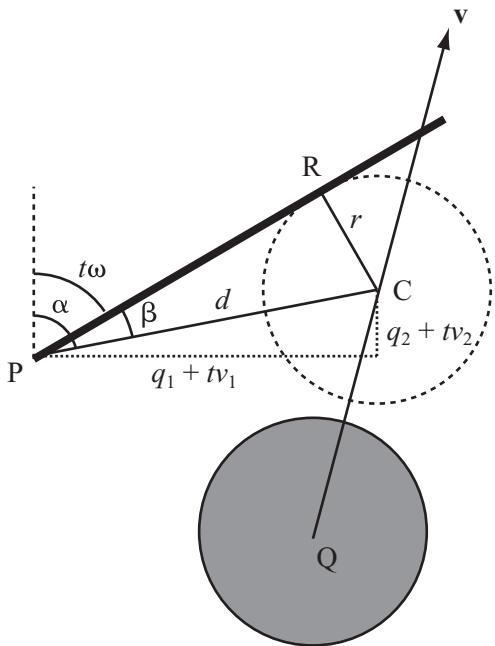


Figure 13.9

A moving circle and a rotating line.

To solve the problem presented by Figure 13.9, consider first the values α and β . The value α is the angle formed by the line CP at the point of collision, where C is the center of the circle. The value of β is the angle formed at the point of collision between CP and the rotating line. Given these values, you can conclude that at the moment of collision, either $\alpha - \beta = t\omega$ or $\alpha + \beta = t\omega$. You can calculate which of these might happen by comparing the initial positions and motions of the two objects. To solve the problem, you can say that $\alpha = t\omega + k\beta$, where $k = \pm 1$.

With a little work using trigonometric identities, along with temporary use of the value d , the length of CP as a function of t can be developed. The objective of the resulting equation is to find the value of t :

$$\begin{aligned}\frac{q_1 + tv_1}{d} &= \sin \alpha \\ &= \sin(k\beta + t\omega) \\ &= \cos \beta \sin(t\omega) + k \sin \beta \cos(t\omega)\end{aligned}$$

$$\begin{aligned}\frac{q_2 + tv_2}{d} &= \cos(k\beta + t\omega) \\ &= \cos \beta \cos(t\omega) - k \sin \beta \sin(t\omega)\end{aligned}$$

$$\begin{aligned}\cos(t\omega) \left(\frac{q_1 + tv_1}{d} \right) - \sin(t\omega) \left(\frac{q_2 + tv_2}{d} \right) &= k \sin \beta (\cos^2(t\omega) + \sin^2(t\omega)) \\ &= k \sin \beta = \frac{kr}{d}\end{aligned}$$

$$\cos(t\omega)(q_1 + tv_1) - \sin(t\omega)(q_2 + tv_2) = kr$$

As becomes evident from inspecting the work so far, the equation does not isolate t . This is all that can be done algebraically. Since the equation cannot be solved exactly, you must bring forward the approximation methods introduced in Chapter 6. In this case, because the function is fairly smooth and you are looking over a small range with clear boundaries (or brackets) at 0 and 1, the safest approach is to use the simple bisection or Regula Falsa method. This is preferable to a more sophisticated approach, such as the Newton-Raphson method, which could easily find a root outside the range you're interested in.

To complete the function, you also need to perform a check for collision with the end point of the line. The basic algebra remains the same, but this time the value of d doesn't drop out of the equation; instead, you are stuck with using the cosine rule to discover value of β . The equation is $\cos \beta = \frac{l^2 + d^2 - r^2}{2ld}$. Working this out, you are left with a slightly more complicated equation:

$$\begin{aligned} \sin(t\omega)(q_1 + tv_1) + \cos(t\omega)(q_2 + tv_2) &= \frac{l^2 + d^2 - r^2}{2l} \\ &= \frac{(q_1 + tv_1)^2 + (q_2 + tv_2)^2 + l^2 - r^2}{2l} \end{aligned}$$

As before, since the equation can't be solved algebraically, an approximation method must be used. You can save time as you approach this task by performing an appropriate initial check to see whether it is possible for the two objects to collide at all. In fact, it is worth performing two checks. First determine if the circle intersects the complete circle swept out by the line. Complete this check before checking for rotational collision. Second, determine if the angle swept out by the circle during the time interval overlaps with the angle swept out by the line. This check gives you a value for k .

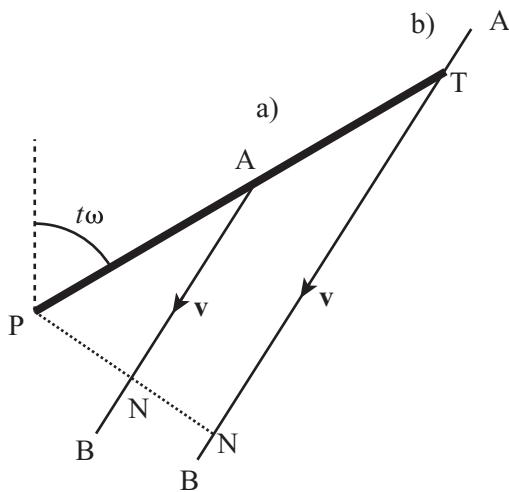
Note

There isn't space here to cover the case where the rotating line is offset from its rotation point. While the algebraic technique is similar, the calculations are messier.

In all of these problems, you must deal separately with the two ends of the rotating line. This involves performing a check for a collision between the circle and the stationary pivot. Otherwise, you must perform the calculation twice, adding π to the initial angle for the reverse side of the line.

Detecting Collisions Between Lines

The next topic that emerges concerns collisions between lines. While it might appear that this problem is fairly straightforward, as it turns out, it is subtle to the extent that, as before, to solve it you must resort to methods of approximation. As before, you must arrive at the solution gradually. To start with, as shown in Figure 13.10, consider a case in which a rotating line meets a stationary line segment. In this case, there are two separate cases to consider. The first involves a collision with the body of the line segment. The second involves a collision with one of the end points. In Figure 13.10, these are shown in reverse order.

**Figure 13.10**

A rotating line and a line segment (a) colliding at an end point and (b) colliding on the flat.

As is common when dealing with such problems, you assume that you know the position vector \mathbf{a} of one end point A and the vector \mathbf{v} from this point to the other end point B. You can also determine the point N, where the perpendicular from P (defined as the origin) hits the line segment. From here, you proceed as was outlined earlier in the chapter. You create a useful function, `clockwise()`. This function returns +1 if the vector \mathbf{v} from \mathbf{a} is directed clockwise around the origin or -1 if it is directed counterclockwise. In some circumstances, it might be useful for the function to return 0 if the vector points directly at or away from the origin. In this version of the function, this capability is not included. Here is the implementation of the `clockwise()` function:

```
function clockwise(p, v)
    set n to clockwiseNormal(p)
    if component(v,n)>0 then return 1
    return -1
end function
```

To supplement the `clockwise()` function, the `clockwiseNormal()` function is also constructed:

```
function clockwiseNormal(v)
    return vector(-v[2],v[1])
end function
```

The `clockwiseNormal()` function specifies a particular direction as clockwise. That the direction is arbitrary creates no problem as long as you use it consistently. For some display purposes, what you call “clockwise” might look as if it is running counterclockwise. For example, your y -value might be measured upward or downward. Still, you must ensure that it ties in correctly with your measure for positive angular displacement. This you can accomplish by creating two general-purpose functions. The `unitVector()` and `angleOf()` functions satisfy this need:

```
function unitVector(ang)
    return vector(sin(ang),cos(ang))
end function

function angleOf(v)
    return atan(v[1],v[2])
end
```

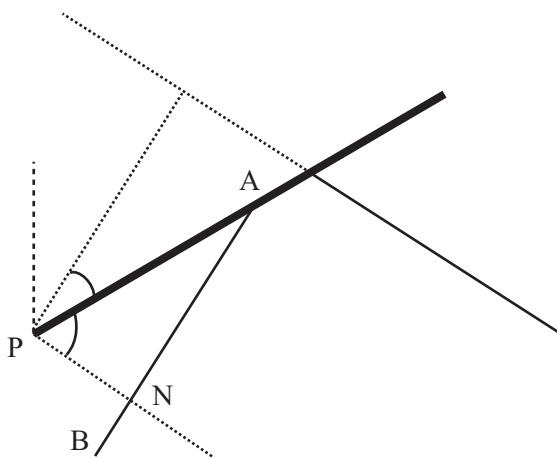
If you combine these functions, you ensure that your values remain consistent.

Note

In three dimensions, this process can be dealt with more formally by using the cross product and the right-hand rule. Both of these topics are addressed in Chapter 16.

Now you can return to the problem at hand—detecting line collisions between a moving and a stationary line. You deal with the simplest case first. Suppose your rotating line strikes the segment at some point T between A and B. In that case, it forms a right-angled triangle PNT, where PT is the length l of the line. This immediately gives you a value for the angle at which it lies. What’s more, if the line is rotating clockwise, it strikes along the counterclockwise direction of \mathbf{v} , and vice versa. You then must check that only T actually lies on the line. This is true if $0 \leq \overrightarrow{AT} \cdot \mathbf{v} \leq |\mathbf{v}|^2$.

Dealing with the end points is fairly simple, too. If the line is rotating in the same direction as \mathbf{v} , you need to look for a collision with A. If it is rotating the other way, you need to check for a collision with B. Assuming that it is rotating in the same direction, the line forms a right-angled triangle PNA. This time, as illustrated by Figure 13.11, you can determine the length AN in advance, and this gives you the correct angle. However, you must use caution since you must verify that you measured the angle in the correct direction. You must check if N lies on AB. In other words, you check whether vector \overrightarrow{AN} lies in the same direction as \overrightarrow{AB} . If it does, you want the counterclockwise angle. If it does not, you want the clockwise angle.

**Figure 13.11**

Determining the correct direction of the collision angle with an end point.

Given the discussion so far, it becomes possible to construct the `angCollLineStatLine()` function. As the name implies, this function addresses collisions involving one stationary line and one moving line.

```

function angCollLineStatLine(theta0, angvel, length,
                           linept, linevect, segment)
  // segment=1 if you are checking for end points,
  // 0 for a continuous wall
  set n to norm(normal(linevect))
  set d to dotprod(linept,n)
  if d<0 then
    set d to -d
    set n to -n
  end if
  // so n is the normal vector directed toward N
  if d>length then return "none" // too far from wall

  // if checking for end points, see if they are relevant
  if segment=1 then
    set pn to n*d // the vector PN
    set dd to length*length-d*d // the squared length TD
    if angvel>0 then
      if clockwise(linept, linevect)=-1 then
        set endpt to linept

```

```
otherwise
    set endpt to linept+linevect
end if
otherwise
if clockwise(linept, linevect)=-1 then
    set endpt to linept+linevect
otherwise
    set endpt to linept
end if
end if

set d1 to sqmag(endpt-pn) // sqmag is the squared magnitude
if d1<dd then // there is a potential collision with the end point
    set a to acos(d1/mag(endpt))*clockwise(endpt,pn-endpt)
    // a is the angle of collision with the end point
otherwise
    set a to acos(d1/l)
    if angvel>0 then set a to -a
    // check if this collision occurs outside the line segment
    set ap to pn+abs(a)*sqrt(dd)/a
    // note that abs(a)/a is 1 if a>0, -1 otherwise
    set k to mag(ap-linept)/mag(linevect)
    if k>1 or k<0 then return "none"
end if

otherwise
// check for collision with an infinite wall
set a to acos(d1/l)
if angvel>0 then set a to -a

end if

set tn to angleof(n)
set t to rangeangle(tn-thet0+a,1)/angvel
if t<=0 or t>1 then return "none"
return t
end
```

Two Rotating Lines

Comprehensively addressing collisions in which both lines are rotating exceeds the scope of this discussion, but it is still possible to lay out the essential problem. As Figure 13.12 illustrates, the problem is in some ways easier than those examined previously because for all collisions except the degenerate case where the two end points collide exactly, every collision is between the end point of one line and the flat of the other.

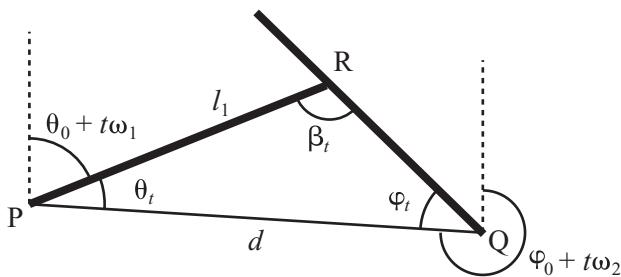


Figure 13.12

Collisions between two rotating lines.

To solve the problem, some preliminary work is needed. First, notice that it is possible to calculate the angle α of the line joining the two pivot points P and Q. Using this angle, you can define two functions giving two angles of the triangle formed by the two lines and the line PQ:

$$\theta_t = \alpha - \theta_0 - t\omega_1$$

$$\varphi_t = \varphi_0 + t\omega_2 - \alpha - \pi$$

Now you know that if the third angle of the triangle PQR can be formed, it will be equal to $\beta_t = \pi - \theta_t - \varphi_t$. To check for whether the lines collide, you first determine whether the value β_t lies between 0 and π . This gives an inequality for t :

$$\pi \leq \varphi_0 - \theta_0 + t(\omega_2 - \omega_1) \leq 2\pi$$

Observe that α drops out. If this inequality holds for any t between 0 and 1, you have a potential collision triangle in this time frame.

If the triangle is formed correctly, you can use the sine rule to determine whether the triangle is a collision. This eventuates if either $PR = l_1$ and $QR = l_2$, or $PR = l_1$. As a result, you have

$$\frac{\sin \beta_t}{l_1} = \frac{\sin \varphi_t}{d}$$

or

$$\frac{\sin \beta_t}{l_2} = \frac{\sin \varphi_t}{d}$$

where d is the length of PQ .

At this stage you have to return to approximation methods once more. Notice also that this explanation has left out discussion of the direction of motion of the two lines. Direction affects exactly which values you should use in the triangle PQR .

Whether the pivot points are moving relative to one another doesn't significantly alter the method. However, you must change d and α into functions of t as well. While such changes complicate the calculations, the theory remains the same.

Resolving Angular Collisions

Angular collisions are only mildly more difficult to resolve than linear collisions. To calculate such collisions, you can make use of ideas presented in Chapter 12, and a primary concern is to consider the behavior of the collision at the point of impact. If the two bodies are rotating, the velocity of a point on the edge may not be equal to the velocity of the body as a whole. Recall from earlier discussions that you calculated the velocity of a point spinning about a fixed axis to be $\omega \mathbf{r}_N$. More generally, if the body is moving with velocity \mathbf{v} , velocity becomes $\mathbf{v} + \omega \mathbf{r}_N$.

At the moment of impact, the impulse J experienced by the two bodies affects both their linear and angular momentum. If you suppose their initial velocities are \mathbf{u}_1 and \mathbf{u}_2 , their final velocities are \mathbf{v}_1 and \mathbf{v}_2 , their initial angular velocities are ω_1 and ω_2 . Their final angular velocities are φ_1 and φ_2 . Use m to represent their masses, I for their moments of

inertia, and \mathbf{m}_1 and \mathbf{m}_2 for the “moment vectors.” The two vectors are clockwise normals of the radius vectors. Finally, suppose the collision normal is \mathbf{n} . This allows you to use four equations to represent the effect of the collision:

$$m_1 \mathbf{v}_1 = m_1 \mathbf{u}_1 + J \mathbf{n}$$

$$m_2 \mathbf{v}_2 = m_2 \mathbf{u}_2 - J \mathbf{n}$$

$$I_1 \varphi_1 = I_1 \omega_1 + J \mathbf{m}_1 \cdot \mathbf{n}$$

$$I_2 \varphi_2 = I_2 \omega_2 + J \mathbf{m}_2 \cdot \mathbf{n}$$

Given the presentation of these equations, it is necessary to make an assumption. The assumption is that the behavior of the points of impact is going to be the same for both of the points, independent of the angular motion of the objects behind them. You can break this down a little. The two colliding points are moving with a relative velocity that you can calculate to be $\mathbf{u}_2 - \mathbf{u}_1 + \mathbf{m}_2 \omega_2 - \mathbf{m}_1 \omega_1$. From the point of view of the collision, it doesn't know anything about the rest of the objects. The collision sees only that the objects are moving with this relative velocity and that they have a certain mass. The result is that the collision will make the object rebound in exactly the same way as it would otherwise. In other words, assuming that the collision is elastic, it “wants” the relative velocity normal to the collision to obey this equation:

$$(\mathbf{v}_2 - \mathbf{v}_1 + \mathbf{m}_2 \varphi_2 - \mathbf{m}_1 \varphi_1) \cdot \mathbf{n} = -(\mathbf{u}_2 - \mathbf{u}_1 + \mathbf{m}_2 \omega_2 - \mathbf{m}_1 \omega_1) \cdot \mathbf{n}$$

Although you haven't encountered this specific equation before, it is actually a restatement of the Conservation of Energy. You can see it in the linear collisions you've already encountered. Since collisions can only affect the normal direction, energy can only be conserved if the magnitude of the normal velocity going in to a collision is equal to the magnitude coming out.

Combining these five equations together, you get

$$\begin{aligned} & \left(\mathbf{u}_2 - \frac{J}{m_2} \mathbf{n} - \left(\mathbf{u}_1 + \frac{J}{m_1} \mathbf{n} \right) + \mathbf{m}_2 \left(\omega_2 - \frac{J}{I_2} \mathbf{m}_2 \cdot \mathbf{n} \right) - \mathbf{m}_1 \left(\omega_1 + \frac{J}{I_1} \mathbf{m}_1 \cdot \mathbf{n} \right) \right) \cdot \mathbf{n} \\ &= -(\mathbf{u}_2 - \mathbf{u}_1 + \mathbf{m}_2 \omega_2 - \mathbf{m}_1 \omega_1) \cdot \mathbf{n} \end{aligned}$$

Combining terms and simplifying, you get

$$2(\mathbf{u}_2 - \mathbf{u}_1 + \mathbf{m}_2 \omega_2 - \mathbf{m}_1 \omega_1) \cdot \mathbf{n} - J \left(\frac{1}{m_2} \mathbf{n} - \frac{1}{m_1} \mathbf{n} + \mathbf{m}_2 \left(\frac{1}{I_2} \mathbf{m}_2 \cdot \mathbf{n} \right) + \mathbf{m}_1 \left(\frac{1}{I_1} \mathbf{m}_1 \cdot \mathbf{n} \right) \right) = 0$$

Assuming \mathbf{n} is a unit vector, this becomes

$$2(\mathbf{u}_2 - \mathbf{u}_1 + \mathbf{m}_2 \omega_2 - \mathbf{m}_1 \omega_1) \cdot \mathbf{n} - J \left(\frac{1}{m_2} + \frac{1}{m_1} + \frac{1}{I_2} (\mathbf{m}_2 \cdot \mathbf{n})^2 + \frac{1}{I_1} (\mathbf{m}_1 \cdot \mathbf{n})^2 \right) = 0$$

which you can rearrange to find J . Finally, you can plug the value for J back into the four momentum equations to find the new velocities.

Notice that this formula yields different results if you set particular masses or moments of inertia infinitely high, creating an object that is fixed in place either linearly or angularly. For example, if you set both of the moments of inertia infinitely high, these terms drop out of the formula, leaving you with the formula for linear collision resolution. Likewise, notice that angular velocities are then unaffected by the impulse.

Including inelastic collisions isn't much harder. If you do so, it affects the "2" term, replacing it with a $(1 + e)$ term, where e is the *coefficient of restitution*. Its value can vary from 0 (putty) to 1 (atom).

Drawing the discussion together, you can construct the `resolveAngularCollision()` function:

```
function resolveAngularCollision (obj1, obj2, n, mom1, mom2)
    set u1 to obj1.getVelocity()
    set u2 to obj2.getVelocity()
    set om1 to obj1.getAngularVelocity()
    set om2 to obj2.getAngularVelocity()

    set J to 2*dotProduct(u2-u1+mom2*om2-mom1*om1,n)
    set denom to 0
    if not obj1.fixedLinear() then
        set m1 to obj1.getMass()
        set denom to denom + (1/m1)
    end if
    if not obj2.fixedLinear() then
        set m2 to obj2.getMass()
        set denom to denom + (1/m2)
    end if
    if not obj1.fixedAngular() then
        set moi1 to obj1.getMOI()
        set dp1 to dotProduct(mom1,n)
        set denom to denom + (dp1*dp1/moi1)
```

```

end if
if not obj2.fixedAngular() then
    set moi2 to obj2.getMOI()
    set dp2 to dotProduct(mom2,n)
    set denom to denom + (dp2*dp2/moi2)
end if
if denom=0 then exit // coincident axes or other weirdness
set J to J/denom
if not obj1.fixedLinear() then obj1.setVelocity(u1+J*n/m1)
if not obj2.fixedLinear() then obj2.setVelocity(u2-J*n/m2)
if not obj1.fixedAngular() then obj1.setAngularVelocity(om1+J*dp1/moi1)
if not obj2.fixedAngular() then obj2.setAngularVelocity(om2-J*dp2/moi2)
end function

```

As before, use of the `resolveAngularCollision()` function assumes that the two bodies are represented by some sort of object whose velocity can be directly altered.

Spin Applied to the Pool Game

As a closing topic of discussion, it should be reasonably simple to see how these ideas might be incorporated into the pool game discussed in Chapter 11. In addition to defining a vector of motion, you must decide whether to strike the ball to the left or right of center. Striking the ball imparts a force that is offset from the center of mass, with the result that an angular momentum is created. You can account for the angular momentum using a linear function of the distance from the center.

Note

Spin is sometimes called “English,” but it is clearer in this context to refer to it as “sidespin.” The term English originated as a transformation of the French word *angle*, which designates a geometric angle, but with an accent mark also designates “English.”

Since pool balls are circular, angular motion has no effect on collision detection, but it does affect collision resolution. The reason for this is that one of the assumptions from the previous section does not hold in pool. As a result of friction between the cushions and the balls, and also with the table, the impulse on colliding objects acts normal to the collision. This means that some of the ball’s angular momentum can be transformed into linear momentum, and vice versa.

Figure 13.13 shows the effect of the collision between the pool ball and the cushion. You can imagine the collision taking place in two parts. First, the angular velocity decreases by some amount ϕ , but in order to achieve this, there must be an impulse J at the point of contact, along the wall. You can calculate J by seeing that $I\phi = Jr$. Now you can apply this impulse to calculate the resulting change in linear velocity:

$$mv = mu + Jt$$

$$v = u + \frac{I\phi}{m} t$$

Having calculated this new velocity, you can apply the perpendicular impulse, as usual.

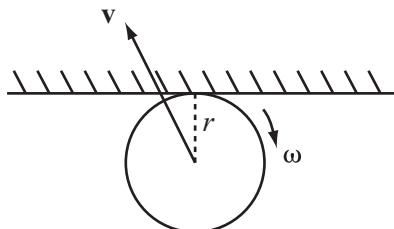


Figure 13.13

Collision with a sticky surface.

Exercises

EXERCISE 13.1

Write a function named `resolveCushionCollision(obj1, obj2, normal, moment, slow)`, which calculates the result of a collision with a wall that removes some constant amount of angular velocity.

This function should calculate both the angular and linear velocities, given that the proportion prop of angular velocity has been lost.

EXERCISE 13.2

Write a function that calculates a collision between a moving, rotating square and a wall. This is a tough one and not for the squeamish.

Summary

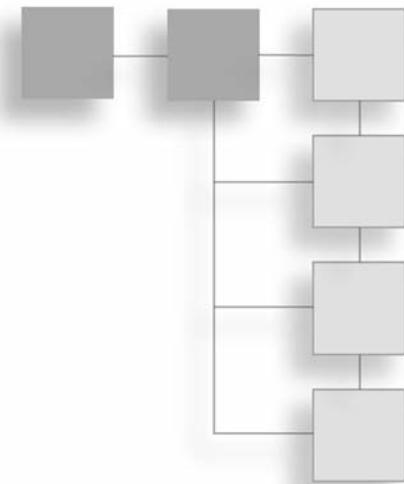
Angular motion is simple in essence but complicated in the details and implementation. It is no surprise that, as a rule, game designers avoid it whenever they possibly can. In this chapter, you have at least seen the beginnings of how to implement a collision system that takes spin into account. At the very least, you now know how to deal with it when working with circles and lines. In the next chapter, you'll extend this subject a bit more by looking at how spin is related to friction.

You Should Now Know

- How a lever allows you to move and balance objects
- The meaning of torque and its relationship to force
- How angular momentum and energy are similar to their linear counterparts
- How to find the point of collision between some simple spinning shapes
- How to resolve a collision between one or more spinning objects

CHAPTER 14

FRICITION



In This Chapter

- Overview
- How Friction Works
- Friction and Angular Motion

Overview

Chapter 13 addressed some initial topics concerning rotation, and this chapter continues with that theme. Previous discussions dwelled on sideways spin in the game of pool but glossed over topspin and backspin, which are more common. Both topspin and backspin are usually referred to as *rolling*, and rolling is probably what first comes to mind when you consider what a pool ball does. The question arises, however, as to why and how a ball rolls at all. After all, if a ball were moving through empty space, it would not be expected to roll. In fact, in calculations of momentum in Chapter 13, spin was completely ignored.

A ball rolls for the same reason that a block stays still: the force of friction. As a ball rolls, its surface is in contact with another surface. Whenever two objects are in contact with each other and move in opposing directions tangentially to their point of contact, friction arises. You've already made use of this notion to explain why pool balls rebound at an angle when they strike a cushion with sidespin (or "English") applied. In this chapter, this phenomenon is examined in greater detail.

How Friction Works

Friction is principally caused by irregularities on the surface of an object. Consider the surface of a brick compared to the surface of marble. The surface of a brick is rough while the surface of marble is smooth. A brick will have more friction than a piece of marble if it slides along a given surface. If you were to use a microscope to examine two surfaces moving against each other, you would see pits and ridges on the surfaces. As these surfaces collide with each other, they create a force that acts tangentially to the plane of contact—or along the direction of motion. This, then, is friction. In the last chapter, you looked at the result of an instantaneous frictional force applied during a collision. Here, you look at the more conventional issue of friction acting continuously on a sliding object.

The Coefficient of Friction

Because its magnitude varies according to the force perpendicular to it, friction is an unusual force. In fact, friction is proportional to the normal force. For any two objects there is a value μ , the *coefficient of friction*, such that

$$\text{Force due to friction} = \mu \times \text{force perpendicular to friction}$$

Figure 14.1 shows an example of friction in action. A box with weight W moves on a slope (often called an *inclined plane*) with an angle of θ to the horizontal. The weight of the box acts downward, which means that it experiences a force of $W \cos \theta$ perpendicular to the slope. This gives it a frictional force of $\mu W \cos \theta$ along (and opposing) the direction of motion. The total force experienced tangential to the plane is $W \sin \theta - \mu \cos \theta = W (\sin \theta - \mu \cos \theta)$.

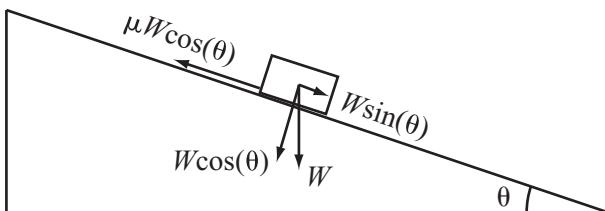


Figure 14.1

Friction on an inclined plane.

In Figure 14.1, if you reflect on the implications of Newton's third law, since the box experiences an opposing normal force of $W \cos \theta$, it is in equilibrium normal to the plane. The friction is created by the opposition of these two forces. The frictional force applies as long as the box is moving, so for example, if the plane has a slope of exactly the correct angle, friction will cancel the force due to gravity. The result is that the box will move with a constant speed, a fact that you use to measure the coefficient of friction. To calculate the angle at which friction is canceled, you can use the following approach:

$$\begin{aligned} W(\sin \theta - \mu \cos \theta) &= 0 \\ \sin \theta &= \mu \cos \theta \\ \tan \theta &= \mu \end{aligned}$$

If θ is smaller than μ , then the box gradually slows to a stop; if θ is greater than μ , then gravity overcomes friction and the box accelerates.

The discussion so far has been somewhat simplified because in fact there are two different coefficients of friction. The previous discussion concerns the *coefficient of kinetic friction*. However, a second form of friction also applies. This is the *coefficient of static friction*, which applies when an object is not moving but is experiencing a sliding force. In this situation, the force of the object resists movement.

The coefficient of kinetic friction is usually written as μ_K while the coefficient of static friction is written as μ_S . Since it represents a maximum possible force, rather than the actual value, static friction is slightly harder to grasp than kinetic. As long as you push it with a smaller amount of force than what is accounted for by static friction, the object will not move. An equal force resists the force of movement.

In general, the coefficient of static friction (μ_S) is greater than the coefficient of kinetic friction (μ_K). The result is that more force is required to make something move than to accelerate it when already moving. In the example given by Figure 14.1, involving an inclined plane, imagine that the box is resting on the plane. The static friction is given by $\mu_S W \cos \theta$, which is the maximum possible frictional force applied while static. As long as the force due to the box's weight is less than this, the box will not move. In other words, the box remains stationary if $\tan \theta < \mu_S$, and the value of θ in this condition is sometimes called the *critical angle*.

Ultimately, then, there is a region of angles for the plane such that if the box is stationary, it will not move. Likewise, there are also regions or angles for the plane for which, if the box is moving, it will continue to move. This applies as long as $\mu_K < \tan \theta < \mu_S$.

The `resultantForceOnObject()` function applies this generality:

```
function resultantForceOnObject (nonFrictionalForce, velocity,
                                coefficient)

    set tang to norm(velocity)
    set norm to normalVector(tang)
    set normalForce to component (nonFrictionalForce, norm)
    set tangentialForce to component (nonFrictionalForce, tang)
    set frictionalForce to coefficient * magnitude(normalForce)
    if frictionalForce > tangentialForce and magnitude(velocity) > 0
        then return vector(0,0)
    otherwise return (tangentialForce - frictionalForce) * tang
end function
```

The `resultantForceOnObject()` function works equally well for static and kinetic friction. To accommodate the two types of friction, replace the coefficient with the correct value depending on whether the object is currently moving or stationary. The parameter `nonFrictionalForce` is the force on the object due to any factors other than its interaction with the plane—the force it would be experiencing if the plane was not there. The return value is the total force on the object, including both friction and the reaction force due to Newton’s third law, and is therefore parallel to the plane. You might like to try making a more specialized version of this function to deal with the specific case of an object sliding down an inclined plane.

Friction and Energy

When friction must be accounted for in a mechanical simulation, issues arise that involve conservation of energy and momentum. Since it isn’t useful when dealing with objects that are fixed in place, however, conservation of momentum is not an important concern. The energy is mostly lost to heat to the detriment of both bodies involved.

You can calculate the energy lost due to friction by looking at the amount of *work*. Work is another word for energy. It designates the energy used to perform a task, usually making an object move. Here is how work is expressed:

$$Work = Force \times Distance\ moved$$

This expression indicates that the harder you push something, the more energy it takes to travel a particular distance. And notice also that if something doesn’t move, no work is done, no matter how hard you are pushing it. Work is *useful* energy.

Defining work as useful contrasts to the normal usage of the word “work.” The normal usage suggests that whether something moves does not matter. For example, consider what happens if you push at an immovable brick wall with all your strength. As furious as it might be, this effort involves energy only on your part. The wall does not change position. As a result, in terms of the physics of the event, no work is done. Your energy is mostly spent on chemical reactions in your muscles. This is a case involving static friction, and friction of this type does not affect energy calculations.

However, if an object is moving and experiencing kinetic friction, then you can calculate the work done by the friction, which is a constant multiple of the distance traveled:

$$\text{Work done} = \text{Frictional force} \times \text{distance} = \mu \times \text{Normal force} \times \text{distance}$$

This equation is complicated by the fact that the distance traveled is also a function of the force. Suppose your object is sliding on a table, like a hockey puck. In this case, the friction is the only force acting on it. In a particular time t , the puck moves by a distance equal to $ut + \frac{1}{2}at^2$. By Newton's second law, since the acceleration is the force divided by the mass, you have $d = ut + \frac{Ft^2}{2m}$, so the energy lost is equal to $Fut + \frac{F^2t^2}{2m}$. Still, since the force depends on the puck's weight, mg , the equation must take the following form:

$$\text{Energy loss} = \mu mg \left(ut + \frac{\mu gt^2}{2} \right)$$

On the other hand, it's easier to calculate the loss of speed, which is a simple linear decrease:

$$\text{Speed loss} = \mu gt$$

If you look back at the pool game simulation as explained in previous chapters, you can see that this turns out to be exactly as you calculated it. However, since events become more complex when you deal with balls rather than boxes, it will be necessary to once again examine what happens.

Air Resistance and Terminal Velocity

There is another kind of friction you haven't looked at yet, which is *air resistance*, or more generally, *fluid resistance* or *drag*. The word “fluid” is used to mean either a liquid or a gas since, generally, liquids and gases have similar properties. With respect to how drag can be understood, the previous definition of friction must be qualified. The primary reason for this is that force is applied in a different way. A body falling through the air

meets resistance due to constant forward collisions with air molecules. This type of collision differs from what has been discussed previously, where a sliding force like a box on the ground has been involved.

Calculating drag is more complicated than calculating friction. Like friction, the important value is a constant coefficient, the *drag coefficient*, commonly expressed as C_D . The equation for drag assumes the following form:

$$\text{Drag force} = \frac{1}{2} C_D \rho v^2 A$$

In the equation for drag, the Greek letter rho (ρ) is the pressure of the fluid, A is the forward area presented to the fluid, and v is the speed. Do not worry about the pressure and area terms. In this context, since you are dealing with simpler problems, the values for these variables are constant. All you need to know is that the drag force is proportional to the square of the speed. You can represent this using your own drag coefficient, which in this context can be represented by μ_D . Then you can write,

$$\text{Drag force} = \mu_D v^2$$

Note

Other terms should play a part, also. For example, the pressure of a fluid is a measure of how densely packed its molecules are, so the higher it is, the more collisions the falling body will experience. Similarly, the greater the surface area of the body, the more molecules it will hit on the way down. This is why a parachute falls slower than a pea. Since it, too, is dependent on the area of contact, you could also split up the coefficient of friction in much the same way.

Because drag is dependent on speed, at some stage, a falling object will reach a speed at which the drag is equal to the pull of gravity. At this speed, the object stops accelerating. This speed is called *terminal velocity*. You can calculate it as follows:

$$mg = \mu_D v^2$$

$$v = \sqrt{\frac{mg}{\mu_D}}$$

Since μ_D is proportional to area, terminal velocity will be inversely proportional to the square root of area. Furthermore, the area of a circle or a sphere is proportional to the square of its radius, so terminal velocity is inversely proportional to radius. If you double the radius of your parachute, you will halve your terminal velocity (which has to be a good thing).

Friction and Angular Motion

As the previous section revealed, drag is a reasonably straightforward topic. With respect to simulations, it is just another small thing to consider. In practical terms, it can be dealt with by simple methods such as those used previously for the pool game. However, other considerations also arise, somewhat complicating things.

Wheels and Grip

Cars have wheels, and forces move the wheels. When a car moves, linear forces in the cylinders are converted to a torque that is applied to the wheel axle (or axles in a four-wheel-drive machine). This causes the wheels to turn, but it remains to settle the question of just why does the car move. Tires are mounted on the wheels. Why don't the tires simply spin, as they do when a car is stuck in the mud? The answer lies in the behavior of friction. Friction makes the car move, and when friction works in this way, it is called *traction* or *grip*.

Grip is basically static friction. When your car's tires are in contact with the road and trying to turn, they meet with resistance from the static friction with the road. This resistance counteracts the torque on the wheels, meeting it with a forward force. Since it causes the car's wheels to roll instead of spinning in place, the force that moves your car forward is friction, and it acts in the direction of motion. In fact, the car only moves as a consequence of Newton's third law. The force that moves it forward is a reaction to the force exerted by the tires on the road. Because the road can't move backward, the car has to move forward.

You can calculate the force on the road at the point of contact. If the wheels are experiencing a total torque of T and have a radius of r , the force at any point on the surface of the wheel is $\frac{T}{2r}$. What happens if this force is greater than the static friction of the tire with the road? Then the wheel starts to slip, and the car doesn't move. Instead, it sits and literally "burns rubber." As a result, a major part of successful driving involves controlling the torque applied to the wheels.

Gearing provides even more control over the level of torque. When you press the accelerator, it increases the amount of fuel the engine burns, which increases the *power*, or amount of energy released per second. But power can be transferred in different ways. Just as work is force multiplied by distance, so power is given by torque and angular speed:

$$\text{Power} = \text{force} \times \text{speed} = \text{torque} \times \text{angular speed}$$

When increasing the power, the result is that the force/torque is increased, and thus the acceleration, or the speed can be increased. Gearing allows you to make the trade-off. At a low gear, you have high torque and high acceleration but low maximum speed. At a higher gear, you have less acceleration but can travel faster. This is why you use low gears to get up to speed, but then switch to a higher gear for efficient cruising. It's also why you're better off shifting down a gear to overtake than shifting up. Conversely, it's why in order to get out of a skid, you shift into a higher gear, reducing the torque and so making it easier to maintain your grip on the road.

Braking behaves in much the same way as acceleration, but the effects are the opposite. When you brake, you apply a reverse torque on the wheels. You don't want to stop them altogether by locking them since this creates a frictional force against the road that is too strong for static friction to overcome, resulting in a skid. Instead you try to bring the wheels to a controlled halt, maintaining traction with the road.

This leads back to pool balls. When you strike head-on with another ball or a cue stick, a pool ball at first starts to move without spinning, as if it were on a Newton's Cradle. However, this does not last long, for as it slides along the table, it experiences kinetic friction. This slows the whole ball down, but it also imparts a spin, since it's a force applied at an angle. Reaching critical speed, when the surface velocity of the spinning ball is equal to the current speed of the ball, it is no longer sliding. Instead, it is rolling like a wheel.

At this moment, kinetic friction ceases to be an issue, and static friction comes into play. Unlike kinetic friction, static friction doesn't slow the ball down, so its rate of speed changes. The change is characterized by lost energy due to drag in the air, inelasticity of collisions, and some kinetic friction affecting the speed of rolling. With respect to kinetic energy, each time the ball moves, it has to squash down fibers of cloth, and this imparts a certain amount of drag in itself.

The result of this is that balls on a pool table have a tendency to spin forward in the direction of travel. They spin. This type of spin is called *topspin*. In the reverse direction, it is called *backspin*. As with sideways spin, you can impart topspin or backspin when you strike the cue ball by applying the cue impulse off-center. Whatever you do, as soon as the ball starts to move, it begins to try to roll.

An additional complication applies to this situation. Topspin or backspin makes the ball act a bit like a wheel. If the ball has topspin greater than the natural rolling pace, the friction is going to act forward instead of backward, accelerating the ball forward at the same time as it slows its spin. Conversely, if it's spinning fast enough, with backspin the frictional force may be enough to slow the ball and make it travel in the opposite direction.

All of this is particularly noticeable when you deal with collisions. When colliding with another ball (characterized by a very low coefficient of friction), topspin is unaffected. The result is that the ball is still trying to roll forward even as it rebounds backward. This spin then acts like a wheel, driving the ball to try to continue in the same direction it was going before. If the ball has backspin at the moment of collision, then this increases its rebound.

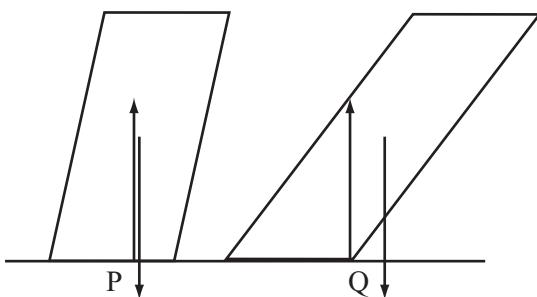
You should try these things out when you next encounter a pool table. Remember Newton's Cradle. When one ball strikes another along the line of the velocity vector, the first ball stops dead while the other moves off at the same speed. But if you hit the cue ball near the top, when it collides with the object ball, it stops briefly and then rolls forward. If you hit it near the bottom, it stops and then rolls backward.

Friction and Slipping

Not only wheels move by using friction. Humans propel themselves by the same principle. They use their feet to exert a backward force on the ground. The ground responds to this force with a frictional forward force.

For you to be able to take a step, it is necessary for traction to exist between your feet and the ground. If there is a low coefficient of friction, this technique won't work. Like a wheel spinning, the force you exert backward quickly overcomes the static friction and acts as a torque on your body. The result is spin, and you slip or fall. Various tricks can be used to prevent the risk of this, including swinging your arms to provide a counter-acting torque. This action is similar to that of a helicopter, which uses a second propeller to overcome the rotation induced by the first. If all else fails, you can switch to a different strategy for movement. Instead of using friction, you can use balance, moving one leg at a time a short distance forward while balancing carefully on the other. Animals with four feet have an advantage here.

Generally, an object falls over if it experiences a net torque. As shown in Figure 14.2, two objects are experiencing an upward force from the ground and a downward force due to their weight. In both cases, the weight can be thought of as passing through the center of gravity. In the first case (P), the weight is directed through a point of contact with the ground at P, which means that the net torque is zero. In the second case (Q), the weight passes through the ground outside the point of contact, which means that the point Q acts as a fulcrum.

**Figure 14.2**

Falling over.

When you walk, you use your sense of balance to manipulate these forces, rolling your foot to change the point of contact with the ground and shifting your arms and your hips. Eventually, you reach a moment when you begin to fall over, but then your other foot reaches the ground and catches you.

Note

With respect to Figure 14.2, you might think in the first case (P) that since the center of gravity is not above the center of the line of contact, there would be a net force to one side of it, giving it a torque. However, just as the weight acts through the center of gravity, the reaction force also balances out and can be thought of as a point force upward at P. The effect is a little like when you use two trestle tables to support a platform. The weight is distributed across the two fulcra. In this case, the trestle is upside down.

Figure 14.3 illustrates how you make use of the forces discussed in the previous paragraphs to maximize the advantage of friction. When running, since you get much stronger torque from the ground, to stop from spinning, you lean forward, creating a counter-torque. As a bonus, it also means that the force you can apply to the ground is greater, since a greater proportion of the strength of your legs is directed along the line of the ground.

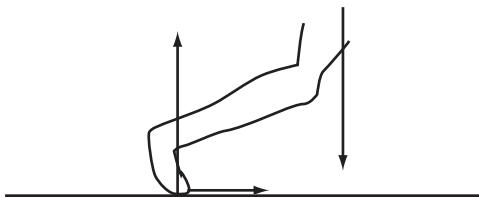


Figure 14.3

Using friction when running.

Exercise

EXERCISE 14.1

Write a function named `applyFriction()` with parameters for `velocity`, `topSpin`, `radius`, `mass`, `muK`, `muS`, and `time`. Taking friction into account, calculate the motion of a pool ball.

The function should take the current status of the ball and return its new velocity and spin. It should account for the coefficients of friction. Assume that friction is applied at a constant level during a short time period.

Summary

This is the first of three chapters that deal with some slightly less common topics in mechanics. These topics involve situations that you're unlikely to encounter commonly, but from time to time you do need to understand how they work if you are to employ them in your programs. In light of this, few equations and only the most generic code examples have been provided.

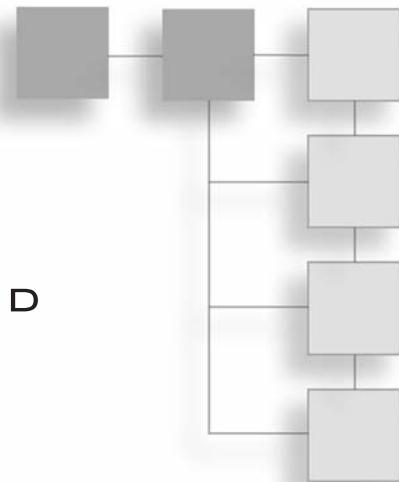
Over the whole of this chapter, perhaps the most important thing to take away with you is a general understanding of how friction works and applies in different circumstances. In this respect, explorations of the pool ball prepare you for working in at least one game context. The next chapter pursues still another set of supplementary topics, concluding your exploration of energy and momentum. There you will look at continuous momentum and tension.

You Should Now Know

- Why friction occurs and how to calculate it
- The meaning of the term *coefficient of friction* and the difference between *static* and *kinetic* friction
- What *drag* is and how it is calculated
- How friction converts between linear and angular momentum
- Why moving objects fall over

CHAPTER 15

STRINGS, PULLEYS, AND CONVEYOR BELTS



In This Chapter

- Overview
- Pulling Things Around
- The Inextensible String
- Continuous Momentum

Overview

In this chapter, you'll look at some special examples of calculations with energy and momentum, with the slightly tenuous link that they all involve objects that are connected in some way. The first section looks at how you deal with the situation when two objects are tied together with an idealized string and why using a pulley makes it easier to lift something. In the second part, you look at examples of momentum changes due to continuous changes of mass, such as loading a conveyor belt or burning fuel in a rocket.

Pulling Things Around

When you attach a string to something, you can make it move without touching it if you pull on the string. This might not seem very interesting at first, but when you think about it, something strange is occurring. Somehow, the force you exert on the string is being transferred down the string to affect the object at the other end. This is due to a force called *tension*.

The Inextensible String

In this section, you deal with idealized objects called “light inextensible strings.” These are somewhat similar to the idealized levers you encountered a couple of chapters ago. They have no mass and are completely without any elasticity. (Elasticity is a topic covered in Chapter 16.) Such strings are impossible to tangle.

The only property that light inextensible strings have is a length l . If two objects are tied with a string, then they can’t be more than a distance l apart. They can be moved nearer to each other, however. When taut, a string experiences the force of tension. As illustrated by Figure 15.1, the tension of a string acts in both directions and at all points on the string. In particular, an object tied at either end of the string experiences the same force T from the string.

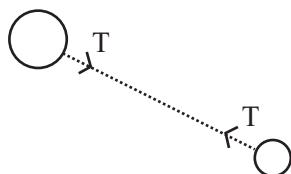


Figure 15.1

Two bodies experiencing tension from a taut string.

If you pull on one end of the string, exerting a force on it, then your action creates a tension. The tension extends Newton’s third law. If you exert a force on the string, it has to exert the same force on you. The result is that your force is “transmitted” to the other object. Of course, if you tug on the string and stop, rather than pulling continuously, you’ll start the other object moving, and the string will cease to be taut.

For ideal strings, length makes no difference to any of this, but it does have effects that are worthwhile to study. These will be dealt with in Chapter 16 and involve such topics as how strings swing like pendulums.

Strings on Tables

From the point of view afforded by physics, the basic string is not very interesting. Still, strings are useful because they allow you to change the direction in which a force acts. For example, as shown in Figure 15.2, a string ties two boxes together, one hanging off the edge of a table. Although bent, the string still represents a single force.

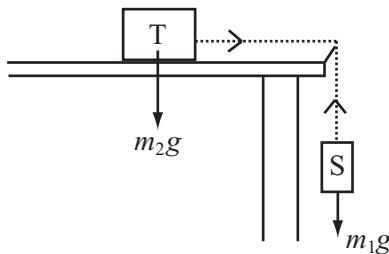


Figure 15.2

A string hanging off a table.

In Figure 15.2, the suspended box S is experiencing a force due to its weight. This creates a tension in the string. The tension is transmitted through the string to exert the same force on box T. Because of this tension, box T accelerates. What's more, because the string is inextensible, you know that both boxes must accelerate at the same rate. This gives you two equations to use with Newton's second law:

$$\text{Tension} = m_2a$$

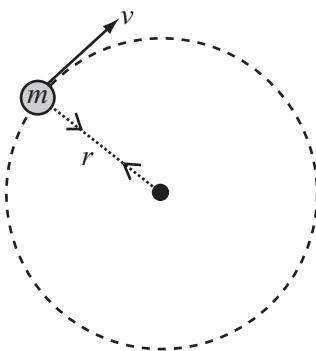
$$m_1g - \text{Tension} = m_1a$$

Eliminating the tension from these equations, you can see that $(m_1 + m_2) a = m_1g$. In other words, the two objects together accelerate as if they were both experiencing the weight of S.

You can also consider friction. Box T experiences friction from the table, and this depends only on the weight of T. If they are stationary, then the static friction on T is $\mu_s m_2 g$. Also, if they are stationary, then you know that box S is in equilibrium, so the tension in the string is equal to its weight. Since this tells you that $\mu_s m_2 g \geq m_1 g$, you can conclude that $\mu_s m_2 \geq m_1$. Equivalently, if this inequality is false, then the boxes cannot remain in static equilibrium.

Strings and Circular Motion

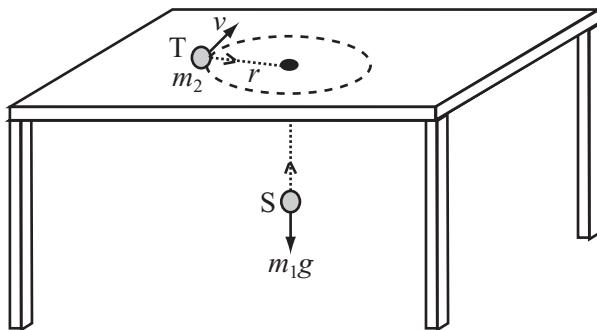
Figure 15.3 illustrates another situation in which you encounter strings. If an object is attached to a string and you swing it around yourself, then you can make the object fly in a circle.

**Figure 15.3**

Using a string to create circular motion.

As was mentioned in Chapter 12, to make an object move in a circle, you must apply a constant centripetal force toward the center of the circle. The string provides this force through its tension, and in turn it exerts this force on the person holding the string. The person holding the string then experiences a centrifugal force. You calculated this force to be $\frac{mv^2}{r}$, where v is the speed, r is the radius of the circle (in this case, the length of the string), and m is the mass of the orbiting body.

Now picture the situation as shown in Figure 15.4. Here, a string is passed through a hole in the table. Suppose the ball T on the table is spun around with a speed v . If you then release the ball S hanging under the hole, this exerts a downward force of m_1g . Meanwhile, ball T is exerting a force of $\frac{m_2v^2}{r}$, where r is the length of string from the hole to T.

**Figure 15.4**

A string passing through a table.

If the force due to S is greater than the force due to T, then the balls will sink farther through the hole. The sinking decreases r , but since angular momentum m_2vr must be conserved, v must increase proportionally. The result is that you now have a radius $\frac{r}{k}$ and a speed kv for some $k > 1$, giving you a new force of $\frac{km_2v^2}{r}$. As the ball sinks farther into the hole, the force due to the circular motion increases, while the force due to the weight stays constant. Eventually, the two forces even out at a stable orbit, with $\frac{m_2v^2}{r} = m_2v$.

There is an infinite number of such orbits, depending on the initial angular momentum of T. In practice, friction will slow the spinning ball down too fast for you to observe this phenomenon clearly, however. Still, the scenario that unfolds with respect to Figure 15.4 provides a good example of the same principle that applies to gravitational orbits. In fact, mathematically, like the ball S hanging through the hole, it's similar to the interpretation of gravity as seen in general relativity, where an object's gravitational field is represented by a curvature of space due to its mass.

Pulleys

In the real world, one of the principal uses of ropes (which are generally thick strings) involves creating a pulley system, such as a block and tackle. Figure 15.5 illustrates the simplest kind of pulley. Such a pulley is a variant of the “box on a table” scenario discussed previously. It provides a means of changing the direction in which a force is exerted. Lifting an object with this system requires exactly the same amount of effort as lifting it without using the string at all. The difference is that you don't have to climb to the top of the building to do it.

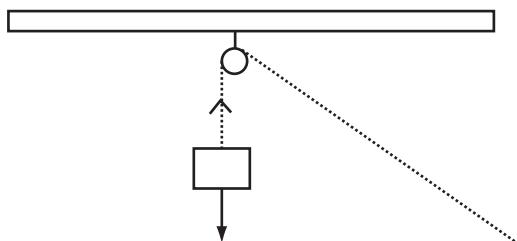


Figure 15.5
A simple pulley.

A more useful kind of pulley system is shown in Figure 15.6. This configuration features two pulleys. One holds the object to be lifted; the other is attached to a support positioned above it, as was done with the previous example. In addition, however, one end of the rope is attached to the support above the object, and given this feature, the pulley changes in a significant way. Both ends of the rope now exert force. Force is exerted by both ends because you exert a force F by pulling on the loose end of the rope, but it also happens that the ceiling exerts a corresponding reaction force F on the attached end of the rope. Together, two forces combine to create a force of $2F$ on the object, doubling the lifting force.

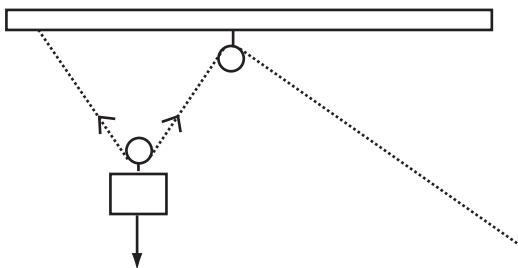


Figure 15.6
A two-pulley system.

How the lifting force is doubled might seem to present a contradiction. It remains, however, that regardless of the number of pulleys used, the same amount of energy is required to lift the object a particular distance. The key to understanding how this is so requires only that you consider the *work* being done. Although you are lifting the object with twice the force, you move it only half as far. You require twice as much rope to make it lift the same distance. Thus the amount of work done, and hence energy used, is the same.

At least until the friction in the pulleys becomes significant, by adding more pulleys, you can make the force as small as you like. For example, Figure 15.7 illustrates four pulleys at work. The amount of rope used for them is greater, cutting increasing force but preserving the amount of work. What applies to pulleys is analogous to gearing, which was discussed in Chapter 14. With gearing, you trade force for distance or speed.

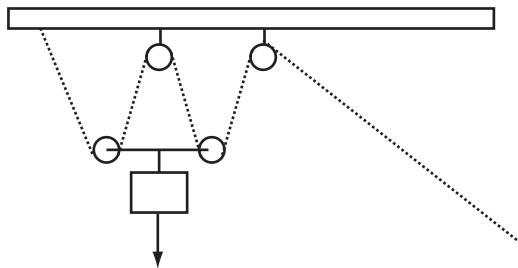


Figure 15.7
A four-pulley system.

Note

In practice, the arrangement of pulleys shown in Figure 15.7 is highly unstable, and for this reason, the pulleys at the bottom would be replaced by a single pulley with the rope wrapped around it in two separate loops

Continuous Momentum

Chapter 7 provided a brief discussion of rockets and asked how it might be possible to account for the fact that the masses of rockets change as they burn up fuel. The question of how this happened was left open. In this context, it now becomes appropriate to again address it. How can the decrease in mass be calculated?

Conveyor Belts

To account for how the mass of a rocket decreases as the rocket ascends, you can start by looking at a problem that involves the opposite situation. Suppose a conveyor belt is being loaded with a fine granular substance, such as sand, at a rate of k kg per second. Assume that the sand is loaded gently, so that the grains don't roll about. If your conveyor belt is traveling at a constant speed v , how much power does it need?

Apart from friction in the belt, the length of the conveyor belt does not matter, and after the sand is moving, keeping it moving is not difficult because of Newton's first law. This means that the only question you're interested in is how much energy is being given to the sand as it is loaded. To answer the question, consider that each second, k kg of sand is accelerated to a speed of v , so its kinetic energy goes from zero to $\frac{1}{2}kv^2$. This means that the power of the conveyor has to be $\frac{1}{2}kv^2$.

What about the force exerted? With the aid of the power calculation, this is not difficult: since $power = force \times velocity$, the force is $\frac{1}{2}kv$.

You can use the same kinds of calculation to deal with escalators. An escalator closely resembles a conveyor belt, with the difference that instead of moving things horizontally, it also lifts them up. Since it depends on how long the escalator is, the fact that something is lifted makes the calculation a little more complicated.

Figure 15.8 shows an escalator in action with a speed of v , a horizontal length l and a vertical height h . Although it is usually the case that people step onto an escalator at uneven intervals and place all their weight on it with one or two steps, to perform your calculations, assume at first that the escalator is fully loaded, and that people walk onto it at a constant rate of k kg per second.

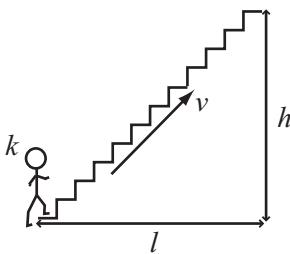


Figure 15.8

An escalator.

Proceeding, your next action is to define the length d of the escalator, which is $\sqrt{l^2 + h^2}$. Using this value, you can calculate the weight of the escalator. It takes a person a time $\frac{d}{v}$ to reach the top of the escalator. During this time, $\frac{kd}{v}$ kg will join the escalator, which means that this is always the mass of people on the stairs. Since each second the riders go up a distance of $\frac{hv}{d}$, the potential energy they are given is $\frac{kd}{v} \times g \times \frac{hv}{d} = kgh$.

A couple of further considerations are needed, however. First, you haven't taken into account the mass of the escalator itself. Likewise, it is helpful to consider kinetic energy.

Since the kinetic energy added per second is $\frac{1}{2}kv^2$, the total power is $\frac{1}{2}kv^2 + kgh$, and the force is $\frac{1}{2}kv + \frac{kgh}{v}$. Notice that the horizontal length of the escalator cancels out in the end.

It's only the vertical height that is relevant. You can see how this is so if you consider that a conveyor belt is just a special case of an escalator with height 0.

Maintenance engineers often complain about the additional wear and tear on the machinery due to people walking up escalators. If people choose to walk up an escalator, then the escalator has to exert the force necessary to propel them faster. This means it needs greater power. On the other hand, increasing the velocity decreases the time during which the riders are on the escalator, which decreases the total mass being lifted. The result of this action is reflected in the second term of the force equation. So although the power needed is greater, the force may not be. When $v > \sqrt{2gh}$, the force starts to decrease as the velocity increases.

If the escalator riders walk down a downward escalator, the same kinds of calculations apply. However, in this case, the power used is less, and up to a certain speed, the force will decrease. After that, the force required is higher.

Rocket Fuel

To return to the discussion of rockets, dealing with rockets is a little like dealing with an escalator when people stop getting on. As they reach the top, they step off the escalator. As they step off, they decrease the mass being raised. A rocket is much the same. Because it uses up fuel as it goes, the longer it travels, the lower its mass.

Suppose a rocket burns fuel at a constant power P . Likewise, suppose that it uses up fuel at a rate of k kg per second. If it starts with a mass of m_0 and travels for a time t , how fast will it be traveling at the end? Answering this question involves differential equations, which are outside the scope of this book. Still, in general terms, the equation required to answer the question makes use of the fact that the force on the rocket results from the ejection of material through its engines. If you know the speed at which the material is ejected and the mass of the fuel, then you know its momentum, and just as with the conveyor belt, you can determine the power and the force. Conversely, if you know the power, then you can calculate how fast it is ejected. Each second, you give k kg of fuel an energy of P , so its velocity is $\sqrt{\frac{2P}{k}}$.

The rocket equation is based on this velocity, which you'll call u . Given this understanding, if the rocket starts with a velocity of v_0 and moves under constant gravity, its velocity is given by

$$v_0 + u \log\left(\frac{m_0}{m(t)}\right) - gt$$

This applies to the fuel burned. For more specific information, you can plug in the values of this particular problem to get

$$v = v_0 + \sqrt{\frac{P}{2k}} \log\left(\frac{m_0}{m_0 - kt}\right) - gt$$

To find how high it will travel in this time, you need to integrate the rocket equation.

As a general rule, these equations get very complicated very fast, and you're better off using numerical methods. In other words, simply calculate the rocket's position from moment to moment. Here's a set of functions that will do this. They make certain assumptions, such as a constant gravitational field. Exercise 15.1 poses the challenge of including a changing gravitational field.

The `currentPosition()` function takes the parameters mentioned in the preceding discussion and returns the current position:

```
function getRocketPosition(currentPosition, currentSpeed,
                           currentMass, massBurnRate,
                           fuelVelocity, gravity, time)
    set MassBurned to massBurnRate * time
    set speed to getRocketSpeed(currentSpeed, currentMass,
                                massBurned, fuelVelocity,
                                gravity, time)
    return currentPosition + speed * time
end function
```

Supplementing the calculations of the `currentPosition()` function, you make use of the `getRocketSpeed()` function:

```
function getRocketSpeed(currentSpeed, currentMass,
                        massBurned, fuelVelocity,
                        gravity, time)
    return currentSpeed + fuelVelocity *
        log (currentMass/(currentMass-massBurned))
        - gravity * time
end function
```

Exercise

EXERCISE 15.1

Modify the `getRocketSpeed()` and `getRocketPosition()` functions to deal with a varying gravitational field. Your revised functions should calculate the acceleration due to gravity at a particular point and adjust accordingly. If you feel even braver, try adjusting them to deal with velocity in any direction, rather than only vertical.

Summary

This chapter has tied up a few loose ends. Among other things, it clarifies the concepts of power and work. It has also provided examples of more calculations involving force, energy, and momentum. In the process, you have had a look at the basics of calculations involving rockets in flight.

In the next chapter, you'll complete your exploration of the more obscure aspects of mechanics by looking at springs, elastics, and pendulums. These turn up more often than you might think.

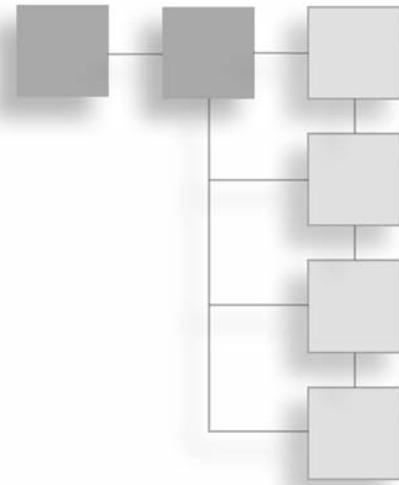
You Should Now Know

- The meaning of the word *tension* as it applies to a string, and how it acts
- How to calculate the motion of objects tied together by a string
- The action of a *pulley* as related to force and energy
- How to calculate motion of objects going on to a conveyor belt or escalator
- How to calculate the motion of a rocket burning fuel

This page intentionally left blank

CHAPTER 16

OSCILLATIONS



In This Chapter

- Overview
- Springs
- Simple Harmonic Motion
- Damped Harmonic Motion
- Complications of Springs
- Calculating Spring Motion
- Waves

Overview

Chapter 15 dealt with inextensible strings. Now you're going to see how the behavior of a string changes when it can stretch. In particular, you're going to look at how a particle moves when it is attached to a *spring*. One specific manifestation of this is the bouncing motion referred to as *oscillation*. You'll see how in nature the same motion occurs in many circumstances, and then you'll create a function describing a complex spring that has both extensible and inextensible properties. Finally, you'll see how these concepts extend to help you deal with waves and explain some of the properties of light.

Springs

When you use the word *spring*, you have in mind the same kind of idealization that applied to inextensible strings in the previous chapter. A spring is a light string with a certain natural length l that can be stretched to a greater length. When stretched, the spring experiences a tension, which acts equally in both directions. These calculations apply to both elastic bands and to actual springs, and since the term “elastic” has another meaning in the context of collisions, you’ll use the term *spring* alone to refer to both.

The Force in a Stretched Spring

The tension in a spring can be calculated very simply. To describe how stretchy it is, a spring has a value called the *coefficient of elasticity*, k . The tension in a stretched spring is then proportional to its *extension*, and the extension is the difference between its current length and its length when it is not stretched. The value of this difference is given by Hooke’s law:

$$\text{Force} = -k \times \text{Extension}$$

Why the negative sign? This is conventional because the force is always directed backward, toward the unstretched “equilibrium” length. As shown in Figure 16.1, if a particle is attached to the end of the spring, it experiences the tension backward along the length of the spring.

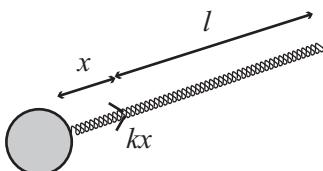


Figure 16.1

A particle on the end of a spring.

Some springs are only extensive. In other words, they experience a tension only when stretched outward. While an extensive spring becomes something more like a rigid rod, an elastic band has no tension when unstretched. Other springs are *compressive*. Compressive springs experience a tension outward when their length is less than the standard length. One generalization you can draw from observing extensive and compressive springs is that a negative extension leads to a positive tension.

When a spring is under tension (compressive or extensive), it contains energy, known as *elastic potential energy*. It takes work to stretch a spring, and the energy is released when it is allowed to bounce back. The energy is given by

$$\text{Energy} = \frac{1}{2} \times k \times \text{extension}^2$$

From this and the equation for force, you can see that the coefficient of elasticity has units of kg s^{-2} .

There is one other facet of real-life springs that is important to consider. Real-life springs cannot be stretched indefinitely. Instead, they reach a length called the *elastic limit*. Beyond this point, the coefficient of elasticity increases significantly, making them much harder to stretch. In addition, after they have reached their elastic limit, springs will no longer return to their original length when released. At an extreme, springs can be stretched to the point that even the bonds between their constituent molecules start to break. At this point, the springs are likely to snap. For the current context of discussion, you can simply say that beyond the elastic limit, springs start to act like inextensible strings, with constant inward tensions.

Using Springs to Measure Weight

Figure 16.2 illustrates the canonical example of springs in action. Here, you have an object with mass m attached to one end of an extensive spring with unstretched length l . The other end of the spring is attached to a supporting beam or the ceiling.

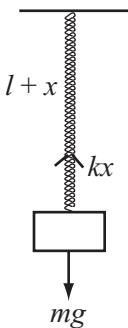


Figure 16.2

A particle hanging from a spring.

If the object hanging on the spring is in equilibrium, you can say that the tension in the spring must be equal to its weight. This is expressed as follows:

$$k \times \text{extension} = mg$$

$$\text{extension} = \frac{mg}{k}$$

Since for a particular spring in constant gravity the values of g and k are constant, the extension of a spring is directly proportional to the mass of the object hanging on the spring. For this reason, you can use a spring to measure something's weight.

If the object is somewhere other than in the equilibrium position, with an arbitrary extension x , then the force on it is equal to $mg - kx$. This difference describes a different situation. In this situation, the object is moving horizontally under the action of a spring that is both compressive and extensive. The unstretched length of this spring is given by the expression $l + \frac{mg}{k}$, the equilibrium length of the first spring. This value is useful since it allows you to ignore gravity when calculating the motion of the object attached to the spring.

Simple Harmonic Motion

When you pull an object hanging from a string downward from the equilibrium position and release it, it bounces up and down. The up and down motion constitutes an oscillation. Oscillations turn out to have a common characteristic, which is called *simple harmonic motion*, or SHM.

The Equation of SHM

To calculate the equation for simple harmonic motion, you can begin by looking at the formula for Hooke's law. By applying Newton's second law, you see that

$$ma = -kx$$

where x is the extension, m is the mass, and a is the acceleration. All of these quantities must be measured in the same direction. Here is another version, given as a differential equation:

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x$$

As you have seen (in Chapter 6 and elsewhere), differential equations require a great deal of work. In this case, it turns out that a simple function solves it. You've already encountered two functions whose second derivative is their own negative. These are $\sin()$ and $\cos()$. Given these two functions, recall that

$$\frac{d}{dt}\sin t = \cos t; \frac{d}{dt}\cos t = -\sin t$$

so

$$\frac{d^2}{dt^2}(\sin t) = -\sin t$$

$$\frac{d^2}{dt^2}(\cos t) = -\cos t$$

It takes only a little tweaking to adapt these functions to create a general formula that solves the differential equation for SHM:

$$x = A \sin(\omega t) + B \cos(\omega t)$$

where A and B , are arbitrary constants and ω is equal to $\sqrt{\frac{k}{m}}$. Another useful way of presenting this relationship is as follows:

$$x = C \sin(\omega t + p)$$

where C and p are arbitrary constants. You'll be using this form of the equation here, although both forms are common.

Note

It is simple to convert from one form of the equation to the other. For example, given that

$$\sin(\omega t + p) = \sin(\omega t) \cos p + \cos(\omega t) \sin p$$

you have two equations,

$$A = C \cos p$$

$$B = C \sin p$$

You can calculate the velocity of the object attached to a string at a particular time by differentiating the SHM equation, which gives you

$$\dot{x} = C\omega \cos(\omega t + p)$$

Differentiating again gives you

$$\ddot{x} = -C\omega^2 \sin(\omega t + p) = -\omega^2 x$$

as required.

As usual with a differential equation, you have a family of equations that all serve as valid solutions. Any one of these provides a possible motion for the object on the spring. By setting the *initial condition*, you choose which to use. In other words, you choose what the object is doing at the start. If you pull it down by a certain distance d and then release it, then you will get $C = d$, $p = 0$. If you give the object on the spring a push from equilibrium, so that it has an initial velocity v , you get $C = \frac{v}{\omega}$, $p = \frac{\pi}{2}$. You'll look at these calculations more in a moment.

So what does all this mean? What does the motion look like? One answer is that when $t = \frac{p}{\omega}$, since $\sin(0) = 0$ and $\cos(0) = 1$, the particle has the position 0 and velocity $C\omega$. If C is positive, the extension gradually increases, until at time $t = \frac{1}{\omega} \left(\frac{\pi}{2} - p \right)$, when it reaches a maximum of C . Then it decreases to 0 again and goes back the other way, before returning to 0 at $t = \frac{1}{\omega} (2\pi - p)$.

As shown in Figure 16.3, the result of this action is the generic sine wave discussed in Chapter 4. The time $\frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}}$ is called the *period of the motion*. The period of the motion is the time required for one complete oscillation. The value ω is called the *frequency*. The value p is called the *phase of the motion*. The phase of the motion is how far the waveform is shifted from zero along the time axis. And C is called the *amplitude*, or *maximum displacement from equilibrium*.

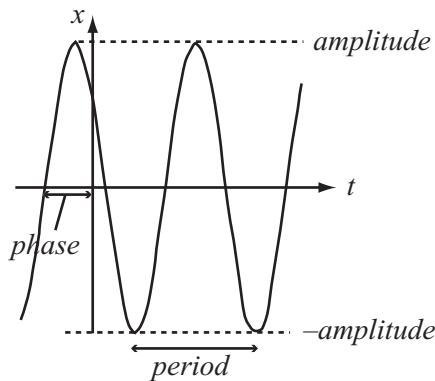


Figure 16.3

The position of a particle over time under SHM.

Other Examples of SHM

As you might recall, the position of a particle when moving along a sine wave is equivalent to the y -coordinate of a point on the circumference of a turning wheel. This means that a point on a wheel moves under SHM. In fact, as Figure 16.4 illustrates, the values C and p relate to this interpretation. C is the radius of the wheel, and p is a measure of how far around the wheel the object is located or has traveled.

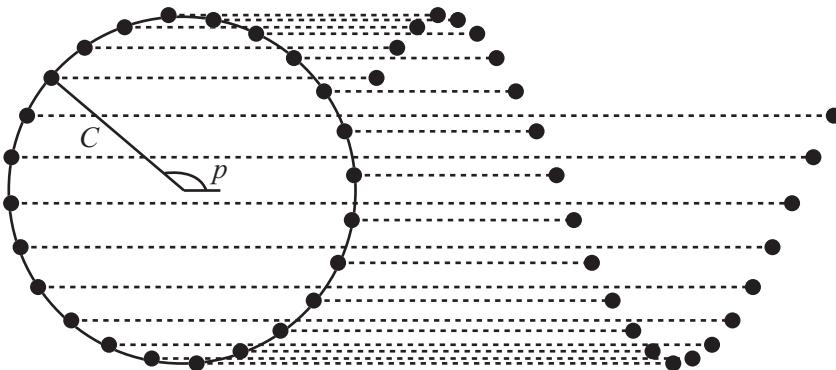


Figure 16.4

SHM and circular motion.

You take advantage of this when using circular motion to drive pistons. When you attach a rod to a point on a wheel and allow it to slide up and down as the wheel turns, its tip approximately oscillates under SHM.

SHM also occurs in the motion of a pendulum. Again, it is an approximation and only works when the oscillations are small. When an object is attached to a pendulum by a light inextensible string of length l at an angle θ , it experiences a force W downward due to its weight and a tension T from the string. In particular, the radial force on the object (which provides torque) is $-W \sin \theta$. The minus sign is used to indicate that the force is directed in the opposite direction to the angle. As noted before, for small values of θ —oscillations of about 5° — $\sin \theta$ is very close to θ , so the torque acting on the particle is approximately $-Wl\theta$. This gives you another example of Hooke's law:

$$\begin{aligned} \text{Torque} &\approx -Wl\theta = -mgl\theta \\ \text{Angular Acceleration} &\approx \frac{-mgl\theta}{\text{moment of inertia}} = \frac{-mgl\theta}{ml^2} = \frac{g\theta}{l} \end{aligned}$$

In this case, the mass of the particle cancels out, and the frequency depends only on the length of the string. Consequently, the period is proportional to the square root of the length.

Other examples of SHM in action include a buoy floating on wavy water, the vibration of a plucked guitar string, the vibrations of atoms in a crystal, the variation of an alternating current, and the variation in a population of animals over time. Whenever you find a situation where there is an equilibrium position and a force that is exerted continuously to try to restore that equilibrium, you will find SHM.

It's also worth noting that waves themselves are formed of components moving under SHM. A light wave consists of oscillating electric and magnetic fields. The strength of these fields at any particular location varies over time in an SHM pattern. Each oscillation induces an oscillation next to it, and the next oscillation lags a little behind. The result is that the situation looks like a sine wave traveling forward over time. This phenomenon will be examined further later in the chapter.

Calculating the Parameters

To return to the question of the parameters C and p , as was discussed previously, C represents amplitude and p represents phase of the motion, respectively. While a particular mass on a particular spring will always oscillate with the same frequency, the amplitude and phase vary from situation to situation, according to how fast the particle is put into motion and how far away from equilibrium it is released.

The easiest way to calculate these values is to know the velocity v and extension d at time 0. To determine the parameter values, you can use the velocity and position formulae of SHM as simultaneous equations:

$$d = C \sin(p)$$

$$v = C\omega \cos(p)$$

so

$$\frac{d}{v} = \frac{\tan(p)}{\omega}$$

$$\omega^2 d^2 + v^2 = C^2 \omega^2$$

which is to say,

$$p = a \tan\left(\frac{\omega d}{v}\right)$$

$$C = \frac{\sqrt{\omega^2 d^2 + v^2}}{\omega}$$

More generally, you can do the same calculations for any time t : the calculation for C remains unchanged while the calculation for p merely changes to

$$p = a \tan\left(\frac{\omega d}{2v}\right) - \omega t$$

Damped Harmonic Motion

Real life is not as simple as SHM would have you believe. Most significantly, real oscillations don't go on forever. Instead, they lose energy over time. The loss of energy over time is called *damping*. It is not difficult to take the damping into account to create more realistic and varied motion.

The Equation of DHM

Damped harmonic motion (DHM) is a slight modification to the SHM equation. You add a new damping factor to the differential equation. This factor is proportional to the velocity rather than the acceleration:

$$\ddot{x} = -\omega^2 x - 2D\dot{x}$$

Note

In this book, the coefficient $2D$ represents the damping factor. It is used to simplify later calculations. It is more conventional, however, to use the letter b to represent this coefficient.

Solving this differential equation is a little more work than before. It involves using a “trial solution,” which leads to a particular quadratic equation. Leaving out the details, the end result is as follows:

$$x = Ae^{-rt}$$

where

$$r = -D \pm \sqrt{D^2 - \omega^2}$$

Depending on the values of D and ω , this equation renders different results. If you define the variable α to be $D^2 - \omega^2$, if $\alpha > 0$, this equation has real solutions, and you end up with a family of equations of the form

$$x = Ae^{(-D-\sqrt{\alpha})t} + Be^{(-D+\sqrt{\alpha})t}$$

When $\alpha = 0$, so that $D = \omega$, while the motion is similar to what has been previously described, only one value of r occurs. The result is that you get a slightly simpler equation:

$$x = (A + Bt) e^{-Dt} = (A + Bt) e^{-\omega t}$$

If $\alpha < 0$, then the equation has no real roots, and you end up with a complex number. This is not the place to discuss complex numbers, but it turns out that you can still solve the resulting differential equation by using the imaginary number i . This number, by definition, equals the square root of -1 . Using this approach, you can derive this formula:

$$x = C \sin(\varphi t + p) e^{-Dt}$$

where

$$\varphi = \sqrt{-\alpha}$$

While harder to discover, this equation resembles the SHM equation discussed previously, but there are two main differences. The first is that it has the additional exponential term, which is negative. As a result of this, the amplitude of the motion decreases over time. The second is that the frequency is different. The greater the value of D , the lower the frequency. This continues until D reaches the critical damping value where $D = \omega$. Above

that value, you go into the first kind of behavior, and the frequency is essentially zero. In other words, rather than oscillating, the object follows an exponential curve. In sum, then, there are three “zones” of behavior for DHM:

- **Underdamping.** This resembles SHM but decreases exponentially in amplitude: $\alpha < 0$.
- **Critical damping.** Only one value of r occurs: $\alpha = 0$.
- **Overdamping.** Exponential decrease with no oscillation: $\alpha > 0$

Figure 16.5 illustrates the different behaviors of these three forms of damping.

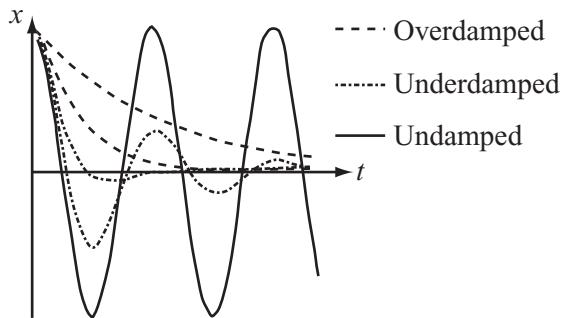


Figure 16.5

Possible motions of a particle under different levels of damping. Each starts with the same initial conditions, and all the springs have the same coefficient of elasticity.

Damping in Practice

As was done with previous equations, you can differentiate each of the motion formulas to get the velocity function.

For underdamping, you arrive at the following:

$$\begin{aligned}\dot{x} &= C\varphi \cos(\varphi t + p) e^{-Dt} - CD \sin(\varphi t + p) e^{-Dt} \\ &= C\varphi \cos(\varphi t + p) e^{-Dt} - Dx\end{aligned}$$

For critical damping, the result is as follows:

$$\begin{aligned}\dot{x} &= Be^{-\omega t} - \omega(A + Bt)e^{-\omega t} \\ &= (B - \omega(A + Bt))e^{-\omega t} \\ &= Be^{-\omega t} - Dx\end{aligned}$$

(Remember that for critical damping, $D = \omega$).

And for overdamping, you get this result:

$$\dot{x} = A(-D - \sqrt{\alpha})e^{(-D - \sqrt{\alpha})t} + (B - D + \sqrt{\alpha})e^{(-D + \sqrt{\alpha})t}$$

Expressed more simply, differentiating overdamping appears this way:

$$\dot{x} = Ar_1e^{r_1 t} + Br_2e^{r_2 t}$$

Applying damping involves little more than plugging values into these equations. But calculating the parameters is more detailed than before because the velocity function is more complicated.

When you work with the equations, recall that D is a known constant property of the spring, like the coefficient of elasticity, not a parameter like C and p . If you know the distance and velocity at time t as before, then you can work with them through a sequence of equations.

To start with, for underdamping, you have

$$\begin{aligned}d &= C \sin(\omega t + p) e^{-Dt} \\ v &= C\omega \cos(\omega t + p) e^{-Dt} - Dd\end{aligned}$$

so

$$\begin{aligned}\frac{d}{v + Dd} &= \frac{1}{\omega} \tan(\omega t + p) \\ d^2\omega^2 + (v + Dd)^2 &= C^2\omega^2 e^{-2Dt}\end{aligned}$$

and this gives you

$$p = a \tan\left(\frac{d\omega}{v + dD}\right) - \omega t$$

$$C = \frac{\sqrt{d^2\omega^2 + (v + Dd)^2}}{\omega e^{-Dt}}$$

With critical damping, you have this formulation:

$$d = (A + Bt) e^{-\omega t}$$

$$v = (B - \omega(A + Bt)) e^{-\omega t}$$

so

$$\omega d + v = B(\omega t + 1) e^{-\omega t}$$

$$(1 - \omega t) d - tv = Ae^{-\omega t}$$

If $t = 0$, then working with critical damping becomes even simpler:

$$A = d$$

$$B = \omega d + v$$

As for overdamping, you have:

$$d = Ae^{r_1 t} + Be^{r_2 t}$$

$$v = Ar_1 e^{r_1 t} + Br_2 e^{r_2 t}$$

so

$$r_1 d - v = B(r_1 - r_2) e^{r_2 t} = -2B\sqrt{\alpha} e^{r_2 t}$$

$$r_2 d - v = -A(r_1 - r_2) e^{r_1 t} = 2A\sqrt{\alpha} e^{r_1 t}$$

and this gives you

$$A = \frac{r_2 d - v}{2\sqrt{\alpha}} e^{-r_1 t}$$

$$B = \frac{r_1 d - v}{2\sqrt{\alpha}} e^{-r_2 t}$$

Again, if $t = 0$, you can use a simpler formulation:

$$A = \frac{r_2 d - v}{2\alpha}$$

$$B = \frac{r_1 d - v}{2\alpha}$$

Complications of Springs

While the previous sections provided a fairly comprehensive discussion of the physics of springs, it is worthwhile to look at a couple of important consequences of the equations. The first, *resonance*, is important when trying to build a simulation involving springs, for it can lead to instability. The second, *coupling*, is also important, for it means that the actions of springs can become extremely complex.

Resonance: Pushing the Swing

Everyone has heard stories about people who can use their voices to break glass. This isn't just a matter of producing particularly high notes that miraculously cause things to break. A physical phenomenon is behind it that is a major consequence of SHM.

Imagine a simple variation in the situation represented by Figure 16.2. This time, the top of the spring is attached to a vibrating rod. The vibrating rod moves it up and down at a certain *driving frequency* f with an amplitude of A . As becomes apparent after a brief discussion, the driving oscillation imparts a force to the system. Each time the bar moves up, it increases the tension in the spring. When it drops, it releases it. What then happens to the object suspended from the spring?

To answer this question, while the object bounces around rather erratically, some patterns emerge. In particular, if you change the driving frequency, the amplitude of the motion of the object increases as you get closer to the natural frequency of the spring. When you reach the natural frequency, the particle starts to go crazy. It bounces higher and higher without end. In fact, theoretically, it might bounce infinitely high. Then as you increase the driving frequency further, the particle calms down again.

There's a reasonably simple relationship between the amplitude C of the motion and the driving frequency. This relationship can be expressed as follows:

$$C = \frac{kA}{\sqrt{m^2(f^2 - \omega^2)^2 + 4D^2\omega^2}}$$

Here the value kA represents the maximum force exerted by the driving oscillation—the driving force.

Note

This formula applies when the driving oscillation is *sinusoidal*. As the adjective implies, a sinusoidal wave is in the shape of a sine wave, which is also the shape of waves associated with SHM. When the driving oscillation has some other pattern, the formula is different, but it differs only by a constant factor. The result is that the basic behavior is the same.

When the driving frequency is equal to the natural frequency ω , the first term in the square root drops out. In SHM, with $D = 0$, this means that C becomes infinitely large. Otherwise, C is inversely proportional to the damping factor. As a result, in systems with little or no damping, the oscillation starts to spiral out of control. Such spiraling is called *resonance*, and the natural frequency is called the *resonant frequency*.

You use this principle every time you push a child on a swing. At the topmost point of the swing, you apply a force. This force is naturally applied at the same frequency as the oscillation, with the result that the oscillation grows in amplitude and fun. Because a pane of glass also has a natural frequency, if you drive it with a sound wave of the same frequency, it vibrates so hard that it shatters.

All of this shows that when you create a spring system in a simulation, it's important to give it some damping factor, or at least an elastic limit. Otherwise, you might find things getting out of hand.

Coupling: Linked Motion

When you combine two springs together by means of another, the combined springs are said to be *coupled*. Coupling means that the motion of each spring is affected by the other. Energy is constantly being transferred between them. The result is that extremely complicated motion can result. There's no need to look into the mathematics in detail, but it's worth noting one particular result.

If two identical coupled springs are set in motion in the same phase, then they will swing in parallel at the natural frequency. This is true despite the fact that both are experiencing a tension due to the coupling spring. If they are set in motion exactly *out of phase*, meaning that one is lifted up as the other is pulled down and then they are both released at the same time, they will continue to oscillate out of phase. However, the frequency will be higher than before. These are called the two *natural modes* of the motion.

In general, any two coupled oscillators yield a system with two natural modes, each with its own frequency. The modes depend on the frequencies of the two oscillators and of the coupling spring. Any other motion of the system is actually a linear sum of two such oscillations. These concepts are very important in acoustics.

Calculating Spring Motion

No code has as yet been presented in this chapter. Now that you have examined the math and physics, however, you are prepared for the code. Here you will create three functions. The functions calculate the motion of a particle attached to an arbitrary spring. You'll give the spring a number of characteristics, such as coefficients of elasticity and length. In addition, the object on the spring has a mass, and you'll include an optional force of gravity.

Force Due to a Spring

The three functions are useful in different circumstances. The first is a pure mechanical system. With such a system, the function returns the force on the object on the spring due to the spring. In this case, you must need to apply gravity separately. This is the only method you can use in general circumstances, such as when neither end point of the spring is fixed in place. Here is the `forceDueToSpring()` function:

```
function forceDueToSpring(end1, end2, velocity1,
                           velocity2, springLength, elasticity,
                           damping, elasticLimit,
                           compressiveness, minLength)

// The object you're interested in is attached to end2
set v to end1-end2
set d to magnitude(v)
if d=0 then return vector(0,0)
// skip for this time-step if they coincide

// loose elastics have no force when compressed
if d<=springLength then
  if compressiveness="loose" then return vector(0,0)
end if
```

```

// apply second elastic limit (inextensible behavior)
if d>=elasticLimit*1.2 or d<=minLength*0.9
    or (d<=springLength*0.9
        and compressiveness="rigid") then
    return "bounce"
end if

// apply first elastic limit (increased force and damping)
if d>=elasticLimit or d<=minLength
    or (d<=springLength
        and compressiveness="#rigid") then
    multiply elasticity by 20
    set damping to max(damping*10,20)
end if

// calculate force by Hooke's law
set e to d-springLength
set v to v/d
if damping>0 then
    set vel to component(velocity1-velocity2,v)
    set f to damping*vel+elasticity*e
else
    set f to elasticity*e
end if
return f*v
end function

```

The only complicated part of the `forceDueToSpring()` function involves dealing with the elastic limit. Simulated springs are more complicated than real ones because you can get impossible situations, like a spring that is extended significantly beyond its elastic limit. This happens both by incorrectly setting up the simulation. This can happen if users are allowed to drag objects around. Such situations can be avoided. On the other hand, other problems arise, such as when resonance is coupled with a gradual accumulation of rounding errors.

To deal with this, you can create a tiered elastic limit system. Beyond the set elastic limit of the spring, you increase both the coefficient of elasticity and the damping coefficient. The effect of increasing the coefficient of elasticity is to create a strong force inward, which is important. But the damping coefficient is also necessary because it ensures that the system loses energy rapidly, which means that on the next oscillation it doesn't exceed the elastic limit.

You also create a second elastic limit, arbitrarily set at 1.2 times the first. If the spring is trying to extend beyond this point, you treat it as a collision with a solid wall perpendicular to the spring. This ensures that the spring can never extend past the second limit.

Note

You're not taking into account the principal aspect of the elastic limit, which is that after it is exceeded, the physical properties of the spring itself would be changed. Its natural length might increase, for example, or it might snap.

Undamped and Uncoupled Springs

The second method is designed for situations that are slightly simpler than those typical of the previous section. These situations often involve undamped, uncoupled springs. With such springs, the spring attaches a movable object to a fixed point. The object can move freely in all directions. For example, this method would be suitable for situations where a user can click and throw the object, or where the object is part of a system with collisions. In these cases, you can take advantage of conservation of energy to avoid the inevitable rounding errors that appear when dealing with forces applied individually at each time step. By knowing that the total energy of the particle is constant, you can calculate its speed at any moment as long as you know its position. The system can even deal with the situation when the “fixed” point is in fact moving under the user’s control. The `particleOnSpring()` function addresses undamped and uncoupled springs.

```
function particleOnSpring (end1, end2, speed,
                           direction, mass, totalEnergy,
                           springLength, elasticity,
                           compressive, timeStep, g)
  // Returns a list of position, speed, direction, and total energy.
  // totalEnergy can have the value "unknown",
  // in which case the function calculates it and returns it.

  set v to end1-end2
  set d to mag(v)
  set e to d-springLength
```

```
if totalEnergy="unknown" then
  // calculate energy
  set totalEnergy to mass*speed*speed/2
  if e>0 or compressive=TRUE then
    set epe to elasticity*e*e/2
    add epe to totalEnergy
  end if
  if g>0 then
    set gpe to mass*g*end2[2]
    subtract gpe from totalEnergy
  end if
end if

// calculate force
set f to vector(0, mass*g)
if e>0 or compressive=TRUE then
  if d>0 then
    add v*elasticity*e/d to f
  end if
end if

// calculate new position
set a to f/mass
set displacement to direction*speed*timeStep + a*timeStep*timeStep/2
set pos to end2 + displacement

// calculate new elastic energy
set newd to mag(pos-end1)
set newe to newd-springLength
if newe>0 or compressive=TRUE then
  set epe to elasticity*newe*newe/2
otherwise
  set epe to 0
end if
```

```

// calculate new kinetic energy and hence speed
set ke to totalEnergy-epet+mass*g*pos[2]
if ke<=0 then // NB: for safety
    set speed to 0
otherwise
    set speed to sqrt(2*ke/mass)
    set velocity to norm(displacement)
end if

return Array(pos,speed,velocity,totalEnergy)
end function

```

Pure DHM Oscillation

Your final function is the simplest. It addresses pure damped harmonic motion oscillation (or SHM if damping = 0). Feed it an initial position and velocity and a time t , and it calculates the position and velocity at that time. This calculation is performed through a set of three functions. The primary function is the `calculateDHMparameters()` function. Since it is wasteful to calculate the parameters more than once, the best approach is to store them. The first function calculates the parameters and the form of the motion. The other two calculate the actual values when fed with the results of the first.

```

function calculateDHMparameters(initialPos, initialVel,
                                  elasticity, damping)
    set omega to sqrt(elasticity)
    set d to damping/2
    set alpha to elasticity-d*d
    if d=0 then
        set p to atan(omega*initialPos/initialVel)
        set c to sqrt(elasticity * initialPos *
                      initialPos + initialVel * initialVel)/omega
    return array("SHM", p, c)
    else if d<omega then
        set v to initialVel + d * initialPos
        set p to atan(initialPos * omega / v)
        set s to initialPos * initialPos * elasticity + v * v
        set c to sqrt(s)/omega
    return array("UnderDamped", p, c, sqrt(-alpha))
    else if d=omega then
        return array("Critical", initialPos, omega *
                      initialPos + initialVel)
    end if
end function

```

```

else
    set sq to sqrt(alpha)
    set r1 to -d-sq
    set r2 to -d+sq
    set a to (r2*initialPos - initialVel)/(2*sq)
    set b to -(r1*initialPos - initialVel)/(2*sq)
    return array("OverDamped", a, b, r1, r2)
end if
end function

```

Given use of the calculateDHMparameters() function, you then call the getOscillatorPosition() function, which is defined as follows:

```

function getOscillatorPosition (elasticity, damping, params, time)
    set omega to sqrt(elasticity)
    set d to damping/2

    if params[1] is
        "SHM": return params[3] * sin(omega * time + params[2])
        "UnderDamped": return params[3] *
            sin(params[4] * time + params[2]) * exp(-d * time)
        "Critical": return (params[2] + time * params[3]) * exp(-d*time)
        "OverDamped": return params[2] * exp(params[4]*time +
            params[3] * exp(params[5]*time))
    end if
end function

```

Having called both the calculateDHMparameters() function and the getOscillatorPosition() function, you can then call the getOscillatorSpeed() function. This function is defined as follows:

```

function getOscillatorSpeed(elasticity, damping, params, time, pos)
    // determine pos before running this function
    set omega to sqrt(elasticity)
    set d to damping/2

    if params[1] is
        "SHM": return params[3] * omega * cos(omega *
            time + params[2])
        "UnderDamped": return params[3] * omega * cos(params[4] *
            time + params[2]) * exp(-d * time) - d*pos

```

```

"Critical": return params[3] * exp(-d*time) - d*pos
"OverDamped": return params[2] * params[4] *
               exp(params[4]*time] + params[3] *
               params[5] * exp(params[5]*time)
end if
end function

```

All three of these functions represent little more than the application of the equations shown previously for SHM and DHM. It is important, however, to notice that the variables used to pass the values from function to function are global, so the three are interdependent.

Waves

When coupled oscillators are joined together in a row, an interesting phenomenon occurs: an oscillation induced at one end can transfer its energy to the next in line, so the energy is gradually moved through the system, like a Newton's Cradle. The resulting cascade of oscillations is called a *wave*.

Wave Motion

A wave is a set of individual coupled oscillators. Each performs some oscillation and so induces an oscillation further down the line, slightly out of phase with the previous oscillation. Showing the oscillations at a frozen moment in time, Figure 16.6 illustrates the result.

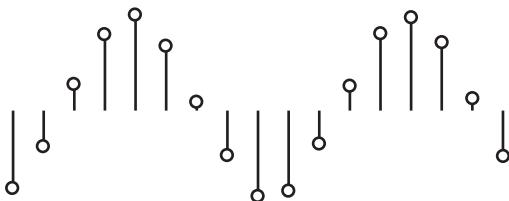


Figure 16.6

A series of coupled oscillations

As you can see in Figure 16.6, the silhouette of this picture is a sine wave. As it is, the shape of the wave does not always follow a sine pattern, but whatever the shape, the oscillations are copies of the driving oscillation at one end of the wave.

That waves are generated by a driving oscillation gives rise to a second way of picturing a wave. According to this view, the wave is a moving *waveform*. A waveform is an object moving at a certain velocity. To describe the waveform as an object is not quite accurate, however. Rather than an object, it is a kind of virtual object. It is a packet of energy transmitted through some medium.

Given the view provided by the waveform, the velocity of a wave is the distance traveled over time by the waveform. This value is determined by the physical properties of the medium. For a sine wave, you can measure the position of successive *wavefronts* over time. A wavefront is a peak of the waveform.

Once you know the speed v of the wave and its frequency f , you can calculate the distance between successive *wavefronts*. This value is referred to as the *wavelength*, denoted λ . This gives you a simple equation relating the three quantities: $v = f\lambda$.

Types of Wave

There are two principal kinds of wave, *transverse* and *longitudinal*. A transverse wave is the kind pictured in Figure 16.6. With a transverse wave, the oscillators are aligned perpendicularly to the direction of travel of the wave. Among such waves are water waves and electromagnetic waves. Electromagnetic waves are also known as light.

Longitudinal waves are a little harder to picture. The simplest example is a coiled spring (such as a Slinky, the children's toy). When the string is stretched out and a sudden push is given to one end, a ripple travels through the coils. Each coil vibrates forward and backward along the direction of motion of the ripple.

Another type of sound wave, as illustrated in Figure 16.7, is caused by air molecules vibrating backward and forward, creating small areas of lower and higher pressures. The low and high areas tend to restore equilibrium. Low areas are referred to as *rarefaction*. High areas are referred to as *compressions*.

Although they have different physical causes, the transverse and longitudinal waves behave in essentially the same way. You often display longitudinal waves by a graph of pressure against time, which looks exactly like any other waveform. Longitudinal waves also reflect, refract, and diffuse exactly as transverse waves do.

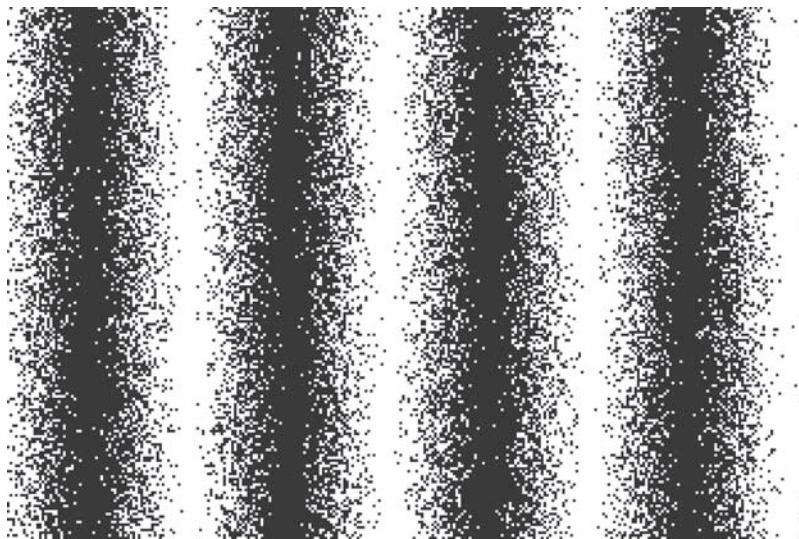


Figure 16.7
A longitudinal wave.

As shown in Figure 16.8, both types of wave are often also drawn in a kind of “plan view,” where the wave is represented by a series of lines representing particular wavefronts.

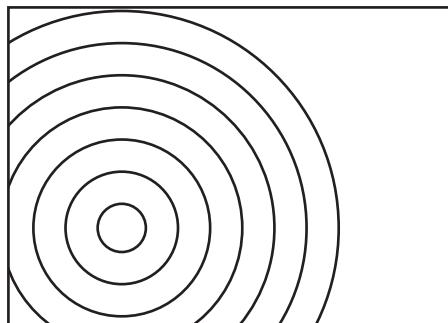


Figure 16.8
A wave represented as a succession of wavefronts.

Wave Addition and Subtraction

Because a wave is a virtual object, it's perfectly possible for several waves to be traveling through the same medium. In fact, this happens routinely. The air around you, for example, contains what could be considered to be an infinite number of different waves of light traveling in all directions. However, because waves are virtual, it's only humans who consider this to be happening. In reality, all that is going on is a constant fluctuation of electromagnetic fields in space. The fact that energy is being transferred in the process is almost accidental.

You can conceive of waves in these two contradictory ways because they have an important property. They can be combined together into a single, more complex wave. For example, you can combine the two waves in Figure 16.9 into a single, more complex waveform just by adding the displacement values at each point.

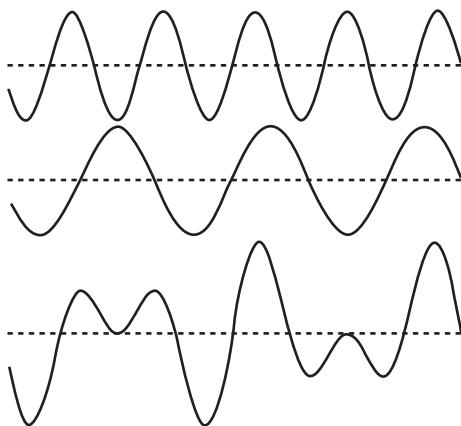


Figure 16.9

Adding waves.

Combining waves creates some interesting effects. Among other things, if you add two waves together that are exactly out of phase, they combine to give a straight line. No wave at all results. Three waves, all mutually out of phase by $2\pi/3$, produce the same result. Drawing on this occurrence, in order to decrease the total current flow, electricity is often transmitted in three simultaneous waves out of phase with one another.

If you can combine waves, you can also *decompose* a wave into others by subtracting them. A technique called *Fourier analysis* allows you to decompose any waveform, no matter how complex, into sine waves of varying amplitude and phase. This is essentially what you do when listening to sound. You try to separate out different waveforms that correspond to different sound sources.

The result of decomposing a wave into individual sine waves is the *spectrum* of the wave. The spectrum of the wave is a kind of “signature” for a particular wave emitter. The plural of *spectrum* is *spectra*, and spectra are used in many contexts for different types of analysis. In one application, they are used to distinguish different kinds of chemical elements. When burned, each element emits a characteristic spectrum of light radiation. This allows you to work out the chemical composition of distant stars, and to find out some other interesting facts.

In contrast to light, particular musical instruments also have a characteristic sound spectrum (*timbre*), consisting of different integer and fraction multiples of the primary note. A sound wave is perceived as a particular note according to the frequency. Your perception of a complex waveform as a single sound at a single frequency is a very impressive bit of mental computation, and is the equivalent to your perception of a complex spectrum of light wavelengths as a single color. This topic is addressed once again in Chapter 20.

Wave Behavior

Several important behaviors of waves are a consequence of their physics. Consider first *reflection*. When a wave strikes a barrier of some kind, depending on the natures of the wave and the barrier it encounters, it reflects off the barrier. If the barrier can't absorb the energy, the wave is sent back to where it came from. As Figure 16.10 illustrates, the wave reacts exactly like an elastic collision, bouncing off the wall at the same angle as it is struck. The angle at which the wave strikes the wall is the *angle of incidence*. The angle at which it bounces off the wall is the *angle of reflection*.

A further form of behavior typical of waves is called *refraction*. Refraction occurs when the velocity of the wave changes. If you imagine a line of cars traveling at a speed of 60 mph hitting a zone where it has to travel at 30 mph, you can see that in the slower zone, the cars will end up bunched closer together. The same thing happens to the wavefronts of a wave. As the wave hits a new medium where its velocity is lower, the wavelength has to decrease. This can also be seen directly from the wave equation. The frequency of the wave is being generated somewhere else, and since the frequency of the wave depends only on the driving oscillation, it is fixed for any particular wave.

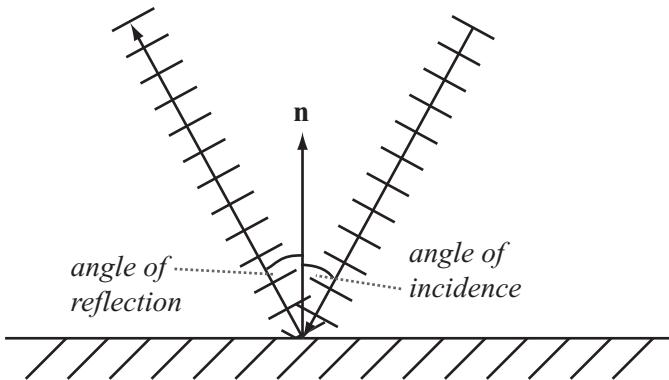


Figure 16.10

Reflection.

Changing the wavelength has other consequences, as is illustrated in Figure 16.11. When a succession of wavefronts hits a new medium at an angle, certain parts of the wave hit the start of the new medium (called the *interface*) earlier than others. This causes the wave to change its direction, like steering a tractor or tank by altering the relative speeds of the two tracks.

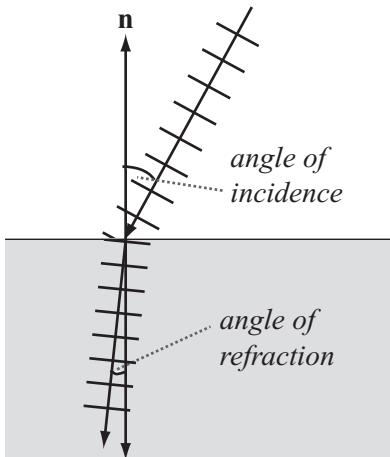


Figure 16.11

Refraction.

You see this effect whenever you look at a straw in a glass of water. At the point at which it enters the water, the straw appears to be broken. The appearance of the straw is further complicated by the fact that different wavelengths travel at different speeds in the water. The result is that the waves change directions by different amounts. A wave made up of several different wavelengths will split into a fan of rays, creating a prism effect as seen with rainbows. The amount by which a particular wave is deflected is described by Snell's law, which says that if α is the angle of incidence and β is the angle of refraction, they are related by

$$n_1 \sin \alpha = n_2 \sin \beta$$

where the n values are constant properties of the media through which the wave is traveling, called the *index of refraction*. For light, this value is equal to the quotient of the speed of light in a vacuum, c , with the speed of this wavelength in the medium.

Another phenomenon is called the *Doppler effect*. To understand how this works, consider the images of Heathcliff and Cathy. Suppose Heathcliff is traveling past Cathy on a train and calls out to her. The speed of the train causes the wavelength of the sound, as Cathy hears it, to shorten. This is so because, as Heathcliff travels forward, each wavefront is emitted closer to the previous one.

You can also calculate the *Doppler shift*. The Doppler shift is the amount by which the wavelength is shifted. If the train has a speed s and the sound has a frequency f , then the wavelength will be decreased by a value $\frac{s}{f}$. If this takes the wavelength to a value less than zero, however, you'll hit a problem. This occurs when s is equal to the speed of sound, approximately 330 m/s. At this point, the wave is overtaken by the wave emitter, and you get a sonic boom. The calculation works equally well as Heathcliff disappears into the sunset. In this case, you use a negative speed for s . If Cathy is somewhere over the hills from the train, then you'll need to use the dot product to find the component of the train's velocity that's traveling in her direction.

The Doppler effect is used as one of the principal pieces of evidence for the expansion of the universe. As mentioned earlier, the characteristic spectrum of elements is used to determine the composition of stars. When you examine these spectra for particular stars, you find that on average they are all *red-shifted*. In other words, their wavelengths have increased because the stars are moving away from you. What's more, the rate at which they are moving away depends on their distance from you, and this suggests a picture of the universe as constantly expanding from an explosion.

Note

There's nothing special that puts Earth or the solar system at the center of the universe. All the stars are moving away from all the other stars relative to each other. Space itself is expanding.

One final property worth mentioning is *diffraction*. Diffraction occurs when a wave hits a partial barrier, such as an obstacle or a wall with a hole in it. In this case, you get a situation such as the one depicted by Figure 16.12, where the obstacle starts to act like a new source of the waves.

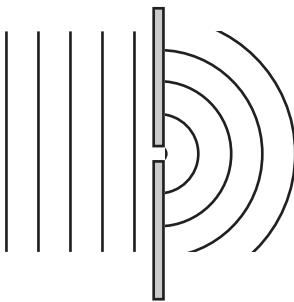


Figure 16.12

Diffraction.

When two holes or slits are near one another, the result can be quite interesting, since the two resulting waves, which have the same wavelength, frequency and velocity, combine at each point, creating an interference pattern. As Figure 16.13 shows, the two waves are at some points out of phase with each other, resulting in no energy transfer. At other points, they are exactly in phase, creating a double effect. If you place a bunch of detectors in a line, then you see interference fringes, whose discovery in the case of light was one of the principal pieces of evidence that light is a wave.

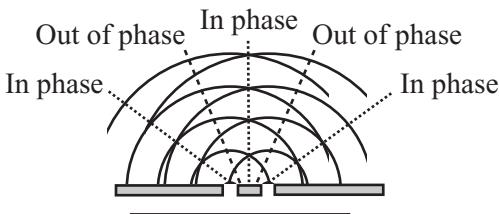


Figure 16.13

Interference patterns in double-slit diffraction.

Exercise

EXERCISE 16.1

Incorporate the functions in this chapter into a system that allows you to click, drag, and throw an object on a virtual spring. Try doing this for each of the systems given; you'll find subtle differences between them. The hardest is the `forceDueToSpring()` function.

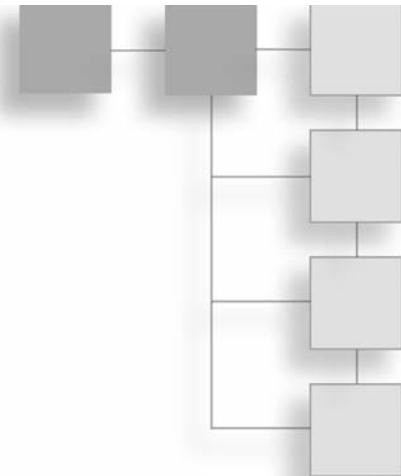
Summary

In this chapter, you've seen a number of ways that an object can move when attached to an elastic band or spring. Springs are very useful in creating physical simulations since they allow you to connect objects together, making virtual cloth, ropes, chains, and similar systems of connected objects. You've also seen how such connected objects can create virtual packets of energy in the form of waves.

This concludes your examination of the more obscure topics of physics, and apart from a few more discussions of collisions in three dimensions, at this point you are mostly finished with physics. You're now going to move back into mathematics and extend what you have already covered into the third dimension.

You Should Now Know

- How a *spring* works
- How to calculate the *tension* and energy in a stretched or compressed spring
- The meaning of the terms *coefficient of elasticity*, *damping*, and *elastic limit*
- How to calculate the position and velocity of a particle under *simple* or *damped harmonic motion*
- How and when the phenomena of *resonance* and *coupling* occur
- What a wave is and the meanings of *frequency*, *wavelength*, and *velocity* as applied to a wave
- How the physics of waves creates *reflection*, *refraction*, *diffusion*, and *Doppler shifting*



PART IV

3-D MATHEMATICS

You've already done most of the work to enable you to move from two to three dimensions. The concepts of vectors, forces, energy, and momentum are the same in three dimensions as in two, and collision detection, while more complex, involves all the same techniques as before. In this fourth part of the book, you'll finish your exploration of mathematical and physical theory by taking this final step into 3-D.

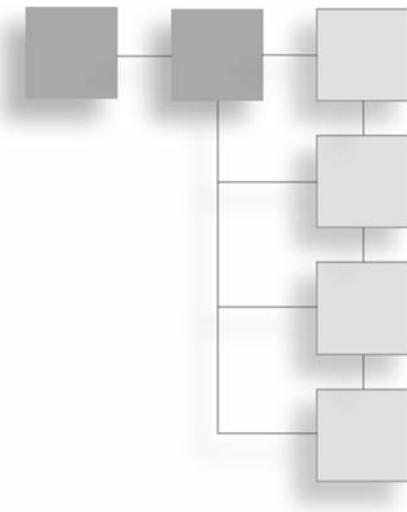
You'll start by looking at the basics of 3-D space, and how it can be represented on a two-dimensional screen. Then you'll spend another chapter further extending the vector work you have done up to now, and I'll introduce the concept of a transform. Chapter 18 deals with collision detection for 3-D shapes, Chapter 19 looks at lighting and shading, and the final chapter covers various 3-D modeling techniques for creating complex objects and moving surfaces, such as waves on water.

3-D is a subject much more widely covered than the more general techniques in this book, and so you won't retread the ground of many other authors. Most of the topics are dealt with briefly. For a full mathematical treatment, along with much more information on the more obscure topics, it is helpful to consult the references given in the appendices of the book.

This page intentionally left blank

CHAPTER 17

3-D GEOMETRY



In This Chapter

- Overview
- 3-D Vectors
- Rendering
- Casting a Ray

Overview

Thus far, most of the discussion in this book has involved 2-D problems. While adding a third dimension is mostly just a matter of adding another number, doing so has many implications. For one thing, since your computer screen has two dimensions, not three, when you enter the third dimension, you must find some way to flatten the extra dimension back to a 2-D image. Further, objects in 3-D have sides that are visible and that occlude other sides. To understand how to work with such problems, in this chapter you'll start by looking at how 3-D space can be represented.

3-D Vectors

By now you are used to representing a point in 2-D space in terms of a pair of Cartesian coordinates measured along two dimensions (x and y) from a fixed origin. Adding the third dimension (z , for example) involves little more than adding a third number to a list.

Adding the Third Dimension

As with 2-D geometry, with 3-D geometry you start by creating a space. Creating a 3-D space involves defining three orthogonal (mutually perpendicular) axes and an origin. How you orient the axes is arbitrary. You can point your axes in any direction you like. It is conventional, however, to consider the x -axis as “left to right,” the y -axis as “down to up” and the z -axis as “front to back.” If you move down two units, right one unit and forward three units, your actions correspond to the vector $(1 - 2 3)^T$.

Note

One of the most common orientations of a 3-D space involves a *left-hand axis*. With the left-handed axes, imagine grasping the z -axis with your left hand, with the thumb pointing in the positive z direction. Your fingers curl around from the positive x to the positive y direction.

The three-dimensional vector is represented in the same way as the two-dimensional vector. You just add one more component. Most other aspects of vector geometry stay the same. In fact, there is only one addition, which will be discussed shortly. As an example of how 3-D vectors work, consider applying the Pythagorean Theorem. You represent the magnitude of a 3-D vector $(x \ y \ z)^T$ as $\sqrt{x^2 + y^2 + z^2}$. The dot product is given by multiplying three instead of two components pairwise:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \cdot \begin{pmatrix} d \\ e \\ f \end{pmatrix} = ad + be + cf$$

Just as in 2-D space, in 3-D space you can define a *line* by using a point with position vector \mathbf{p} and a direction vector \mathbf{v} so that every point on the line has position $\mathbf{p} + t\mathbf{v}$ for some t . You can also define a *plane* in 3-D space in much the same way. As shown in figure 17.1, you might employ a point \mathbf{p} and two non-collinear vectors \mathbf{v} and \mathbf{w} , so that each point on the plane is $\mathbf{p} + t\mathbf{v} + s\mathbf{w}$ for some s and t . Another approach is to choose a point \mathbf{p} and a normal vector \mathbf{n} . The vector \mathbf{n} is perpendicular to all the vectors in the plane. While this second method is more efficient, it is less convenient for some calculations. For this reason, in some contexts in this and subsequent chapters, the first approach will be used.

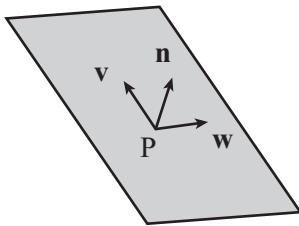


Figure 17.1
Defining a plane.

It's worth noting that the normal gives you a useful equation relating points on a plane. If the normal is $(a \ b \ c)^T$, then the points on the plane all conform to the equation $ax + by + cz = d$, where d is the perpendicular distance of the plane from the origin. This is the 3-D equivalent to the line equation $ay + bx = c$. In addition, note that if \mathbf{n} is a unit vector and \mathbf{p} is on the plane, then $d = \mathbf{p} \cdot \mathbf{n}$.

The Vector (Cross) Product

Dot products involving 3-D values work much the same way as 2-D dot products. However, when working in three dimensions, you have a new way to combine two vectors. This is known as the *vector product*, or more commonly the *cross product*. The vector or cross product is usually designated with a multiplication sign. For example, you represent the vector or cross product of the vectors \mathbf{x} and \mathbf{w} as $\mathbf{x} \times \mathbf{w}$. Unlike the dot product, which returns a scalar value, the cross product returns a vector. In other words given two vectors, the vector product returns a third vector that is perpendicular to the two vectors you have used in the product. The result is essentially the three-dimensional equivalent to the `normalVector()` function introduced in Chapter 5.

Calculating the vector product is a little more awkward than calculating the dot product. The formula for accomplishing this is as follows:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} bf - cd \\ cd - fa \\ ae - bd \end{pmatrix}$$

To remember the equation, one approach is to think of it as a determinant represented by a 3×3 matrix:

$$\begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a & b & c \\ d & e & f \end{pmatrix}$$

In this matrix, since \mathbf{i} , \mathbf{j} , \mathbf{k} are the basis vectors $(1 \ 0 \ 0)^T$, $(0 \ 1 \ 0)^T$, and $(0 \ 0 \ 1)^T$, each element of the determinant corresponds to one component of the cross product vector. Given this understanding, you can then develop the `crossProduct()` function fairly readily:

```
function crossProduct(v1, v2)
    set x to v1[2]*v2[3]-v2[2]*v1[3]
    set y to v1[3]*v2[1]-v1[1]*v2[3]
    set z to v1[1]*v2[2]-v1[2]*v2[1]
    return vector(x,y,z)
end function
```

Here is a list of some of the most prominent features or properties of the cross product:

- The cross product is not commutative. In fact, $\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}$. Nor is it associative. In general, $\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) \neq (\mathbf{v} \times \mathbf{w}) \times \mathbf{u}$.
- The cross product is distributive over addition: $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = \mathbf{u} \times \mathbf{v} + \mathbf{u} \times \mathbf{w}$.
- The cross product of a vector with itself is the zero vector.
- If you take the scalar product with either of the two original vectors, if the two vectors are perpendicular, you get zero.
- If \mathbf{v} and \mathbf{w} are orthogonal unit vectors, the cross product is also a unit vector.
- Generally, if the angle between \mathbf{v} and \mathbf{w} is θ , then $|\mathbf{v} \times \mathbf{w}| = |\mathbf{v}| |\mathbf{w}| \sin \theta$.
- The magnitude of the cross product of two vectors is the area of the parallelogram whose sides are defined by the vectors. In other words, the area of the shape ABCD, where $\overrightarrow{AB} = \overrightarrow{DC} = \mathbf{v}$ and $\overrightarrow{BC} = \overrightarrow{AD} = \mathbf{w}$, equals $|\mathbf{v} \times \mathbf{w}|$.
- The direction of the cross product always follows the same handedness as the axes. Stated differently, if you rotate the space so that the input vectors are aligned as closely as possible to the x - and y -axes, the output vector is aligned in the positive z -direction. As long as your basis maintains the same handedness, the cross-product is independent of the basis.

Using the Cross Product

The cross product, like the dot product, is a useful way to *regularize* a situation. Regularization involves removing unnecessary elements. For example, suppose you have a plane for which you know the normal \mathbf{n} but want to describe it instead by using two vectors on the plane. You can employ the cross product to do this. First, choose an arbitrary vector \mathbf{w} that is not collinear with \mathbf{n} . Next, take the cross product $\mathbf{w} \times \mathbf{n}$. This gives a new vector \mathbf{v} that is perpendicular to \mathbf{n} (and also to \mathbf{w}) and that therefore lies in the plane you are interested in. Finally, take the cross product again to get $\mathbf{u} = \mathbf{v} \times \mathbf{n}$. Taking the cross product gives a second vector perpendicular to \mathbf{n} , which is also perpendicular to \mathbf{v} .

Note

When you work with 3-D calculations, it becomes difficult to remember the distinction between *position* and *direction* vectors. A position vector goes from the origin to a given point. Generally, a direction vector goes from a point established by a position vector to another point established by a position vector.

Another common use for the cross product is to find an *axis of rotation*. You'll look at this further later, but for now consider Figure 17.2. An arrow at \mathbf{p} is pointing along the vector \mathbf{v} , and you want it instead to point along the vector \mathbf{w} . By finding the cross product $\mathbf{v} \times \mathbf{w}$, you have a vector perpendicular to both. This serves as an axis of rotation. You can rotate the arrow around this vector to turn it in the right direction. What's more, by finding the dot product $\mathbf{v} \cdot \mathbf{w}$, you can also calculate the angle of rotation.

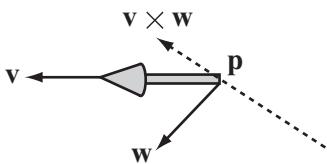
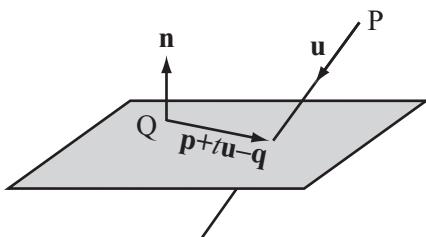


Figure 17.2

Using the vector and scalar products to calculate a rotation.

As a further exploration of the dot products, it's worth noting one more important calculation. This calculation involves the point of intersection of a line and a plane. Suppose you have a line defined by the point \mathbf{P} and a vector \mathbf{u} , and a plane defined by a point \mathbf{Q} and a normal \mathbf{n} . As illustrated by Figure 17.3, your objective is to know a value t , such that $\mathbf{p} + t\mathbf{u}$ lies on the plane.

**Figure 17.3**

Finding the point of intersection of a line and a plane.

There are several ways to discover the value of t . One in particular keeps things simple. Notice that for any point \mathbf{a} on the plane, $\mathbf{a} - \mathbf{q}$ is perpendicular to \mathbf{n} . In particular,

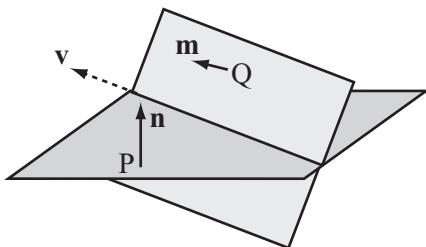
$$(\mathbf{p} + t\mathbf{u} - \mathbf{q}) \cdot \mathbf{n} = 0$$

Because the scalar product is associative, you can reason that $(\mathbf{p} - \mathbf{q}) \cdot \mathbf{n} = t\mathbf{u} \cdot \mathbf{n}$. As a result, can also reason that

$$t = \frac{(\mathbf{p} - \mathbf{q}) \cdot \mathbf{n}}{\mathbf{u} \cdot \mathbf{n}}$$

As you might expect, however, since there is no intersection between the line and plane, this approach fails when \mathbf{u} and \mathbf{n} are perpendicular. Otherwise, it is quick and reliable.

You can use a similar technique to find the line of intersection of two planes. Accordingly, if two planes are represented by the points \mathbf{p} and \mathbf{q} and the normals \mathbf{n} and \mathbf{m} , respectively, then as shown in Figure 17.4, your objective is to find the line that lies on both planes.

**Figure 17.4**

The line of intersection of two planes.

To find the line, first notice that since this line lies on both planes, it must be perpendicular to both **n** and **m**. Given that it is perpendicular, you can find its direction vector **v** as **n** \times **m**. Having gone this far, you now need only to find a single point on the line. To find this point, you apply the previous result. You choose an arbitrary vector in the first plane (a good choice is **n** \times **v**) and see where the line through **p** along this vector intersects the second plane. The `linePlaneIntersection()` and `planePlaneIntersection()` functions encapsulate these activities. Interactions of lines and planes are attended to by the first function:

```
function linePlaneIntersection(linePt, lineVect, planePt, planeNormal)
    set d to dotProduct(lineVect, planeNormal)
    if d=0 then return "no intersection"
    set v to linePt-planePt
    return dotProduct(v, planeNormal)/d
end function
```

The second function attends to intersections of planes:

```
function planePlaneIntersection (pt1, normal1, pt2, normal2)
    set v to crossProduct(normal1,normal2)
    set u to crossProduct(normal1, v)
    set p to linePlaneIntersection(pt1, u, pt2, normal2)
    if p="no intersection" then return p
    return array(p,v)
end function
```

Homogeneous Coordinates

Although you need only three coordinates to represent 3-D space, in practice you often use four. There are a number of reasons for this, and discussion of these reasons will be presented in Chapter 18. However, for now, consider that in addition to the *x*, *y*, and *z* coordinates, you add one more, usually designated as *w*. The result is a four-dimensional vector. This vector follows the rules that have been reviewed for two- and three-dimensional vectors.

While adding a fourth coordinate value might seem arbitrary and confusing, the w -coordinate comes in very handy in a number of instances. The w -coordinate is referred to as a *homogeneous coordinate*. The term *homogeneous* means something like “having a similar dimension.” For a position vector, w is set to 1. For a direction vector, w it is 0. To see how this works out, consider the following equation:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} - \begin{pmatrix} a \\ b \\ c \\ 1 \end{pmatrix} = \begin{pmatrix} x - a \\ y - b \\ z - c \\ 0 \end{pmatrix}$$

The vector from one position vector to another should have a zero w -component. However, at this point, it is not important to emphasize this point too strongly. As becomes clear upon further study, vector addition with homogeneous coordinates is not quite the same as vector addition using only normal coordinates.

Still, to understand the purpose of homogeneous coordinates in 3-D calculations, it's helpful to briefly draw from a discussion of two-dimensional activities. As shown by Figure 17.5, a two-dimensional plane is defined using axes x' and y' . The plane is placed in a three-dimensional space with axes x , y , and w , and it coincides with the $w = 1$ plane. As a result, any point (x', y') in the plane coincides with a point $(x, y, 1)$ in the 3-D space.

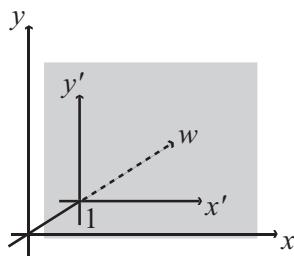


Figure 17.5
Homogeneous coordinates in 2-D.

However, it's not just points on the plane that you can map to the 2-D space. You can also compress the whole of the 3-D space onto the plane by a process of projection. As shown by Figure 17.6, for any point P , you draw a line from the 3-D origin through P and find its point of intersection with the plane.

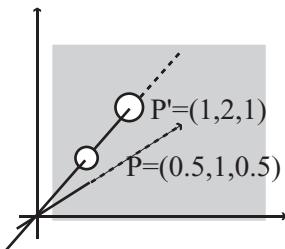


Figure 17.6

Projecting 3-D homogeneous space to the plane.

You can even do this with homogeneous points that have $w = 0$. Although the line OP is parallel to the plane, you still say that it intersects the plane “at infinity.” In other words, it intersects the plane infinitely far along the line on the plane parallel to OP . In all other cases, the point on the plane that corresponds to (x, y, w) is $\left(\frac{x}{w}, \frac{y}{w}\right)$. In the strictest sense,

in fact, there is no difference between the homogeneous points (x, y, w) and $\left(\frac{x}{w}, \frac{y}{w}, 1\right)$. Mathematically, they are considered to be equal. The outcome is that homogeneous coordinates are scale-invariant. If you multiply them by a constant factor, they remain the same.

As an introduction to using homogeneous coordinates, look again at the problem of finding the point of intersection of two lines in 2-D. The line through (a, b) with a vector $\begin{pmatrix} p \\ q \end{pmatrix}$ can be represented by the equations

$$a + tp = x$$

$$b + tq = y$$

which give

$$\begin{aligned} \frac{x-a}{p} &= \frac{y-b}{q} \\ qx - py - aq + bp &= 0 \end{aligned}$$

In general, a line in 2-D can be represented as $ax + by + c = 0$. Using homogeneous coordinates, this is expressed as $ax + by + cw = 0$. Comparing these two forms of expression, you can see that a line in 2-D corresponds to a plane in homogeneous coordinate space. If you represent two lines in this way, you can find their point of intersection using this approach:

$$ax + by + cw = 0$$

$$px + qy + rw = 0$$

$$\frac{x}{w} = \frac{br - cq}{aq - bp}$$

$$\frac{y}{w} = \frac{cp - ar}{aq - bp}$$

From these operations, you end up with the homogeneous coordinates $(br - cq, cp - ar, aq - bp)$. Notice that if $aq - bp = 0$, then the two lines are parallel. One resulting observation is that while in standard coordinates there is no solution to the equation, in homogeneous coordinates the two lines meet at a point at infinity with $w = 0$. Further, the position vector of this point is given by the cross product

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

The cross product makes sense when you consider that since the two vectors give the normals of the planes in 3-D space, their cross product gives the intersection of the planes. The intersection of the planes corresponds to the intersection of the 2-D lines.

Homogeneous coordinates will be addressed again in the next chapter, where attention is given to how they can be applied in 3-D. Still, it is beneficial to keep them in mind as you read the next section, where I'll discuss the projection plane for 3-D rendering. The two are closely related.

Rendering

The process of translating a 3-D scene into a 2-D picture is called *rendering*. While this is a complex process that you'll return to at a later stage, at this point, it is worthwhile to consider a few preliminaries, some of which involve the physical media through which rendering is accomplished.

The Projection Plane

With respect to projection planes, it is helpful first to consider how your eyes receive light reflected from the objects around them. Since light rays travel in a straight line, if you draw a line from your eye out into the world, what you see is the first thing in line with your eye, and your field of vision encompasses anything taken in by the lines coming from you within a particular angle from your eye.

To consider this in light of what happens in the construction of a 3-D world, imagine an observer standing still and looking straight ahead. The world visible to the observer, from the area straight ahead to the peripheral regions on the sides and top and bottom, constitutes the view *frustum*, as shown in Figure 17.7. The frustum is cut or intersected by different *viewing* or *projection planes*. Figure 17.7 shows two planes, the near plane and the far plane.

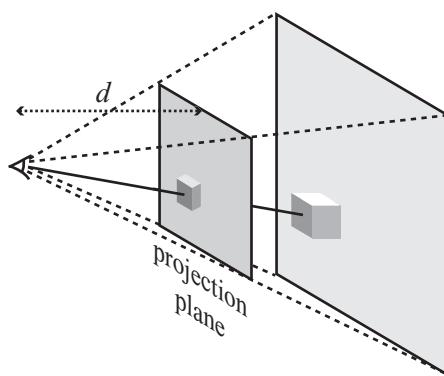


Figure 17.7

The field of view and the view frustum.

Note

One of the difficulties in describing 3-D geometry is drawing clear diagrams on a 2-D page. Figure 17.7 shows a 3-D cube transformed into a 2-D picture on the projection plane.

The planes in the frustum are the primary devices used to translate a 3-D into a 2-D image. Referring to Figure 17.7, imagine that an observer is viewing the 3-D world as shown by the projection plane at some distance d . To work out what is to be found at each point of the plane, you can draw a line from the observer and see what it hits. This is called *raytracing*. For real-time animation, you can calculate where in the space each object is, draw a line to the observer, and see where this line intersects the plane. The precise situation is complicated by lights and texturing, but this is the essential principle.

As illustrated by Figure 17.8, to find the point on the projection plane that corresponds to a particular point in space, assume that the observer is facing in the direction \mathbf{n} . The point you are interested in is at point \mathbf{p} . Likewise, assume that the projection plane is at a perpendicular distance d from the observer, who is at the point \mathbf{o} .

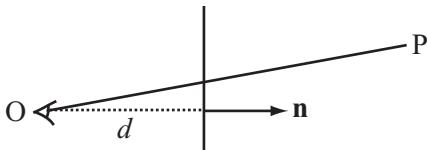


Figure 17.8

Projecting a point to the projection plane.

Figure 17.8 represents a cross-section, and in it a line is drawn from the point at \mathbf{P} to the observer. This line passes through the projection plane at some unknown point. The situation depicted, then, is nearly the same as what is represented by Figure 17.7.

In mathematical terms, the plane normal of Figure 17.8 is the same as the direction the observer is facing. You can find a reference point on the plane drawn at the center of the screen as $\mathbf{o} + d\mathbf{n}$, assuming \mathbf{n} is normalized. So the point on the projection plane corresponding to \mathbf{P} is given by the intersection of the line starting at \mathbf{P} in the direction $\mathbf{o} - \mathbf{p}$ with this plane, which is the point $\mathbf{p} + t(\mathbf{o} - \mathbf{p})$, where

$$t = \frac{(\mathbf{p} - (\mathbf{o} + d\mathbf{n})) \cdot \mathbf{n}}{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}} = -d \frac{\mathbf{n} \cdot \mathbf{n}}{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}} - 1 = \frac{d}{(\mathbf{p} - \mathbf{o}) \cdot \mathbf{n}} - 1$$

Note that since \mathbf{n} is normalized, $\mathbf{n} \cdot \mathbf{n} = 1$.

Having calculated what the projection plane looks like, you can now draw its contents at the correct size on the computer screen, called a *viewport*. There are a number of different ways you can do this. One way involves specifying the scale. For example, you might determine that one unit of the projection plane corresponds to one pixel on the screen. As shown in Figure 17.9, more commonly you specify the viewport by giving one of the maximum angles in the field of view of the observer, such as the angle θ . In some instances, the whole field of view angle, given by 2θ , is specified.

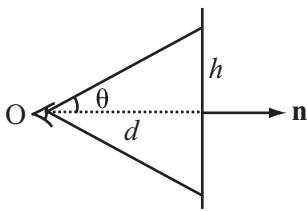


Figure 17.9
Specifying the field of view.

Since you know that $\tan\theta = \frac{h}{d}$, you also know the height of the topmost point on the projection plane. This then becomes a scale for drawing to the screen.

This approach to calculation, however, can become highly overcomplicated, since you don't need the value of d . All you actually need to know is the size of the picture on the screen and the angle of the field of view. To make the figures come out correctly, you can then set your projection screen at the appropriate place. Then you simply need to use similar triangles.

You can calculate the distance of the point from the observer along the camera vector to be $(\mathbf{p} - \mathbf{o}) \cdot \mathbf{n}$ and its vertical displacement as $(\mathbf{p} - \mathbf{o}) \cdot \mathbf{u}$, where \mathbf{u} is the “up” vector of the camera. If the projection plane is to have a height of h , then you must also have that

$d = \frac{h}{\tan\theta}$, so the height of the point as projected onto the screen is given by $\frac{h_{\max} \cdot (\mathbf{p} - \mathbf{o}) \cdot \mathbf{u}}{\tan\theta(\mathbf{p} - \mathbf{o}) \cdot \mathbf{n}}$.

Here's a function that draws together some of the previous observations, translating a point in space to a point on the screen. As arguments, the `pos3DToScreenPos()` function takes the observer position and normal, the vertical field of view angle, the height of the screen, and the up-vector of the observer.

```
function pos3DToScreenPos(pt, observerPos, observerVect,
                         observerUp, fov, h)
    set observerRight to crossProduct(observerUp, observerVect)
    set v to pt-observerPos
    set z to dotProduct(v,observerVect)
    set d to h * tan(fov)
    set x to d*dotProduct(v,observerRight)/z
    set y to d*dotProduct(v,observerUp)/z
    return vector(x,-y)
end function
```

The `pos3DToScreenPos()` function returns the point's position on the screen relative to the center of the 3-D viewport. The position of y is measured downward. Since in practice you precalculate many such values, the function could be made more efficient. In addition, there are also some complications about points behind the observer. On the other hand, in practice you can let your 3-D card handle this part, as will become apparent momentarily.

To explore further how an image is displayed, as mentioned previously, the set of points that is visible to a particular observer is called the view frustum. As illustrated by Figure 17.7 (shown previously), the view frustum is defined in part by the base of a truncated pyramid. A pyramid is a six-sided shape. Its sides are at an angle determined by the field of view. Its front and back faces are determined arbitrarily and are sometimes referred to as the *hither* and *yon* of the camera. While you can theoretically see infinitely far and infinitely near, when rendering it is convenient to *clip* the scene at a certain distance, often using fogging to obscure the distant objects before they disappear from view.

In the equation for screen projection, notice that although changing \mathbf{o} and θ both affect the image on screen, they do so in different ways. Moving the observer while leaving θ constant moves the view frustum. It changes the scale of the image linearly. By changing the field of view, however, the shape of the view frustum can be changed. This is like altering the angle of a camera lens. Choosing the correct field of view for your scene can take a little fiddling with the parameters to make something that looks right, just as a director or cinematographer may spend a long time choosing the right lens. None is more correct, but they have different visual effects. The most natural-feeling will be a viewing angle that is the same as the angle the computer user has to the computer screen—but this depends on how near they are to the screen and the resolution they are using.

Perspective

Perspective is a technique for representing objects on a flat surface. The technique in art of using perspective was mathematically defined during the Renaissance, but the basic principles of perspective had been used prior to that. As defined mathematically, perspective is the approach used to render a 3-D scene on paper. While useful for drawing, however, it must be technically translated if it is to be an effective approach to rendering images on a monitor.

To draw a scene in perspective, you start by creating a horizon. The horizon is supposed to be drawn at the height of the observer's eyes. Each point on the horizon is a vanishing point. The vanishing point represents an object infinitely far away along a line drawn from the observer's vertical axis. As shown in Figure 17.10, the classic example of this is a road stretching into the distance.

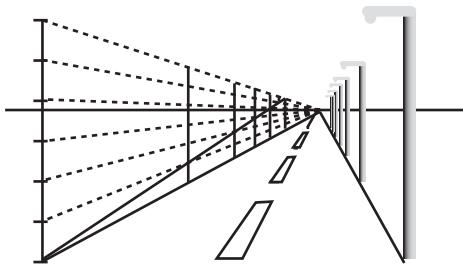


Figure 17.10

A perspective drawing.

In Figure 17.10, in addition to the road, a number of lampposts are drawn. If you were to travel along the road, the lampposts would be spaced along the road, as would be the lines on the road. The dotted lines on the left of the figure illustrate the basic technique. Two types of lines are used. Those that begin with the first post and converge at a point in the horizon are called orthogonal lines. Those that represent the lampposts or are transverse lines.

To develop the perspective, you divide the transverse line of the front post into several equal parts, joining each one to the vanishing point using an orthogonal line. You then draw in a line from the bottom of the first post to the top of the post that represents the visible background. Your remaining lampposts are placed so that they coincide with the points where this line meets the orthogonal lines.

Mathematically, you can calculate that the image size of an object decays exponentially with distance:

$$\text{Height in image} = \text{height} \times e^{-k(d - d_0)}$$

where k is some constant and d_0 is the distance of an object whose height in the image is equal to its height in reality. There's nothing special about using e in the equation. Any base will do if you scale k accordingly.

The reason that perspective is not particularly useful in computer 3-D engines is that it requires many recalculations if your observer moves. However, if you're working with simple scenes and a fixed observer, it can be an easy way to create a quick 3-D effect. Exploring the difference between drawing and rendering involves seeing how the values k and d_0 in perspective drawing relate to d and θ in standard 3-D rendering.

Orthographic Projections

Although the standard perspective projection method described in the previous section is the most common way to draw a 3-D scene, other methods prove more useful in some circumstances, particularly in architectural and technical drawings.

Standard perspective projection is sometimes called *central projection*. With central projection, something distinctive occurs as you move farther away from the projection plane. As shown in Figure 17.11, as d increases and θ decreases, the objects on the screen always appear the same size, and the view frustum becomes a simple box.

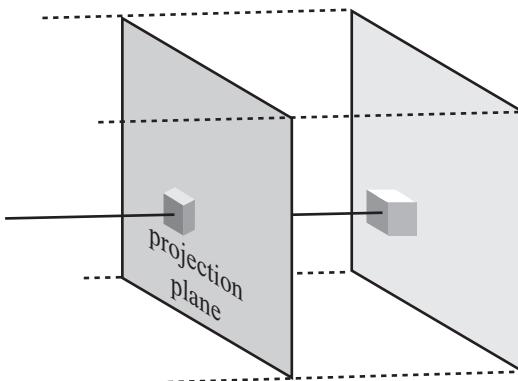


Figure 17.11
Orthographic projection.

The image shown in Figure 17.11 is called *orthographic*. When the object you are looking at is at a different angle, it can also be called *oblique*, *axonometric*, or *isometric*, but all these are equivalent to the orthographic view. Only the orientation of the object has changed. In an orthographic view, the distance of an object from the viewer or any other object has no effect on its size. It does, however, affect the position of the object on the screen and the drawing order. Otherwise, all you need to know is the size of the viewport.

In gaming, the most common of these orthographic views is the *isometric* view. This is where you look down on an object in an orthographic projection at such an angle that each side of the object is at the same angle to the projection plane, so they are all viewed as the same length. (You'll look at isometric games briefly in Part V.)

One more projection technique is worth noting. Instead of a projection plane, you use a *projection sphere* or *cylinder* around the observer. In some ways, you might imagine this to be more realistic. After all, you aren't actually looking at the world through a window but through two movable eyes. However, using a projection sphere creates some rather strange effects. One example is the Mercator projection of the earth, which is the most commonly seen map of the planet. In a Mercator projection, you imagine placing the earth inside a cylinder of paper and shining a light from the center. The shadow cast on the cylinder forms a map, which you can then unroll.

This is a good way to deal with the problem of mapping a spherical object like the earth onto a planar map, but it has some problems. Among other things, measuring distance is extremely difficult. The farther away you move from the equator, the more spread out the map is, until the North Pole is actually infinitely spread out, and infinitely high up the cylinder. On a map, this is why Greenland is so enormously out of proportion to its actual size. Navigating through a world using this kind of projection would be very weird and confusing.

Casting a Ray

As became evident in the discussion of 2-D images, it can often be useful to get a list of all objects along a *ray*. A ray is a line with one point of termination that extends infinitely. Chapter 10 discussed how to calculate the intersection of a ray with a plane. In the next couple of chapters, you'll calculate ray collisions for both simple shapes and polygonal meshes. In the current passage, it is worthwhile to look at a few applications of this method, assuming that you can employ a 3-D engine to calculate the ray for you.

As anticipated by the discussion in Chapter 10, *raycasting* is useful in many circumstances, but two are the most common. The first is in user interaction. For example, consider a situation that arises when someone clicks on the 3-D scene and must calculate what object is underneath it. The second is in collision detection. Consider a situation in which one or more rays cast in front of a shape can be used like the beam of a headlamp to determine whether there are any obstacles ahead.

Using a 3-D Engine to Find Objects Along a Path

The basic principle of raycasting is that you send a query to your real-time engine, passing it a starting point and a direction vector. The query then returns a list of models with which the ray intersects. Depending on your particular 3-D engine, the query might contain some optimization methods. In some cases, for example, you can specify a maximum number of models to return, a list of models to check against, and a maximum length for the ray. More detailed results might involve collision points and collision normals or the texture coordinates clicked on.

A good example of how this can be useful is in *terrain following*. Terrain following involves an object that is moving along a ground of varying height. Suppose your ground is a continuous triangular mesh, and you are creating a 4×4 vehicle that is driving along the terrain. How can you make the vehicle move realistically across the various bumps? To do so, you must know how high each wheel needs to be, orienting the vehicle accordingly.

One way to assess the heights of bumps is to store the information about the ground as a 2-D height map. This approach provides rapid information, but at the same time, it also requires high amounts of storage space. Another method—which might be quicker, depending on the speed of your 3-D engine and card and the complexity of the geometry of the terrain—is to cast a ray downward from each wheel and orient according to the collision points. This approach has the additional advantage of working equally well with a terrain that is changing over time, such as water waves.

A similar simple example is gunfire. In a first-person shooter (FPS), each player or enemy has a weapon that generally fires in a straight line. At the moment the gun is fired, a quick raycasting collision check determines where it will hit.

The technique is less useful when dealing with objects three or four times bigger than the mesh against which they are colliding. It is also not very useful with objects that are colliding with other objects that may be smaller. In such cases, you must use a large number of rays to determine whether the path is definitely clear. Certain workarounds are available, however. For example, if the objects are significantly different in size, it becomes possible to approximate the collision by using a proxy shape, such as a sphere or bounding box.

Picking, Dragging, and Dropping

Using interaction requires raycasting. The simplest example is when the player of your game clicks on the screen to select an object in 3-D space. How do you determine which object they have clicked on?

By choosing a point on the screen, your user has actually selected an angle for a ray. In fact, you're running the process of drawing a 3-D point to the screen in reverse. The simplest way to determine this point is to use the projection screen method to find a particular 3-D point under the mouse position. You then use this point to create your direction vector. To do this, as shown in previous calculations in this chapter, you need the field of view angle θ and the height h of the viewport. Then the distance to the projection plane can be found as $d = \frac{h}{\tan\theta}$.

Having found this distance, and knowing the camera's forward and up-vectors, you can quickly calculate the point under the mouse. The `screenPosTo3DPos()` function performs this calculation:

```
function screenPosTo3DPos(viewportPos, observerPos,
                           observerVect, observerUp, fov, h)
    set observerRight to crossProduct(observerUp, observerVect)
    set d to h / tan(fov)
    return observerVect*d - observerUp*viewportPos[2] +
                           observerRight*viewportPos[1]
end function
```

You now have all you need to cast a ray. The start point is the camera position, and the direction is the vector to the point you found. This tells you the model clicked on.

Suppose you need to drag the model to a new position. This question gives you a technical problem. While there are three dimensions to move it in, there are only two dimensions for the mouse. Somehow, the user's movements need to be translated into 3-D space.

How the movements are translated depends on the particular circumstances. Consider what happens if the model is part of a plane. It might be an object on the ground, a sliding tile, or a picture on the wall. If this is the case, then your two directions of motion are sufficient. You need only to find a way to constrain the motion in the particular direction.

If the model is free to move in space but not rotate, then the best solution is likely to combine the mouse movement with a key. You might program your game so that the user presses the Shift key while moving the mouse. This might cause the object to move in the camera's xz -plane rather than the xy -plane. Alternatively, while making the drag occur only in the current xy -plane, you can allow the user to move freely around the object, changing the plane of view or providing multiple views of the same object. Finally, if the model is supposed to be rotating in place, then mouse movements can correspond to rotations in the direction of two axes.

To accomplish such tasks, you can start with constraining to the plane. This is an extension of what you've already seen. As the mouse moves, you cast a ray from its current position to the plane you're interested in. You can use this as the new position for the dragged object, offsetting it by its height or radius as appropriate. As a note, the position does not have to be on a plane. It can be rough terrain or even other models. Think about the pool game example, which involved a "ghost" ball demonstrating where a collision would occur.

For a more sophisticated variant, you can "fake" the ray position so that the users feel they're dragging the middle of the object instead of its base. Figure 17.12 illustrates how this might work. First, you calculate the current offset of the base of the object from its midpoint as they appear on the screen coordinates. After that, you apply this offset to the current mouse position to get the ground position you're interested in. Now you drag the object to sit correctly at that position. The user isn't precisely dragging the midpoint of the object, but it feels that way.

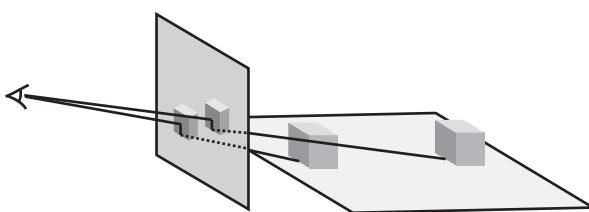


Figure 17.12

Dragging an object along the plane.

Dragging the object freely in space is an extension of this. You'll always have to constrain to a plane. However, you can give a freer choice as to what plane that is. Generally, while the most natural approach involves using a plane normal to the camera, you can also use one normal to the up-vector, or any other vector that seems convenient.

A much more difficult problem comes when you want to allow the user to rotate objects with the mouse. You can start with the simplest of these situations, which is when you want players to be able to rotate themselves, as in a mouse-driven FPS. In that case, a mouse movement corresponds to a camera rotation.

This problem is reasonably easy to solve. This is because there is a straightforward mapping from the mouse movement to the desired movement. At each time-step, you rotate the camera to point along whatever ray is currently underneath the mouse. (You'll see in the next chapter how to rotate an object to point in a particular direction.) To accomplish this, you must decide whether to keep rotating if the mouse is held still but off-center. You must also decide whether to rotate only when the mouse is moved. In the case of mouse movement, you need to reset the cursor position to the center of the screen each time you move. (With this approach, it is best to hide the cursor.)

When using the mouse to rotate an object in the camera view, things get a little trickier. There is no hard-and-fast translation from one to the other. You can make a start, however. The most natural assumption for someone rotating an object is that the point of the object clicked on will remain under the mouse as it rotates. You can do this as long as you know the intersection point of the ray. The difficulty is knowing which way should be up. As will be discussed in Chapter 18, you can specify an up-vector for a point-at operation. In this case, there's no obvious winner.

As an example of how to apply such observations, consider a case in which the object to be rotated has position vector \mathbf{p} , and the point on its clicked surface has position vector \mathbf{q} . Now the mouse is moved so that it defines a ray $\mathbf{o} + t\mathbf{v}$ from the observer. You want to know which direction to point the object so that it points toward the mouse position. In other words, you want to point the object toward the ray.

To find the correct direction, you need to find the point of intersection of the ray with the sphere centered on \mathbf{p} whose radius is the magnitude of $\mathbf{q} - \mathbf{p}$. If there is such an intersection, then this tells you the new position toward which the object should point. If there is no intersection, then the mouse is outside the range of the object, in which case you can either continue to rotate or point it as closely as possible toward the mouse. Another way to put it is to say that you point it along a vector perpendicular to the vector \mathbf{v} through the plane containing \mathbf{v} and $\mathbf{p} - \mathbf{o}$.

But what about the up-vector? One option is to use the camera's up-vector. This works reasonably well, but it leads to problems when you want to rotate the object to align with this vector. Another approach is to use the axis of rotation, but this causes the object to rotate rather erratically. A good compromise is to use the cross-product of the camera's direction vector and the movement vector of the point.

Exercise

EXERCISE 17.1

Create a function that will draw a three-dimensional cube, culling the back faces. While you can determine which faces of a cube are visible by using the normal of the plane, this is a task that is better left for later. For the moment, concentrate on the projection. See if you can work out how to use the cross product to determine if a particular face of the cube is visible. Use the function to make a cube that spins around its center. Look at the effect of using different projection techniques and fields of view.

Summary

In this chapter, you've had a brief introduction to three-dimensional space and how it works. As you've seen, the third dimension can be understood using the concepts used to discuss 2-D objects. What happens, however, is that calculations become more complex due to the extra freedom of movement and because you have to work with an observer who can't see the whole space.

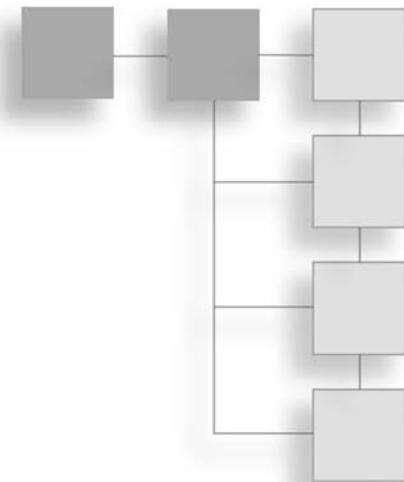
In the next chapter, you'll look further into how the space works and extend the matrix math you met in Chapter 5 to deal with the much more complex motions and transformations available to you.

You Should Now Know

- How to extend vectors into three dimensions
- The meaning and use of the *cross (vector)* product
- How to find points and lines of intersection between *lines* and *planes*
- The meaning of the term *homogeneous coordinates* and how to use them to calculate intersections
- How to use different kinds of *projection* to create your own 3-D engine
- How to use *raycasting* to create user interaction in 3-D space

CHAPTER 18

TRANSFORMS



In This Chapter

- Overview
- Describing Locations in Space
- Applying Transforms

Overview

You've already seen how a 2-D matrix can be used to describe a sequence of transformations in 2-D space. In this chapter, you'll look at the 3-D equivalent of this, the *transform*. You'll also see how transforms can be used to create objects with relationships to each other.

Describing Locations in Space

The 3-D world is populated by objects called *models*. Models are sets of vertices joined together in triangles to make a *mesh*. In order to work out where these models are, the 3-D engine treats them as a single object called a *node*. The node is associated with a particular location and orientation. The 3-D engine works out the positions of the vertices relative to the node. The same method is also used to describe the location of lights, cameras, and other members of 3-D space.

Position, Rotation, and Scale

To describe a node, you must know three things: its position, its rotation, and its scale. Rotation and scale can be handled by a transformation matrix. Position is a simple vector. The simplest way to visualize this combination of transformations is as a single process acting on the “raw” model. The first action is scaling. The second is rotating about the origin. The final is translating to the correct point.

All of these transformations are *affine*, meaning that if two lines are parallel before transformation, then they remain parallel afterward. Shears and reflections are also affine transformations. In fact, any transformation that can be performed by a matrix is affine. Rotations, translations, and reflections are slightly stricter. They are rigid-body transformations because, as well as affinity, they preserve angles between lines in space. In other words, while the relative positions of vertices may change, the angles between them remain the same. As illustrated by Figure 18.1, this is not true of scales in general.

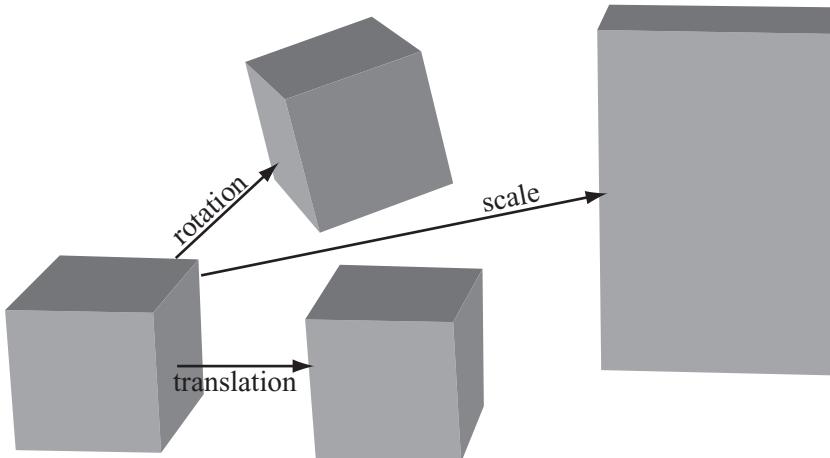


Figure 18.1

The three basic transformations.

Scale is the transformation that is easiest to visualize. Scaling an object by a constant factor involves multiplying each position vector by the scale factor. In terms of matrix mathematics, the matrix is multiplied the matrix $n\mathbf{I}$, where \mathbf{I} is the identity matrix. More generally, you can scale by different amounts in each direction. To accomplish this, you multiply the vectors by the following matrix:

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$$

Any scale transformation can be calculated in this way. For example, a scale factor of 2 in the direction $(1 \ 1 \ 0)$ is equivalent to the scale

$$\begin{pmatrix} \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation is more complicated than scaling. In two dimensions, rotation about the origin is relatively straightforward. It is described by a single angle. When a third dimension is added, however, you must specify an axis of rotation. Any vector in space can act as an axis, but as with the pool balls you looked at in previous chapters, due to topspin and sidespin, any rotation can be decomposed into simpler rotations about the basis vectors.

A rotation about one of the basis vectors can be described by a matrix like this:

$$\mathbf{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}$$

Combining all three such matrices gives you a generic rotation:

$$\mathbf{R} = \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x = \begin{pmatrix} \cos\xi & \sin\xi & 0 \\ -\sin\xi & \cos\xi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}$$

Note

The triple bar symbol (\equiv) here means “is defined as” or “is identical to.” It’s like a stronger kind of equals sign.

Multiplying out all these matrices is an extremely involved process, but it’s not as bad as it might seem. Suppose you transform the vectors \mathbf{i} , \mathbf{j} , and \mathbf{k} under \mathbf{R} . You end up with the column vectors \mathbf{u} , \mathbf{v} , and \mathbf{w} of the matrix. One feature of a rotation is that the vector between a point and its image must always be perpendicular to the axis of rotation. As a result, if you now take the cross product $(\mathbf{u} - \mathbf{i}) \times (\mathbf{v} - \mathbf{j})$, you find a new vector \mathbf{a} perpendicular to both. This vector \mathbf{a} is the overall axis of rotation.

What's more, you can calculate the angle of rotation α by finding the components of \mathbf{i} and \mathbf{v} perpendicular to \mathbf{a} . In other words, you find the result of projecting vectors \mathbf{i} and \mathbf{v} to a plane normal to \mathbf{a} . Here is how this can be accomplished:

$$\begin{aligned}\mathbf{i}_N &= \mathbf{i} - (\mathbf{a} \cdot \mathbf{i}) \mathbf{a} \\ \mathbf{v}_N &= \mathbf{v} - (\mathbf{a} \cdot \mathbf{v}) \mathbf{a} \\ \mathbf{i}_N \cdot \mathbf{v}_N &= (\mathbf{i} - (\mathbf{a} \cdot \mathbf{i}) \mathbf{a}) \cdot (\mathbf{v} - (\mathbf{a} \cdot \mathbf{v}) \mathbf{a}) \\ &= \mathbf{i} \cdot \mathbf{v} - (\mathbf{a} \cdot \mathbf{i}) (\mathbf{a} \cdot \mathbf{v}) (2 - \mathbf{a} \cdot \mathbf{a})\end{aligned}$$

Note

This only works if \mathbf{i} (and therefore \mathbf{v}) is not parallel to \mathbf{a} . If it is parallel to \mathbf{a} , you must use a different basis vector for comparison.

Assuming \mathbf{a} is a unit vector (if it is not, you can normalize it), and given the definition of rotation, noting that the dot product with \mathbf{i} and \mathbf{v} must be equal, you arrive at this equation:

$$\cos \alpha = \mathbf{i} \cdot \mathbf{v} - (\mathbf{a} \cdot \mathbf{i})^2$$

This process can always be used to combine two or more transformations into one. Conversely, since you can decompose a rotation \mathbf{R} into primitive rotations about the basis vectors, a rotation is often described by means of a rotation vector giving the angle of rotation about the three axes. You'll be coming back to this issue several times over the course of this section.

While the notions proposed in this section work fine, you still eventually hit a snag. How can you represent translations? It would be good to be able to perform a complete transformation in one step, including translations. Otherwise, among other things, any combination of transformations would require first translating back to the origin, performing rotations and scales, re-adding the position, and then translating again. Since position information is additive, however, and not multiplicative, such activity becomes seemingly impossible.

The Transformation Matrix

The solution to this problem is to use the homogeneous coordinates introduced in the previous chapter. With the aid of the fourth dimension, you can perform translations using a 4×4 matrix. Here is how this is done:

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ z+c \\ 1 \end{pmatrix}$$

Notice that the last row of the calculation is almost irrelevant. It serves largely as a piece of bookkeeping. Also, notice that if you perform a translation on a direction vector instead of a position vector, so that the w -coordinate is zero, the vector is unaffected by the transformation.

If you combine the resultant matrix with the scaling and rotation matrices mentioned previously, you end up with a generic transform

$$\mathbf{T} = \mathbf{P} \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x \mathbf{S}$$

where \mathbf{P} is a translation, and \mathbf{S} is a scale that appears like this

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$\mathbf{R}_z = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A similar relationship applies for \mathbf{R}_y and \mathbf{R}_x . The non-translation matrices are created simply by replacing the 3×3 matrix at the top-left of the 4×4 matrix with the appropriate 3-D matrix.

Because its bottom row is always $(0 \ 0 \ 0 \ 1)$, \mathbf{T} has 12 unknown entries. However, not all such matrices are valid transforms. Those with a determinant of zero are invalid. Of the rest, exactly half would perform a reflection and most others would create some form of shear. It's easy to see that there are nine degrees of freedom in creating a valid transform. Three are rotation angles, three are scale factors, and three are translation values.

Each of the matrices that make up a transform can be inverted fairly easily. For a translation, replace each value with its negation. For a rotation, negate the rotation angle. For a scale, replace the scale factors by their reciprocals. In fact, as it happens, inverting translations and rotations is even easier than that. Their inverse is actually their transpose. In such situations, you say the matrices are *orthogonal*, which is equivalent to the rigid-body nature of the transformation. This means that you can invert the whole transform. When you do this, it is important to remember that you must apply the inverses in the opposite order:

$$\mathbf{T}^{-1} = \mathbf{S}^{-1} \mathbf{R}_x^T \mathbf{R}_y^T \mathbf{R}_z^T \mathbf{P}^T$$

This process is straightforward providing that you keep track of the primitive transformations making up the transform. Otherwise, you have to either decompose the general transform into its component parts or invert it as it stands.

To decompose a transform into position, rotation, and scale information, you first strip off the translation element from the last column of the matrix. To calculate the scale and rotation, notice first that when you multiply \mathbf{S} by the basis vectors, you multiply the basis vectors by the scale factors a , b , and c . For example, $\mathbf{RSi} = \mathbf{R}(ai) = a(\mathbf{Ri})$. Also, since \mathbf{R} is a rigid rotation, \mathbf{Ri} is a unit vector.

A further point is that \mathbf{Ri} is also the first column of \mathbf{R} . Given this information, a must be the magnitude of the first column of \mathbf{RS} , and the first column of \mathbf{R} is the normalized version of this vector. Since the case is similar for b and c , you can split the transform into the three underlying transformations.

Further decomposing \mathbf{R} into rotations about the three axes is also possible, although not always necessary. You're looking for θ , ϕ , and ξ such that

$$\begin{pmatrix} \cos\xi & \sin\xi & 0 \\ -\sin\xi & \cos\xi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{pmatrix}$$

(Since it is not involved in this transformation, you ignore the w -coordinate.)

Since the z rotation doesn't affect the z -coordinate and the x rotation doesn't affect the x -coordinate, you can see that $z_1 = -\sin\phi$. This tells you the value of ϕ . Having done that, you can see that $z_2 = -\cos\phi \sin\theta$, which gives you θ and $x_1 = \cos\phi \cos\xi$, which gives you ξ . The only problem is that there is more than one set of solutions to this system, so different sets of rotations will combine to give the right answer. This might seem trivial, but it does cause problems, particularly when interpolating transforms.

One advantage to working with the raw transformations underlying a transform is that they provide you with a validation system against rounding errors. Because of the redundancy in the transform matrix, small errors can accumulate to give a transform that is not valid if a small sheer factor is introduced. You can counteract this by checking a transform against its underlying transformations and ensuring that the two are true to one another. To some extent, this decreases the value of combining transforms into a single matrix. In this respect, it is important to remember that the main value occurs when applying a transform to multiple objects in a scene. In such cases, a validation isn't necessary. You need only to validate the transform when modifying it in some way.

Applying Transforms

Transforms simplify 3-D calculations enormously. This section provides a few examples of how this can be accomplished.

Creating Motion with Transforms

In practice, it's easiest to work with transforms by ignoring the full transform matrix and concentrating on the underlying raw transformations. In general, you'll have some node with a transform \mathbf{T} , and you'll want to move it somewhere else. The only time you need to use the full transform is when calculating the actual position of the vertices of a model. For example, the basic unit cube has eight vertices at the eight permutations of the homogeneous vector $(\pm 0.5 \ \pm 0.5 \ \pm 0.5 \ 1)^T$. You can use a transform to move this cube to any other position or size in space. Once you have calculated the appropriate transform, you multiply it by each of the vertices to determine the new location of the cube.

The same process also works for direction vectors. As a result, the vector along one side of the cube will be unaffected. However, you do need to be careful about normal vectors. Because the scale transform does not preserve angles between lines, a direction vector that is normal to a line or plane before transforming is unlikely to still be normal afterward if the transform includes a scaling element.

If you know that \mathbf{n} and \mathbf{v} are perpendicular vectors, to find a vector normal to the transformed vector \mathbf{Tv} , you must find some new transform \mathbf{T}_n , such that $\mathbf{T}_n \mathbf{n} \mathbf{Tv} = 0$. Recall from the definition of the dot product that since $\mathbf{n} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v}$, $(\mathbf{T}_n \mathbf{n})^T (\mathbf{Tv}) = 0$. Likewise, $\mathbf{n}^T \mathbf{T}_n^T \mathbf{Tv} = 0$. Since \mathbf{n} and \mathbf{v} are perpendicular by definition, if $\mathbf{T}_n^T \mathbf{T} = \mathbf{I}$, then the equation is solved. You can conclude, then, that $\mathbf{T}_n = (\mathbf{T}^{-1})^T$, the inverse transpose of \mathbf{T} .

Yet again, it should be stressed that since such calculations are mostly handled behind the scenes by a decent real-time 3-D program, you don't need to worry about them. Still, it is worth keeping them in mind. They become more relevant as you work more deeply into the details of collision detection.

To consider examples of how you can use transforms to perform a common task, consider the action of aligning an arrow to point at a particular position vector. Suppose the arrow is a model whose root position (its appearance under the identity transform) is as shown in Figure 18.2. Notice that it is pointing along the positive z -axis.

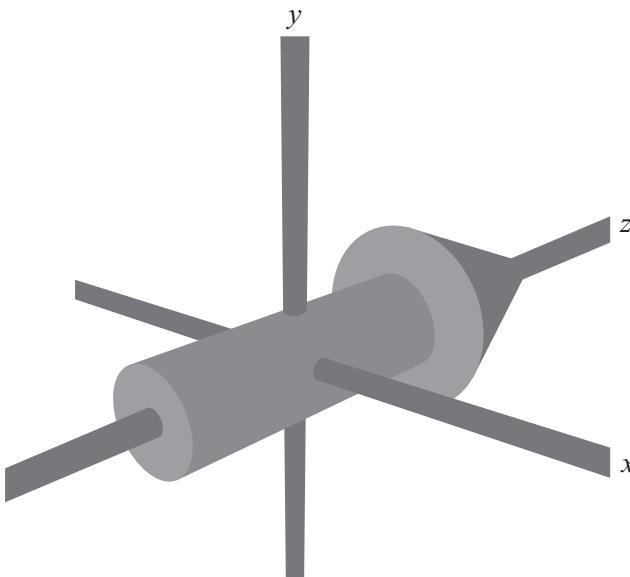


Figure 18.2

An arrow model in its untransformed state.

Imagine that the arrow in Figure 18.2 is located at position \mathbf{p} , pointing along the vector \mathbf{v} . To calculate \mathbf{v} , you start with the transform of the arrow. If you know the transform of the arrow, then \mathbf{v} is the normalized third column of the transform matrix. You want the arrow to point at a point \mathbf{q} instead along the vector \mathbf{v} , so you rotate it appropriately. To do this, you can first calculate the vector $\mathbf{q} - \mathbf{p}$. This is the new direction for the transform's z -axis. Normalized, you call this \mathbf{u} . Taking the cross product $\mathbf{u} \times \mathbf{v}$, you can find the axis around which you would like the arrow to rotate. By taking the dot product $\mathbf{u} \cdot \mathbf{v}$, you can find the cosine of the rotation angle.

Now all you need to do is to perform the rotation. The general matrix for rotating around an arbitrary axis is a little ugly. It appears as follows:

$$\mathbf{R} = \begin{pmatrix} c_+ + a_1^2 c_- & a_1 a_2 c_- - a_3 s & a_1 a_3 c_- + a_2 s \\ a_1 a_2 c_- + a_3 s & c_+ + a_2^2 c_- & a_2 a_3 c_- - a_1 s \\ a_1 a_3 c_- - a_2 s & a_2 a_3 c_- + a_1 s & c_+ + a_3^2 c_- \end{pmatrix}$$

where the axis is given by the vector $(a_1 \ a_2 \ a_3)^T$. If the angle is θ , then s , c_+ , and c_- are defined as $s = \sin\theta$, $c_+ = \cos\theta$, and $c_- = 1 - c_+$.

If you were working with non-homogeneous coordinates, you'd have to combine this with a translation to the origin, the rotation, and then the translation back. In this case you don't need to worry, however. Since translation is applied last, the translation and rotation parts of the homogeneous matrix are essentially independent. As a result, you can create a transform \mathbf{S} that has \mathbf{R} in the top-left portion and an identity position element.

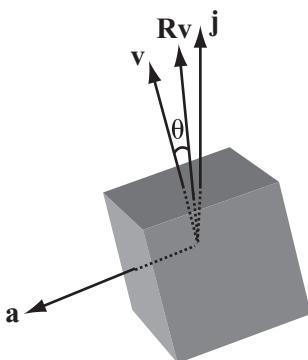
Given this progress, all you need to do is to combine the matrix \mathbf{S} with the original transform \mathbf{T} to apply the rotation. While in most instances this might be done readily, however, there is one potential problem. You must decide whether you should right- or left-multiply the transforms. To solve this problem, consider that since \mathbf{S} is to be applied after \mathbf{T} , you need to combine it on the left as \mathbf{ST} , not as \mathbf{TS} .

Note

In contrast, since it's generally more useful to apply the scale to the object in its unrotated form before applying rotations, consider that a scale transform is more likely to be applied before \mathbf{T} , and therefore right-multiplied with it.

Everything seems to be in order at this point, but still there is another complication. What happens if the node you are interested in is not symmetrical? For example, if the node is not an arrow model but a camera, then it's not enough to point at a particular location. You must also specify which way is up. Otherwise, your camera will spin in strange directions.

To resolve this problem, you perform not just one rotation, but two. First, you rotate the object, as before, to point in a particular direction. This might be along its local z -axis. You then rotate again around this direction vector to keep the preferred up-vector (usually the local y -axis) pointing as close to upward as possible. Figure 18.3 shows this process.

**Figure 18.3**

Keeping a preferred up-vector.

Note

There's nothing special about the up direction. You could just as easily align it to keep a preferred left side. In a gravitational world, however, it's often sensible to single out the vertical direction.

Determining the angle of rotation requires calculus. You are seeking an angle that maximizes $\mathbf{Rv} \cdot \mathbf{j}$, where \mathbf{R} is a rotation matrix around the (normalized) pointing direction $\mathbf{a} = \mathbf{q} - \mathbf{p}$, and \mathbf{v} is the current direction of the up-vector. Plugging this into the matrix for \mathbf{R} , you are trying to find θ that maximizes

$$(a_1 a_2 v_1 + a_2 a_3 v_3 + a_2^2 v_2)(1 - \cos\theta) + v_2 \cos\theta - (a_3 v_1 + a_1 v_3) \sin\theta$$

Differentiating, you see that θ must solve as

$$\tan\theta = \frac{(a_3 v_1 + a_1 v_3)}{(a_1 a_2 v_1 + a_2 a_3 v_3 + (a_2^2 - 1)v_2)}$$

This yields two possible values for θ . One gives the angle to point down. The other gives an angle to point up. Also, if the numerator and denominator are both zero, you are in a situation in which the pointing vector \mathbf{a} is parallel to \mathbf{j} , so all directions are equally far from pointing upward.

Interpolation

One of the most powerful features of transforms is the ability to create smooth motion by *interpolating* between two matrices. For example, you might want to create a camera that automatically follows a race car. If you place the camera at a constant distance relative to the car, the result is that the camera provides a view that makes it appear that it is tied on the back of the car. Instead of this, you want it to appear as though the camera is mounted on a helicopter smoothly following along behind the car.

To do this, you can calculate two transforms. The first is the current transform \mathbf{C} of the camera. This is something that you are likely to already know. The second is the target transform \mathbf{T} , which is at a set position relative to the moving car. At this point, instead of moving the camera to the new transform, you move it part way along, to some transform $t\mathbf{C} + (1 - t)\mathbf{T}$.

The effect of this change is to make the camera ease into position. This strategy works particularly well when following an object that is moving continually, like a car. A similar process can be used for following a third-person perspective character in a game like *Super Mario World* or *Tomb Raider*. In such cases, however, calculating the target transform is more complicated, since you have to deal with the possibility that other objects or landscape features are in the way. Generally, camera controls are one of the most difficult aspects of game design and can often make or break the game in terms of playability.

Quaternions

As mentioned previously, there are simple ways to handle rotations. The key concept is the *quaternion*. The quaternion is a special kind of 4-D vector. A quaternion \mathbf{q} can be written as $(w \ x \ y \ z)^T$. Alternatively, it can be written as $w + xi + yj + zk$, where the numbers i , j , and k are related to the imaginary number i . The imaginary number i is defined to be the square root of -1 .

In this case, i , j , and k are *orthogonal imaginary numbers*, in the sense that squaring any of them gives -1 , and together they satisfy the following equations: $ij = -ji = k$; $jk = -kj = i$; $ki = -ik = j$. No real numbers satisfy these equations. They are not numbers in that sense. Instead, they are mathematical abstractions that do what you want them to do. What they generate is a kind of imaginary vector.

The quaternion \mathbf{q} may also be written in the form $w + \mathbf{v}$, where \mathbf{v} is the vector $(x \ y \ z)^T$. You also define the conjugate of \mathbf{q} to be the quaternion $w - \mathbf{v}$. Having defined your strange imaginary basis, you can calculate the product of two quaternions:

$$\begin{aligned}
 (w_1 + x_1i + y_1j + z_1k)(w_2 + x_2i + y_2j + z_2k) &= w_1(w_2 + x_2i + y_2j + z_2k) \\
 &+ x_1i(w_2 + x_2i + y_2j + z_2k) + y_1j(w_2 + x_2i + y_2j + z_2k) + z_1k(w_2 + x_2i + y_2j + z_2k) \\
 &= (w_1w_2 + w_1x_2i + w_1y_2j + w_1z_2k) + (x_1w_2i - x_1x_2 + x_1y_2k + x_1z_2j) \\
 &+ (y_1w_2j - y_1x_2k - y_1y_2 + y_1z_2i) + (z_1w_2k - z_1x_2j - z_1y_2i - z_1z_2) \\
 &= (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) + (w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)i \\
 &+ (w_1y_2 + y_1w_2 - z_1x_2 + x_1z_2)j + (w_1z_2 + z_1w_2 + x_1y_2 - y_1x_2)k
 \end{aligned}$$

As unpleasant as this product might appear, it can be simplified if you use the vector form. If you say $\mathbf{q}_1 = w_1 + \mathbf{v}_1$ and $\mathbf{q}_2 = w_2 + \mathbf{v}_2$, then the product is given by

$$\mathbf{q}_1\mathbf{q}_2 = w_1w_2 + \mathbf{v}_1 \cdot \mathbf{v}_2 + w_1\mathbf{v}_2 + w_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2$$

Notice that because of the cross-product component, the product of quaternions is not commutative.

Armed with this result, you can take the product of a quaternion with its conjugate:

$$\mathbf{q}\bar{\mathbf{q}} = w^2 - \mathbf{v} \cdot \mathbf{v} - \mathbf{v} \times \mathbf{v}$$

Since for any vector, $\mathbf{v} \times \mathbf{v} = 0$, you can reason that

$$\mathbf{q}\bar{\mathbf{q}} = w^2 - \mathbf{v} \cdot \mathbf{v}$$

And because $i^2 = j^2 = k^2 = -1$, you find that $\mathbf{q}\bar{\mathbf{q}} = \mathbf{q} \cdot \mathbf{q} = |\mathbf{q}|^2$. The inverse (both left and right) of a quaternion can be given by $\mathbf{q}^{-1} = \frac{\bar{\mathbf{q}}}{|\mathbf{q}|^2}$, or in particular for a unit quaternion, $\mathbf{q}^{-1} = \bar{\mathbf{q}}$. The product of a quaternion with its inverse is the quaternion $(1 \ 0 \ 0 \ 0)^T$, which is equal to the scalar 1.

What does all this mathematical work have to do with rotations? It turns out that if you consider the vector \mathbf{u} to be a quaternion with a zero scalar part, and if you calculate the product $\mathbf{q}\mathbf{u}\mathbf{q}^{-1}$ for some quaternion \mathbf{q} , you arrive at the rotation of \mathbf{v} about a particular axis.

Notice first that since the product $\mathbf{q}\mathbf{u}\mathbf{q}^{-1}$ is the same for any scalar multiple of \mathbf{q} , you can consider \mathbf{q} to be a unit quaternion $\mathbf{s} + \mathbf{v}$, whose inverse is $\bar{\mathbf{q}} = \mathbf{s} - \mathbf{v}$. This gives the product as

$$\begin{aligned}\mathbf{q}\mathbf{u}\mathbf{q}^{-1} &= (-\mathbf{v} \cdot \mathbf{u} + s\mathbf{u} + \mathbf{v} \times \mathbf{u})\bar{\mathbf{q}} \\ &= -s\mathbf{v} \cdot \mathbf{u} + (s\mathbf{u} + \mathbf{v} \times \mathbf{u}) \cdot \mathbf{v} + s(s\mathbf{u} + \mathbf{v} \times \mathbf{u}) + (\mathbf{v} \cdot \mathbf{u})\mathbf{v} - (s\mathbf{u} + \mathbf{v} \times \mathbf{u}) \times \mathbf{v} \\ &= \mathbf{v} \cdot (\mathbf{v} \times \mathbf{u}) + s^2\mathbf{u} + 2s(\mathbf{v} \times \mathbf{u}) + (\mathbf{v} \cdot \mathbf{u})\mathbf{v} - (\mathbf{v} \times \mathbf{u}) \times \mathbf{v}\end{aligned}$$

By the identities of the dot and cross products, the first term is zero and the last term is $|\mathbf{v}|^2 - \mathbf{u}(\mathbf{v} \cdot \mathbf{u})\mathbf{v}$, so you have

$$\mathbf{q}\mathbf{u}\mathbf{q}^{-1} = (s^2 - |\mathbf{v}|^2) \mathbf{u} + 2s(\mathbf{v} \times \mathbf{u}) + 2(\mathbf{v} \cdot \mathbf{u}) \mathbf{v}$$

If you say $t = |\mathbf{v}|$ and set \mathbf{a} to the normalized vector $\frac{\mathbf{v}}{t}$, then you can rewrite this as

$$\mathbf{q}\mathbf{u}\mathbf{q}^{-1} = (s^2 - t^2) \mathbf{u} + 2st(\mathbf{a} \times \mathbf{u}) + 2t^2(\mathbf{a} \cdot \mathbf{u}) \mathbf{a}$$

This represents a rotation about the axis \mathbf{a} , with an angle θ given by $\cos\theta - s^2 = t^2$, $\sin\theta = 2st$, and $1 - \cos\theta = 2t^2$. Of these, you need only the last, since the first two together confirm that $1 - \cos\theta = 2t^2$, which you know because \mathbf{q} is of unit length. The last is equivalent to saying that $s^2 + t^2 = 1$, which implies that $s = \cos\left(\frac{\theta}{2}\right)$. You can sum all this up by saying that a rotation around an axis \mathbf{a} with angle θ can be calculated as follows using the quaternion:

$$\mathbf{q} = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\mathbf{a}$$

This is a fairly significant improvement on the matrix form. One reason is that far fewer calculations are involved in the transformation. Another reason is that it's much easier to see the relationship between the rotation and the quaternion than it is with the matrix. Also, just as with matrices, you can interpolate between two quaternions by taking a linear combination of them, although the rate at which the interpolation occurs is not the same as the rate at which the angle varies.

Parents and Children

In addition to combining transforms to move objects from one place to another, you can use them to define relationships between different nodes. This allows you to move whole groups of nodes together by varying a single transform, just as varying a node's transform can affect all the vertices of its model mesh.

You arrange all the nodes of the 3-D world in the form of a tree. In the tree, each node has exactly one parent and any number of children. The topmost node is the world itself, and each of the world node's children has its own transform relative to the world. But the transform of a child node is applied relative to its parent. In other words, if the child's relative transform is \mathbf{C} and the parent's transform is \mathbf{P} , then the child's transform relative to the world's coordinate system is \mathbf{CP} .

For example, suppose one model is sitting in the world with a world transform consisting of a rotation about the x -axis and a uniform scale. Any child of this model will be rotated and scaled before its own transform is applied. As a result, the model and its children act as a group. If the parent model is translated, rotated or scaled, its children, grandchildren, and so on will move along with it.

Note

In this section, what applies to translation in most cases can be applied to rotation and scaling.

Such movement means that there is a potential ambiguity when transforming an object. Say that you want to move an object 5 units along the z -axis. Which z -axis does that mean? It could be the node's "local" z -axis, or its z -axis relative to its parent. It could also be the actual z -axis of the world. Figure 18.4 illustrates this situation.

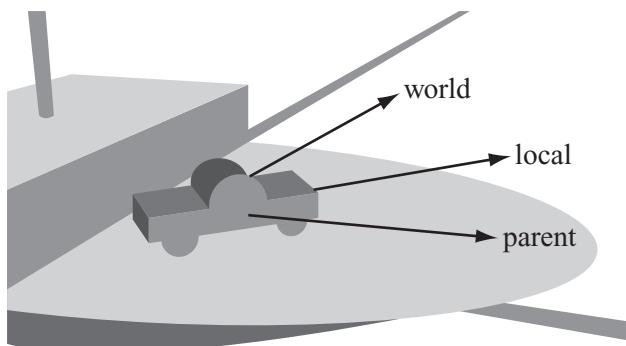


Figure 18.4

Relative motion in 3-D space.

Figure 18.4 represents a model of a car. The parent of the car is a ship on which the car is sitting. The car was modeled with its z -axis pointing along the body of the car. If you want to translate the car by 5 units along a z -axis, this could mean any of the directions indicated. The first is the car's own z -axis, which is pointing along the world vector $(1 \ 0 \ 1)^T$. The second is the ship's z -axis, which is pointing along the world's x -axis. The third is the world's z -axis. Ultimately, you might want to use the z -axis of any other node in the scene.

Fortunately, each of these is easy to calculate. The z -axis of the node is given by the last column of its complete world transform matrix. In general, any direction vector can be found simply by transforming it with the node's world transform. Similarly, while with a little work you can determine the direction of the parent's z -axis or any other model's, working out the world's z -axis is even easier.

Commonly, you want to go the other way around. You need to know how to change the node's personal transform in order to make it move along a particular vector. In this case, the simplest situation involves transforming the object in its local reference frame. Transforming locally is a matter of altering the transform of the object.

To alter the transform relative to the object's parent's transform is a little harder. To understand how this is so, suppose you want to move the object 5 pixels relative to the parent's z -axis. You must first calculate the vector in the child's reference frame. Expressed differently, you need a vector \mathbf{v} such that $\mathbf{Cv} = \mathbf{k}$, which is given by $\mathbf{v} = \mathbf{C}^{-1}\mathbf{k}$. In English, this means that the vector \mathbf{v} , when "untransformed" into parent-space, is equal to the z -vector \mathbf{k} . Notice that this is the same independent of the parent's own transform. Similarly, to translate relative to a world-vector, you must invert the complete world transform of the child node.

Note

The vector \mathbf{v} does not need to be of unit length, since any of the transforms involved may include a scaling element. Actually, this is convenient, for it means you can perform your translation of 5 units without worrying about scale. To do so, you multiply by the already scaled vector \mathbf{v} . But for rotations you need to remember to normalize the axis vector before performing the rotation.

Exercise

EXERCISE 18.1

Create functions that will apply a translation, scale, or rotation for a particular node relative to the world, the node's parent, or the node itself. You should think about which method of storing transform information is most useful: as a single matrix or as three transformation vectors? Using quaternions for rotations?

Summary

In this chapter you have seen how homogeneous coordinates and mathematical gizmos like quaternions can be used to create a simple representation of a node's disposition in 3-D space. In the process, you've also learned how to create smooth motion incorporating simultaneous rotation, translation, and scale transformations by interpolating between transforms.

In the next chapter, you'll take one final look at collision detection by extending your previous work into the third dimension.

You Should Now Know

- The meaning of the term *transform* and how it is represented in various forms: as a single matrix, as a combination of separate matrices, and as three vectors
- How *quaternions* can be used to create a simpler representation of rotations
- How transforms can be combined to move objects from place to place, and to create groups of objects with a fixed position relative to one another

CHAPTER 19

3-D COLLISION DETECTION



In This Chapter

- Overview
- Colliding Worlds
- Colliding Footballs
- Colliding Boxes
- Colliding Cans
- Varieties of Collisions
- Resolving Collisions in Three Dimensions

Overview

Contrasting 2-D to 3-D, while there is nothing fundamentally different about the techniques of collision detection in 3-D, adding another dimension does make things more complicated. For one thing, the range of shapes in 3-D is much more complex. This is especially so with respect to the cylinder and the cone. Also, there are more ways for two objects *not* to collide. For example, while two non-parallel straight paths in 2-D will always intersect, in 3-D they will usually miss each other.

Even if complexities do arise, it remains that most of the techniques you'll examine in this chapter are based on previous explorations. Further, since this chapter concerns conceptual foundations, you won't use as many code examples as appeared in previous chapters. The assumption is made that you can use previously presented examples to apply the mathematics for yourself. Your main goal is to create a toolset for dealing with the topics presented in this chapter, and you'll look at examples of techniques that are useful in different circumstances.

Colliding Worlds

The simplest collisions in 3-D are between balls or spheres. Spheres are the 3-D analogs of the 2-D circles. Most of the techniques for detecting collisions with spheres are direct translations of the techniques that apply to 2-D objects.

Spheres

A sphere is defined as the set of points at a constant distance r from the center. Mathematically, a circle is just a specific kind of sphere. You can have a sphere in any number of dimensions. In more than three dimensions, the sphere is often called a *hypersphere*. To mathematically designate a sphere, you use the terminology $S(c, r)$.

Identifying a point on a sphere is more complex than identifying a point on a circle. In vector terms, the point on a sphere is the point $c + r\mathbf{v}$, where \mathbf{v} is any vector of unit length. Unfortunately, no neat trigonometric expression is available to characterize \mathbf{v} . To accomplish this task, first think of a particular unit vector, such as \mathbf{i} , and then imagine rotating it by a certain amount in two directions. This gives you a general expression for these vectors:

$$\begin{aligned}
 \mathbf{v} &= \mathbf{R}_y \mathbf{R}_z \mathbf{i} \\
 &= \begin{pmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \\
 &= \begin{pmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} \\
 &= \begin{pmatrix} \cos\phi & \cos\theta \\ \sin\theta \\ \sin\phi & \cos\theta \end{pmatrix}
 \end{aligned}$$

For a one-to-one mapping between values of θ and ϕ and points on the sphere, θ should take values between 0 and 2π , and ϕ should take values between 0 and π . Think of them as longitude and latitude. Of course, several other variations on this theme will serve just as well.

A Moving Sphere and a Wall

When seeking to detect a moving sphere's position with relation to a wall or plane surface, as illustrated by Figure 19.1, you formulate the problem mathematically as the intersection of a sphere $S(c, r)$, moving along a vector v , with an infinite plane defined by the point p and the normal vector n . Given this start, you now identify the plane as plane $P(p, n)$.

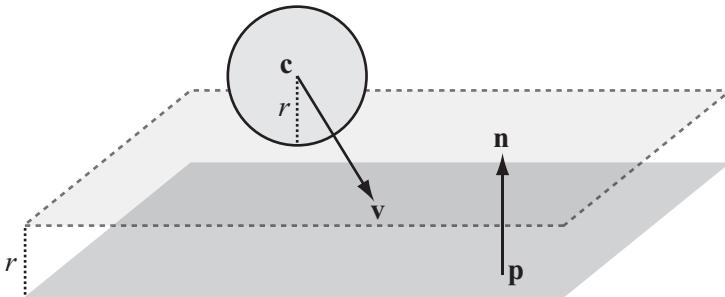


Figure 19.1

A moving sphere and a plane.

As is evident if you review the discussion in Chapter 8, the technique used to describe the collision of a sphere and a plane is almost identical to the technique used to describe the collision of a circle with a line. First of all, you find the value $n \cdot (c - p)$. If this value is positive, the sphere lies on the positive normal side of the plane. If the value is not positive, it lies on the negative normal side, and in this case you replace n with $-n$. To make the problem simpler, drawing from a problem solved in Chapter 17, you can find the intersection of a point with a plane. To accomplish this, you offset the plane by the vector rn and find the intersection of the line $c + tv$ with the new plane $P(p + rn, n)$.

Often you're only looking for a one-way collision with a plane. In particular, when calculating collisions between solid objects, you assume that the normal to each face is pointing outward. This is used in visibility determination. When the normals are pointing outward, you can simplify the problem involving the sphere and the plane because you no longer need to calculate which side of the plane the sphere is on. The sphere will collide with the plane only if the dot product $\mathbf{v} \cdot \mathbf{n}$ is negative.

A Sphere and a Moving Point or Two Spheres

Consider situations in which a sphere collided with a moving point or another sphere. Again drawing from the discussion in Chapter 8, to start with, picture the sphere as stationary at the origin, with a particle or object at \mathbf{p} moving along the vector \mathbf{v} . You now need to solve the equation $\mathbf{p} + t\mathbf{v} = \mathbf{r}$, which is equivalent to

$$(\mathbf{p} + t\mathbf{v}) \cdot (\mathbf{p} + t\mathbf{v}) = r^2$$
$$\mathbf{p} \cdot \mathbf{p} + 2t\mathbf{p} \cdot \mathbf{v} + t^2\mathbf{v} \cdot \mathbf{v} = r^2$$

With this you have a quadratic equation in t . You can solve it using the quadratic formula, and after doing so, you arrive at two possible collision points. To identify the point of collision, you focus on the smallest positive value. If it turns out that one value is negative, then your particle or object is inside the sphere to start with.

Having solved this problem, it's simple to extend it to two general spheres colliding. As with the case of two circles, the problem of two spheres of radius r and s colliding is equivalent to a particle colliding with a single sphere of radius $r + s$.

The Point of Contact

As you move from 2-D to 3-D, given the extra dimension, you find that objects no longer collide along a line. Instead, they meet along a collision plane. The collision normal is the most useful means of detecting the collision. When resolving the collision, only the component of motion normal to the collision is affected. The tangential component is unchanged.

As with finding the collision normal with a circle, finding the collision normal with a sphere is straightforward. The normal lies along the line from the point of contact to the center. When two spheres collide, this line is directly along the vector between the two centers. When the sphere collides with a plane, then the plane's normal is the collision normal.

Colliding Footballs

Having dealt with collisions between planes and spheres, you are now prepared to examine collisions between irregular shapes. The discussion in this section, in this respect, still draws from Chapter 8, but rather than 2-D shapes, those in 3-D are considered.

Ellipsoids

An *ellipsoid* is the 3-D equivalent of the ellipse you explored in discussions of 2-D collisions. As with the relation between a circle and an ellipse in 2-D, in 3-D, one easy way to think of an ellipsoid is as a sphere that has been transformed by a general scale in three dimensions. In this respect, a point on an ellipsoid centered on the origin has the general form $\mathbf{v} = \mathbf{RSi}$. In fact, if you include position, orientation, and scale, you can exactly describe a general ellipsoid using a general transform \mathbf{T} , applied to the unit sphere described by $(\cos\phi \cos\theta, \sin\theta, \sin\phi \cos\theta)^T$.

Note

An ellipsoid with a circular cross section along one axis (one with two of its scale factors equal) is called a *spheroid*. A spheroid can be either oblate (flattened like a UFO) or prolate (extended like a football).

In anticipation of the discussion that follows, the term *transform* is used somewhat loosely in this chapter. At times it is used to describe just the rotation and scale parts of the transform proper, in the form of a 3×3 matrix. To keep the position part of the transform separate, for clarity you employ the notation $E(\mathbf{p}, \mathbf{T})$. In practice, however, it's more efficient to use the full 4×4 transform. Additionally, as with the 2-D ellipse, you can also describe the ellipsoid by means of a list of its principal axes and the values a, b, c . The two descriptions are equivalent.

An Ellipsoid and a Moving Point or Plane

Calculating a collision between an ellipsoid $E(\mathbf{p}, \mathbf{T})$ moving along a vector \mathbf{v} and a point or plane is best accomplished by transforming space so that the ellipsoid becomes a sphere. Given the description of the ellipsoid in the previous section, to accomplish this, you must invert the transform \mathbf{T} . After you invert the transform, you then look for the collision between the transformed plane and the unit sphere starting on $\mathbf{T}^{-1}\mathbf{p}$, with a displacement of $\mathbf{T}^{-1}\mathbf{v}$.

This understanding puts you in a position to again approach a problem left dangling at the end of Chapter 8. This problem involved calculating the normal at the ellipsoid (or ellipse) surface. To proceed, recall the discussion from the previous chapter concerning normals. In the case of the plane, the transformed normal $\mathbf{T}^{-1}\mathbf{n}$ is not normal to the transformed plane. Instead, it is necessary to use the inverse transpose matrix, which is the transpose of \mathbf{T} . Similarly, after arriving at the transpose, to determine the collision normal in world space, you must apply the inverse transpose $(\mathbf{T}^{-1})^T$ to any collision normals. The `ellipsoidPlaneCollision()` function encapsulates these operations:

Here's an example of this method in code:

```
function ellipsoidPlaneCollision(ell, pl)
    set inverseTransform to inverseMatrix(ell.matrix)
    set inverseTranspose to transpose(ell.matrix)
    set planePoint to matrixMultiply(inverseTransform, pl.refPoint-ell.pos)
    set circleVel to matrixMultiply(inverseTransform,
                                    ell.displacement-pl.displacement)
    set normal to matrixMultiply(inverseTranspose, plane.normal)
    set t to circlePlaneCollision(circlePos, 1,
                                   circleVel, planePoint, normal)
    return t
end function
```

Two Ellipsoids

As has been anticipated by the discussion in Chapter 8, the next step in the 3-D version of collisions involves collision between two ellipsoids. Put formally, the question is one of how to find the intersection of two ellipsoids. Based on the difficulties involved with detecting collision in 2-D between two ellipses, it becomes evident that you cannot solve this problem algebraically. However, algebra offers an excellent start. Consider how you can describe this problem in terms of matrix algebra. Suppose your ellipsoids are $\mathbf{E}_1(\mathbf{0}, \mathbf{T}_1)$ and $\mathbf{E}_2(\mathbf{p}, \mathbf{T}_2)$ and the relative velocity is \mathbf{v} . You're looking for two unit vectors \mathbf{u}_1 and \mathbf{u}_2 , which satisfy $\mathbf{T}_1\mathbf{u}_1 = \mathbf{T}_2\mathbf{u}_2 + \mathbf{p} + t\mathbf{v}$. Taking \mathbf{T}_1 to the other side of the equation, you have $\mathbf{u}_1\mathbf{T}_1^{-1}\mathbf{T}_2\mathbf{u}_2 + \mathbf{T}_1^{-1}\mathbf{p} + t\mathbf{T}_1^{-1}\mathbf{v}$.

Given this formulation of the problem, you face five unknowns. These are the unit vectors \mathbf{u}_1 and \mathbf{u}_2 , with two degrees of freedom each, and the scalar t . However, to narrow down the solution, you can add one more condition. This condition is that the two ellipsoids must meet at a point. When the ellipsoids meet at a point, at the point of contact the normals must be parallel. Since the normal of an ellipsoid at the point $\mathbf{T}\mathbf{u}$ is the point $(\mathbf{T}^{-1})^T\mathbf{u}$, you arrive at

$$(\mathbf{T}_1^{-1})^T\mathbf{u}_1 = (\mathbf{T}_2^{-1})^T\mathbf{u}_2$$

Inverting the first transform, this expression becomes

$$\mathbf{u}_1 = \mathbf{T}_1^T (\mathbf{T}_2^{-1})^T \mathbf{u}_2,$$

which you can write as $\mathbf{u}_1 = \mathbf{M}\mathbf{u}_2$. With this version, you can now eliminate \mathbf{u}_1 from the first equation, which tells you that

$$\mathbf{M}\mathbf{u}_2 = \mathbf{T}_1^{-1}\mathbf{T}_2\mathbf{u}_2 + \mathbf{T}_1^{-1}\mathbf{p} + t\mathbf{T}^{-1}\mathbf{v}$$

Since \mathbf{u}_2 is known to be a unit vector, between them, these two results form a system of three independent equations in three unknowns. Containing a mixture of trigonometric and linear terms, the equations cannot be solved algebraically. Instead, a numerical approximation method must be used. However, it is still possible to simplify things if the ellipsoids are the same size (have the same transform \mathbf{T}) or a scalar multiple of one another (transforms \mathbf{T} and $a\mathbf{T}$). If either of these applies, you can invert the transform matrix to turn the problem into a collision of two spheres. This is a problem that you have already solved:

$$\begin{aligned}\mathbf{u}_1 - a\mathbf{u}_2 &= \mathbf{T}^{-1}\mathbf{p} + t\mathbf{T}^{-1}\mathbf{v} \\ (\mathbf{T}^{-1})^T (\mathbf{u}_1 \pm a\mathbf{u}_2) &= 0\end{aligned}$$

The second equation indicates that the normals are parallel, and from it you know the two results, either $\mathbf{u}_1 = a\mathbf{u}_2$ or $\mathbf{u}_1 = -a\mathbf{u}_2$. With the first of the results, one ellipsoid is inside the other. With the second of these results, you can proceed more directly to the solution. By substitution in the first equation, you find that

$$\begin{aligned}(1 + a)\mathbf{u}_1 &= \mathbf{T}^{-1}\mathbf{p} + t\mathbf{T}^{-1}\mathbf{v} \\ |\mathbf{T}^{-1}\mathbf{p} + t\mathbf{T}^{-1}\mathbf{v}| &= 1 - a\end{aligned}$$

This follows since \mathbf{u}_1 is a unit vector. The remainder of the method follows for the collision of two spheres.

Colliding Boxes

The next topic of consideration in the 3-D realm is the *cuboid*. The cuboid is the analog to the 2-D rectangle. However, as with the generalization of rectangles to other convex polygons, the techniques for detecting collisions with a cuboid generalize to shapes with any number of flat faces connected by edges, such as cubes, pyramids, cut diamonds, or soccer balls. Such shapes are referred to as *convex polyhedrons*.

Boxes

A cuboid has eight vertices joined together by 12 edges, making six faces in all. Think of a brick. As with ellipsoids, you can consider a general cuboid to be a transform \mathbf{T} applied to a standard unit cube centered on the origin. Its eight vertices are the eight different combinations of $(\pm 0.5 \ \pm 0.5 \ \pm 0.5)^T$. Knowing this transform, it is simple to test whether a particular point \mathbf{p} is inside the cuboid. To do so, you find $\mathbf{T}^{-1}\mathbf{p}$ and determine whether each of its coordinates has absolute value less than 0.5.

A Box and a Moving Point

Knowing whether coordinates have absolute values less than 0.5 gives you one method for finding the point of intersection of a line $\mathbf{p} + t\mathbf{v}$ with the box whose transform is \mathbf{T} . You're searching for the smallest value of t such that $\mathbf{T}^{-1}(\mathbf{p} + t\mathbf{v}) = \mathbf{T}^{-1}\mathbf{p} + t\mathbf{T}^{-1}\mathbf{v} + t\mathbf{w}$ lies within the cube. This amounts to six inequalities:

$$-0.5 < q_1 + tw_1 < 0.5$$

$$-0.5 < q_2 + tw_2 < 0.5$$

$$-0.5 < q_3 + tw_3 < 0.5$$

An efficient algorithm exists for solving systems of linear inequalities such as this one. It is called the *simplex algorithm*. However, this algorithm is very specific to cuboids and is beyond the scope of the current discussion. A more general method is to check for intersection with each of the faces of the cuboid. As with the rectangle, as mentioned previously in this chapter, if you use the dot product of the face normal with the collision vector, you can immediately discount all faces on the other side of the cube from the particle.

How to calculate the point of intersection of a particle with a plane has already been discussed. One outcome of the discussion was that you were provided with a way to quickly calculate the three possible collision points with the infinite planes through each face. You can then check just these points to find which of them, if any, lies within the rectangle of the face. The check involves a method that will work for any planar convex polygon. As illustrated in Figure 19.2, you test whether the point lies on the correct side of each of the edges of the rectangle.

As Figure 19.2 shows, if you find the dot products $\overrightarrow{AP} \cdot \mathbf{n}$, and $\overrightarrow{AC} \cdot \mathbf{n}$, you can test if the point P lies on the correct side of AB . If these have the same sign (if their product is positive), then P and C must lie on the same side of AB . If you perform this test for each of the sides of $ABCD$, you will show that P is inside the rectangle.

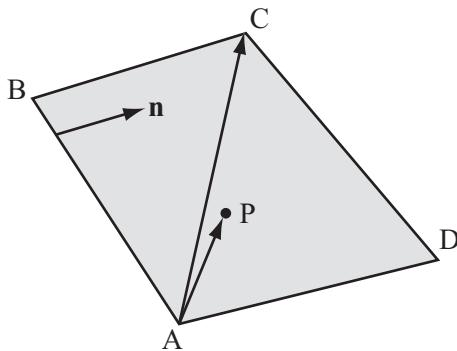


Figure 19.2

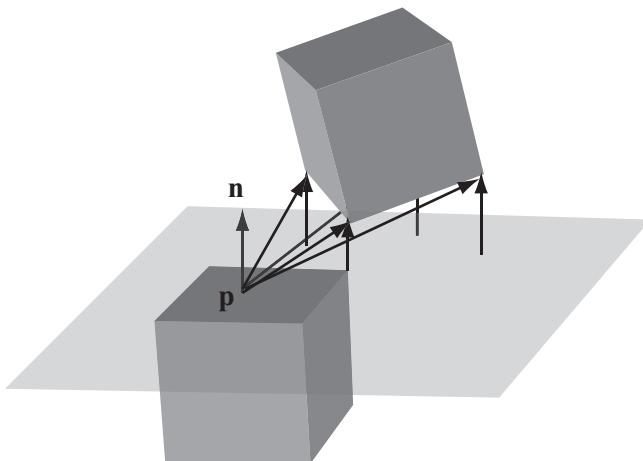
Testing whether a point lies inside a polygon.

The same basic method works for any polyhedron. However, if the faces of the polyhedron are not convex, you must use a more general technique for testing whether P lies on the face. One such method is the raycasting technique discussed in Chapter 10.

Two Boxes

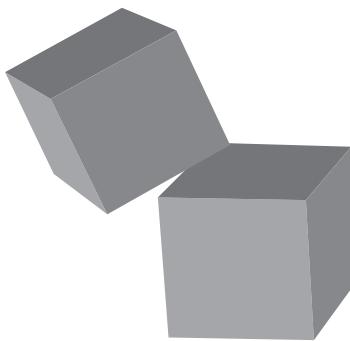
With two colliding boxes, there's not really much of an alternative to face-by-face checking or, more specifically, face-by-vertex checking. When dealing with boxes at any possible size and angle, several possible face-to-vertex collisions might be used, but with each of them, one of the vertices of one box is in contact with one of the faces of the other. This presents a problem since you don't need to check all possible vertices, only those at the leading edge of the cuboid. In fact, unless the cuboids are aligned, for each face there is only one vertex that can collide with it. As shown in Figure 19.3, this is the vertex nearest to the face in the direction of the face normal.

You can calculate the colliding vertex for each face by calculating the distance of each vertex v of one cube to each face (p, n) of the other. This is the minimum value of $(v - p) \cdot n$. If any of the distances is negative, then there is no possible collision on this face. The same goes for any convex polyhedron. If two vertices are the same distance from the face, then either of them could collide.

**Figure 19.3**

The colliding vertex.

Although you might eliminate much work by calculating distances, it remains that quite a few possible collisions will remain to be checked. You've also missed out another possible collision, as illustrated in Figure 19.4. This collision occurs when two boxes collide at the edges. This also includes the cases in which an edge of one box collides with the face of another. In this case, there is no vertex-to-face collision at all. A completely different calculation arises.

**Figure 19.4**

Edge-to-edge box collision.

Approaching the different calculation, as illustrated by Figure 19.5, you can consider the parallelogram swept out by the first edge as it travels along the vector v . If this parallelogram intersects the second edge, then, as Figure 19.5 shows, a collision at point c occurs.

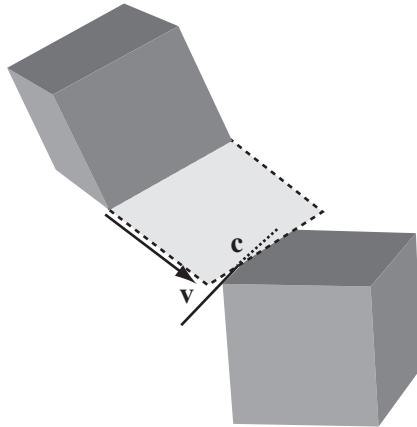


Figure 19.5

Calculating the collision of two lines.

As it is, then, Figure 19.5 depicts a line-polygon collision of the kind already discussed. A collision like this is only possible if the normals of all four of the faces involved have the correct dot product with the velocity vector. The vector is positive for the moving box and negative for the box that is not moving. And as with the vertex-face collisions, you're only interested in the closest edges.

Note

The task of writing a function to show how to detect these collisions will be deferred until Chapter 23. In that context, a specialized function is developed that tests for collisions between an axis-aligned box and a quadrilateral whose edges are aligned to the x - z plane, for use in tile-based 3-D games.

A Box and a Sphere

A sphere can collide with a box in different ways. The sphere might collide with a face, an edge, or a vertex. Collisions with a vertex are straightforward point-sphere calculations. These have already been covered, and with luck, you found the problem simple. Collisions with a face are also simple, and to deal with them, you can adapt the technique already discussed. Toward this end, you first calculate whether the sphere is colliding with the infinite plane containing the face. As with calculations involving polyhedrons, you then check if the collision point is inside the face.

Detecting collisions with an edge is slightly more problematic than detecting those involving vertices and faces. In essence, detecting collisions with an edge involves calculating a collision between a sphere and a line. To set up this calculation, as shown in Figure 19.6, consider a sphere $S(p, r)$ moving along the vector v , and a line (q, u) . How do you determine where they meet?

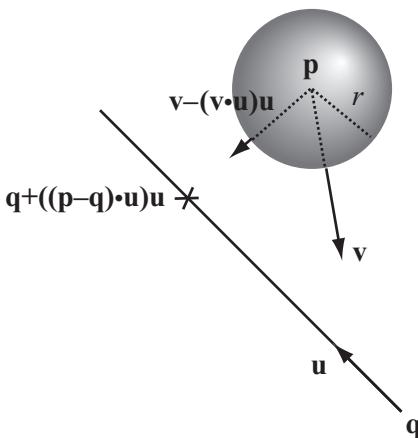


Figure 19.6

A sphere and a line.

One way to simplify the problem given in Figure 19.6 is to realize that you're interested only in the component of motion perpendicular to the line's vector. If you subtract the component $(v \cdot u)u$ from the velocity vector and add the vector $((p - q) \cdot u)u$ to q , you end up with a projection of the problem into 2-D space. Now you're looking for the intersection of a circle of radius r centered on p with the point $q + ((p - q) \cdot u)u$. You can then solve this problem using the approaches shown in Chapter 8 or Chapter 9.

As you proceed with your calculations, a few practical measures bear reviewing. To reduce the number of specific calculations needed, you can eliminate several of the sphere-box collisions on the basis that they are impossible. Consider, for example, that face-sphere collisions are possible only if the face normal is pointing in the opposite direction from the velocity. Edge-sphere and vertex-sphere collisions are possible only if at least one associated face normal is pointing in the opposite direction from the velocity.

Colliding Cans

The preceding discussion leads nicely into one last shape, one that has, alas, no 2-D equivalent. This is the *cylinder*. Cans are cylinders, as are pistons and simple drinking vessels.

Cylinders

Mathematically defined, a cylinder is the set of points at a distance r from some line. The line is the axis of the cylinder. The cylinder can be either infinitely long or have two circular caps whose normals are parallel to the axis. The cylinder's length is defined as the distance between the two caps. The intersection of an infinite cylinder with any plane perpendicular to the axis is a circle. If the intersection is with any other non-parallel plane, the result is an ellipse. You use the notation $C(\mathbf{p}, \mathbf{v}, r, l)$ to represent a cylinder of radius r centered on \mathbf{p} , with axis vector \mathbf{v} and length l .

A cylinder is a specific case of a more general shape called a *cone*. A cone is like a cylinder except that the radius of the two end caps is different. As shown in Figure 19.7, the radius of any circular cross-section is proportional to the distance along the axis.

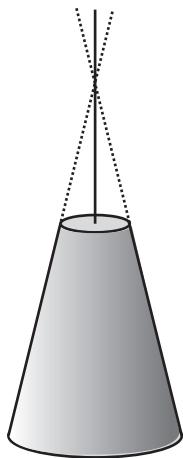


Figure 19.7

A cone.

Cylinders and cones are examples of what is called a *surface of rotation*. Such a surface is a 3-D shape that is formed by taking a 2-D profile and spinning it around an axis, like a pot forming on a potter's wheel. Most 3-D modeling software includes a utility for creating a surface of rotation. In some instances, the utility is referred to as a *lathe tool*. Chapter 21 provides an extended discussion of surfaces of rotation and related topics.

Note

Consider an infinite mathematical cone. Such a cone looks like two identical paper cones joined at the tips. If you cut or intersect such a cone with a plane, you'll get one of three shapes: an ellipse, a parabola, or an asymptotic shape called a *hyperbola*. A hyperbola is represented by the graph of $y^2 = 1 + x^2$. These three shapes are collectively known as *conic sections*.

A Cylinder and a Point or Sphere

In previous sections of this chapter, you have already done most of the work required to determine when a sphere collides with a sphere or point. A point colliding with the body of a cylinder is equivalent to a sphere colliding with a line. The previous section dealt with spheres colliding with lines. In general, a sphere with radius r colliding with the body of a cylinder of radius s is equivalent to a sphere of radius $r + s$ colliding with the cylinder's axis.

The end caps of cylinders introduce the only complication. As shown in Figure 19.8, to deal with this complication, you must find the collision point with a flat circle or disc of radius s and center c .

There are various ways to solve the problem depicted in Figure 19.8. All approaches are equivalent. One approach that gives a different perspective on the problem is often useful for collisions with flat (laminar) objects. It is related to the method used for circle-line collisions. Using this approach, consider that at any moment you can calculate the intersection of a sphere with the infinite plane containing a circle or other lamina. The intersection of a sphere with a plane is a circle whose radius depends on the distance of the sphere from the plane.

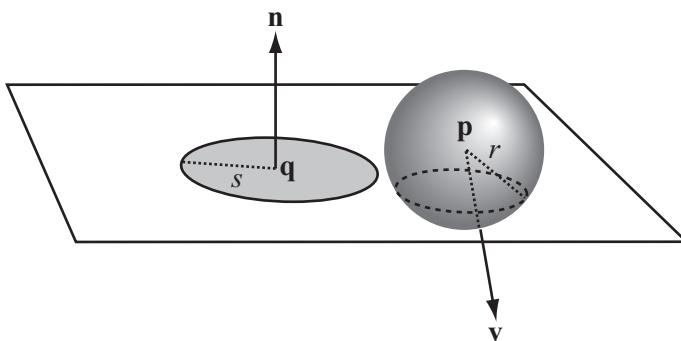


Figure 19.8

A sphere and a flat disc.

To present the mathematics, observe that a sphere $S(\mathbf{p}, r)$ moving along \mathbf{v} intersects a plane $P(\mathbf{q}, \mathbf{n})$. You define the distance of the sphere from the plane at time t to be $d = (\mathbf{p} - \mathbf{q} + t\mathbf{v}) \cdot \mathbf{n}$. The value can be positive or negative. If $|d| > r$, there is no intersection. If there is an intersection, the intersection is a circle with a radius of $rc = \sqrt{r^2 - d^2}$ and a center at $\mathbf{p}_c = \mathbf{p} + t\mathbf{v} - d\mathbf{n}$.

So much for how you can find the collision between a sphere and a disc. Now for the specifics of the collisions. Two possible collisions can occur. With the first, if $d = r$ and $\mathbf{p}_c - \mathbf{q} \leq s$, the sphere collides somewhere inside the disc. From this event, you arrive at the following:

$$\begin{aligned} (\mathbf{p} - \mathbf{q} + t\mathbf{v} - r\mathbf{n}) \cdot (\mathbf{p} - \mathbf{q} + t\mathbf{v} - r\mathbf{n}) &\leq s^2 \\ (\mathbf{p} - \mathbf{q} + t\mathbf{v}) \cdot (\mathbf{p} - \mathbf{q} + t\mathbf{v}) - 2r(\mathbf{p} - \mathbf{q} + t\mathbf{v}) \cdot \mathbf{n} + r^2\mathbf{n} \cdot \mathbf{n} &\leq s^2 \\ (\mathbf{p} - \mathbf{q} + t\mathbf{v}) \cdot (\mathbf{p} - \mathbf{q} + t\mathbf{v}) &\leq s^2 + r^2 \end{aligned}$$

If you find the point of collision of the sphere with the laminar plane, you can use this understanding to check whether it lies within the circle.

Note

If you have some experience in this area, you might know that same result can be found much more simply with the Pythagorean Theorem. Since one objective of this passage is to follow the logic presented in previous sections, the longer approach is used.

With respect to the second type of collision, one involving a sphere and the circumference of a disc, you know that the circle of intersection of the sphere and the laminar plane must be touching. In other terms, expressed mathematically, $|\mathbf{p}_c - \mathbf{q}| = r_c + s$. Since you're looking at a cylinder, you know that the sphere can collide only on one side of the disc. You also know that d is positive. This is not the case, however, with a literally flat disc, such as a DVD.

To find a collision with the circumference, you are looking for t such that

$$\begin{aligned} (\mathbf{p} - \mathbf{q} + t\mathbf{v} - r\mathbf{n}) \cdot (\mathbf{p} - \mathbf{q} + t\mathbf{v} - r\mathbf{n}) &= \left(\sqrt{r^2 - d^2} + s \right)^2 \\ (\mathbf{p} - \mathbf{q} + t\mathbf{v}) \cdot (\mathbf{p} - \mathbf{q} + t\mathbf{v}) - 2rd &= -d^2 + s^2 + 2s\sqrt{r^2 - d^2} \end{aligned}$$

To yield an equation in t , you can replace d by its full dot-product expansion. The only drawback is that the resulting function can't be solved algebraically. While not solvable algebraically, however, the resulting function does generate output that is nevertheless fairly smooth. Essentially, it is quadratic. With respect to the function itself, it shouldn't come as a surprise to find no algebraic solution. The cross-sectional circle is similar to a moving ellipse when viewed from the perspective of the disc. Still, the situation can change with changing parameters. If you're dealing with only a particle of zero radius, you can ignore the possibility of a collision along the circumference.

A Cone and a Sphere or Particle

To calculate the collision of a sphere with a cone, you must take into account the angle of the slope of the sides. As in previous passages of this chapter, while you can expand the cone by r to reduce the case of a colliding sphere to one of a colliding particle, care must be taken when calculating the point of impact.

For simplicity, assume that the cone's *apex* is at the origin. The apex is the point at which the radius of the cone is zero. If the cone is not infinitely long, this point might not be part of the physical cone. From the apex, the side of the cone spreads out at an angle α . If the particle is at $\mathbf{p} + t\mathbf{v}$ and the cone's normalized axis vector is \mathbf{u} , then you can calculate the particle's distance from the axis to be

$$d = \sqrt{(\mathbf{p} + t\mathbf{v}) \cdot (\mathbf{p} + t\mathbf{v}) - ((\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u})^2}$$

The radius of the cone on the plane containing the particle at the time given by t is

$$r = (\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u} \tan \alpha$$

When the particle collides, these two values must be equal, giving you an equation for t :

$$\begin{aligned} \sqrt{(\mathbf{p} + t\mathbf{v}) \cdot (\mathbf{p} + t\mathbf{v}) - ((\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u})^2} &= (\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u} \tan \alpha \\ (\mathbf{p} + t\mathbf{v}) \cdot (\mathbf{p} + t\mathbf{v}) - ((\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u})^2 &= ((\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u})^2 (\tan^2 \alpha + 1) \\ &= ((\mathbf{p} + t\mathbf{v}) \cdot \mathbf{u})^2 / \cos^2 \alpha \end{aligned}$$

As you can see, this is a fairly minor change to the equation for the cylinder. To transform to the case of the sphere involves similar operations, with the difference that in collisions with the end caps, you can no longer be sure that the distance from the sphere to the cap plane is positive at the moment of collision.

Two Cylinders

When you're looking at the collision of two cylinders, a good starting point is to consider the two cylinders as ellipses moving through a plane. You use this approach for collisions along the body of the cylinder rather than at the end caps. To use this approach, first choose a plane in space and project both cylinders onto this plane. It makes things easier if you make a plane that is perpendicular to one axis.

If you set up your problem so that the cylinders are aligned along the same axis, the problem is simple to solve. The problem is then equivalent to two moving circles. If you set up the problem so that the cylinders are not aligned on the same axis, the problem becomes more difficult, for you are then dealing with an elliptical collision. To solve such a problem, you must use numerical methods.

As illustrated by Figure 19.9, to calculate the projected ellipse onto a plane through a cylinder, you first calculate the center of the ellipse. The center of the ellipse is the point at which the plane intersects with the axis. You then have a reading on the *halfminor axis*. The halfminor axis of the ellipse is always the radius of the cylinder. It is directed along the cross product of the plane normal and the axis. By finding the cross product of this with the plane normal, you can find the vector of the major axis. Using the angle between this vector and the axis, you use trigonometry to find the length.

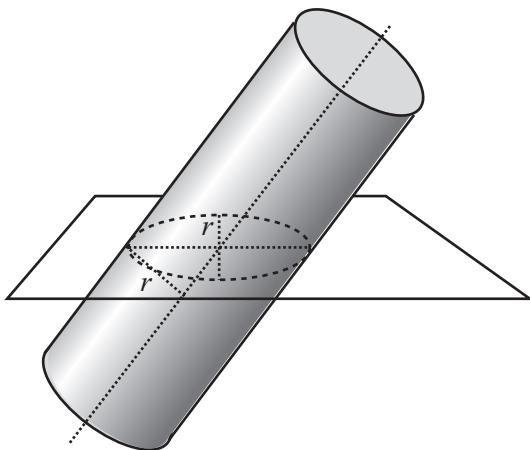


Figure 19.9

Projecting a cylinder to a plane.

Varieties of Collisions

As the discussion in the previous sections reveal, even simple shapes can involve complicated calculations, and some of these calculations lead to situations that offer no algebraic solution. To make things more appealing, for more complex shapes, you require a strategy that allows you to perform calculations in a realistic time scale. This leads to bounding spheres, ellipsoids, and boxes.

Bounding Spheres, Ellipsoids, and Boxes

As in 2-D, the most useful technique for calculating collisions of more complex shapes involves creating bounding volumes. A bounding volume is a shape that is known to contain the whole of your object. When the object is already very nearly the right shape, you can use the bounding volume as a proxy for collision calculations. If the object is not close to the right shape, then the bounding shape can at least be used to perform an initial, simpler collision check before performing a full triangle-by-triangle calculation.

The process of calculating a bounding volume in 3-D is similar to the process used in 2-D. To create a bounding sphere, for example, you can average all the vertices of your model to get the center and then calculate the radius as the maximum distance of one point from the center. As with 2-D calculations, this approach won't usually yield the smallest possible sphere, but it's quick and cheap. As discussed previously, bounding ellipsoids can be found by a process of factor analysis, and you can create both axis-aligned and object-aligned bounding boxes.

Collisions with an Arbitrary Mesh

For more collision detection involving complex shapes, generally no better method exists than checking collisions with an arbitrary mesh. The arbitrary mesh consists of a number of vertices joined together in triangles. The normal of each triangle is known and points "out" of the shape. A two-sided mesh has two sets of triangles, one pointing inward, the other outward. Since the direction of the normal is calculated according to the order of the vertices, when you are looking at the triangle, with its normal pointing toward you, the vertices are ordered in a clockwise direction. In many instances, this process can be made less demanding if you create a simplified shape as a proxy, with fewer triangles to calculate.

The technique for calculating collisions with a triangular mesh is essentially the same as for a box. You employ the technique illustrated in Figure 19.2 to determine whether a point is inside a triangle. You assume that if the point is in the plane of the triangle, if it's on the interior side of each of the three edges, then it is inside of the mesh. This means that the point has to be on the same side as the other vertex of the triangle. Its dot product with the cross product of the edge and the normal must be positive. To express this explicitly, if you set $\mathbf{n}_1 = \mathbf{n} \times (\mathbf{v}_2 - \mathbf{v}_3)$, then $(\mathbf{n}_1 \cdot (\mathbf{v}_1 - \mathbf{v}_2)) \cdot (\mathbf{n}_1 \cdot (\mathbf{p} - \mathbf{v}_2)) \geq 0$.

Resolving Collisions in Three Dimensions

Chapter 9 and other chapters addressed resolving 2-D collisions at some length. With respect to resolving 3-D collisions, the laws of physics don't change a great deal. Things bounce and collide the same way in both worlds. As a result, the techniques used for 2-D calculations carry across almost unchanged to 3-D calculations. It is important to keep in mind, however, that with 3-D calculations, you commence from a tangential plane rather than just a tangential line.

The one exception is rotation. As you've already seen, spin can take place in several directions. A spin in one direction can be conceived of as three spins in perpendicular directions. In more inclusive contexts, it would be necessary to take such phenomena into consideration. In this context, however, you can ignore it. The underlying concepts don't change. As it is, however, the calculations become more involved.

Exercise

EXERCISE 19.1

Translate one or more of the examples in the chapter into a concrete function, and include a function to resolve the collision. Most of the examples are given without code, but with the experience of 2-D collisions, you shouldn't find this challenge too hard for spheres and boxes.

Summary

One objective of this chapter has been to make it clearer how detecting linear collisions in 3-D can be approached. A continuing theme has been that your grounding in 2-D collision detection gives you an excellent start on 3-D collision detection. If you have found this to be so, you can feel pleased with yourself. It means that you understood the techniques already covered. If not, don't be discouraged. Circle back to earlier chapters and see what you've missed. You've looked now at a large number of different shapes and seen how you can make some calculations to detect collisions between them. You've also examined more general techniques that will work on any mesh. There is still plenty of work to do in order to fill in the gaps. The next chapter provides a high-speed tour of 3-D. Among its topics are how surfaces are given shape by light.

You Should Now Know

- The meaning of *sphere*, *ellipsoid*, *spheroid*, *cuboid*, *cylinder*, *cone*, and *mesh*
- How to calculate the intersection of each of these with a ray, or collision with a small particle
- How to calculate collisions between spheres and each of the others, and between pairs of similar objects
- How bounding volumes can be used as an extension of 2-D bounding shapes

CHAPTER 20

LIGHTING AND TEXTURES



In This Chapter

- Overview
- Light
- Materials
- Shading

Overview

In this chapter, instead of looking at mathematical essentials of objects as abstract entities in space, you're going to look at how an object is made to come to life on a monitor. At the heart of this matter is lighting, for to create the illusion of solidity, you need to understand the nature of light and how it can be simulated in real time.

Light

Before a 3-D scene can be drawn to the screen, you must know the position of each polygon that makes up the objects in the scene and what colors to use when drawing the polygons. Colors are made possible by lighting. Color is simply a use of light in the context of the monitor. To understand how this is so, in the sections that follow you take a quick look at how lighting works and how it is used to create complex color effects.

Real Lights

When atoms gain and then lose energy, they emit the energy in the form of a wave of oscillating electric and magnetic fields. These fields are referred to collectively as *electromagnetism*. Depending on the amount of energy, electromagnetism has varying frequencies and wavelengths. Human bodies include detectors that respond to a certain ranges of these frequencies. In the form perceived by humans, this is called *light*. In precise terms, it is called *visible light*.

In the human eye, detectors of light are called *rods* and *cones*. These have different functions. The rod is sensitive to levels of brightness, and levels of brightness are created by differing wave amplitudes. Cones are sensitive to the frequency, and for this reason, they are a bit more complicated than rods. To start with, humans have three different kinds of cone, and each type of cone is sensitive to a different range of light. While one range is described as red, the other two are described as green and blue.

The fact that humans have three types of cones distinguishes them from other animals. Most mammals have two kinds of cone, limiting the colors that they see. At the same time, differences of cones do not mean only that limitations occur. Some animals can see ranges of light that humans consider invisible. Bees see ultra-violet light, and ultra-violet light has a higher frequency light than humans can see. Although not visually, snakes can detect light in the infra-red region, and infra-red light is of a lower frequency, again not naturally visible to humans.

Even though humans have cones that detect red, green, and blue, it remains that they are able to see far more than only three colors. Since most light contains several overlapping waves with a broad range of frequencies, all three kinds of cone are activated to different levels, as are the rods, and this mixture of frequencies is *discriminated* (or experienced) as a single color. Even a single-wavelength beam that doesn't precisely trigger one cone can be discriminated by the amount to which it activates the neighboring cones.

Colors emerge from differences of discrimination. Wavelengths halfway between pure red and pure green are experienced as yellow. Wavelengths halfway between green and blue are experienced as a sky-blue color called cyan. A mixture of blue and red is experienced as a purplish color called magenta. Generally, your visual system processes colors in a cycle characterized by red-yellow-green-cyan-blue-magenta-red. This cycle itself is a product of biological evolution and bears no direct relationship to the underlying wavelengths of light. Along the same lines, a mixture of lots of different wavelengths is experienced as white, while no light at all is experienced as black.

Computer engineers take advantage of the peculiarities of your visual system by mimicking them in the way computer monitors display colors. Each pixel of the monitor screen is made up of three separate emitters of red, green, and blue (RGB). Each emitter can take any value between off and on. For a high-resolution display, each can have a value between 0 and 255. When all three emitters are fully lit, a white dot is created. When none of the emitters is lit, a black dot appears. Depending on the power of your computer and the resolution of your monitor, you can create many different colors this way. In fact, on a *truecolor* scale, you can create 16,777,216 different colors. The true-color scale is a convention sustained by the electronics industry and designates colors that can be defined using 256 shades of red, green, and blue.

In the context of linear algebra, you can represent each color by a 3-D RGB vector. The vector defines the size of each color by a real number between 0 and 1. The color described by $\langle 0 \ 0.5 \ 0.5 \rangle$ is a mid-intensity cyan. The advantage of this, as you'll see shortly, is that it allows you to perform arithmetic with colors.

It's worth going into all this detail about color because many people think color is just "the wavelength of light." Thinking of it this way misses an important issue. Color actually results from micro judgments by the brain based on all the wavelengths the eye receives. It is also affected by environmental influences. If you are in an environment with an "ambient" light that has a blue tinge, or with strong shadows, you perceptually subtract these global values to see the underlying color. A number of optical illusions make use of this phenomenon.

You see objects in the world because light, from the sun or elsewhere, bounces off surfaces and reflects into your eyes. Each surface reacts to light differently. A mirror reflects light exactly as it comes in, like an elastic collision. This is called *specular reflection*. A white ball absorbs the light and then emits it again in all directions, in the process losing detail. This is called *diffuse* or *Lambertian reflection*.

The story goes on. A black piece of charcoal absorbs nearly all of the light but doesn't emit it again as radiation. Instead, it heats up and loses the heat to the air. A red surface is partway between a mirror and a piece of black charcoal. It absorbs most of the light, but releases some of it back in a mixture of wavelengths you experience as red. And a red pool ball has two surfaces, a specular glaze, which reflects some of the light unchanged, and beneath it a diffuse surface that absorbs and emits it, creating a surface with some mirrored qualities and some red.

To model all these factors in a real-time engine takes enormous computing power. Even if you lack the computing power to strive for full emulation, however, you can use certain tricks to fake it. One starting place involves ideal lights.

Fake Lights

The expression *fake light* applies to effects applied to objects rendered in a 3-D world. However, the effects corresponding to fake light represent real-world phenomena. In this respect, then, you have *ambient*, *directional*, *attenuated* light. The light striking any given point on a 3-D object is a combination of three effects. Figure 20.1 shows an object illuminated by different kinds of light.

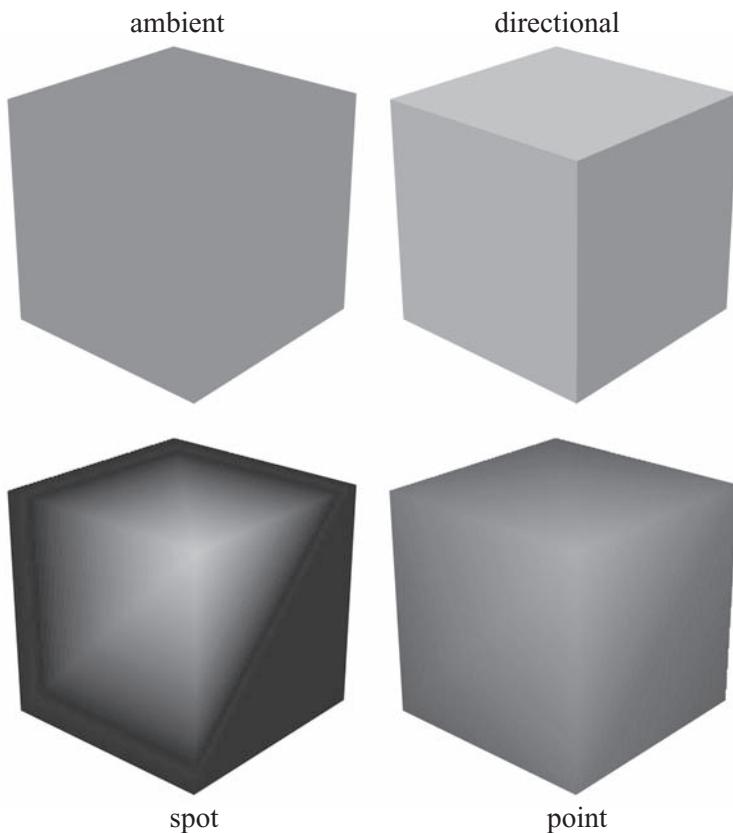


Figure 20.1
Four different lights.

Ambient light simulates the light that surrounds you. One form of ambient light is light that has come in from a window and bounced around the room several times, until it has no real direction. It illuminates everything in the scene. In the context of a graphical simulation, since it bounces around in this way, ambient light is easy to calculate. It acts equally on all polygons in all directions. In some cases, adjustments must be made, however. For example, complex models allow the color and brightness of ambient light to vary through space.

Directional light, as the name suggests, comes from a particular direction, like the light of the sun, but it is not affected by position. It equally illuminates all objects in the scene. A scene can have any number of directional lights. For convenience, these are placed in the scene as if they were ordinary nodes, but only their rotation vector is relevant to their effect.

Attenuated light has two forms. One form is *spotlight*. The other form is *point light*. These forms of light are created by objects that can be placed in a scene to illuminate the objects around them with a particular color. Objects nearer to an attenuated light are illuminated more than those farther away. How much this happens depends on three attenuation constants that modify the brightness of the beam by a factor b . To examine how this is so, consider that for a point light you have the following equation:

$$b = \frac{1}{k_1 + k_2 d + k_3 d^2}$$

On the other hand, for a spotlight pointing in the direction \mathbf{u} and at a unit vector \mathbf{v} with distance d from the surface, you have this equation:

$$b = \frac{\max(-(\mathbf{u} \cdot \mathbf{v}), 0)^p}{k_1 + k_2 d + k_3 d^2}$$

where p is a special constant that measures the spread or focus of the light. A high value of p means that the light mostly illuminates a very narrow beam. A low value makes it spread out more widely. An alternative method is to specify an actual angle for the beam. You use the term $\max(-(\mathbf{u} \cdot \mathbf{v}), 0)$ to ensure that only surfaces whose normals point toward the light are illuminated.

Materials

Each form of fake light strikes the surfaces of objects in a scene and reflects off them to make them visible. How they do this is determined by the quality of the surface. The surface is defined by means of an object called *material*. You can think of material as a coating applied to an object.

The Color Elements of a Surface

A material describes all the different qualities of the surface and includes a number of different color components. The values for these qualities can be represented by either single values or image maps. Image maps are a topic covered in detail in the next section. For now, think of them as single values applied across the whole surface.

The simplest color element is called the *emissive color*. The emissive color is a color actually given off by the object, such as a glowing lamp. Emissive light is a cheat, however. Unlike real light, it has no effect on any other object. Instead, uniformly in all directions, it changes the color of the object emitting it. While compromising the realism of the scene, emissive light is computationally cheap and provides a simple way to create objects of different colors. You'll designate this color with the expression C_{em} .

The next color element is *diffuse color*. Diffuse color tells you the color of light that would have a Lambertian reflectance if the surface were illuminated with full-spectrum white light. In other words, such light would be the best candidate for the color of the surface. The diffuse color does not depend on the position of the observer, but it does vary according to the angle of the light falling on the surface. As shown in Figure 20.2, since the center surface is illuminated by more of the beam, the closer the light angle is to the surface normal, the more of it is reflected.

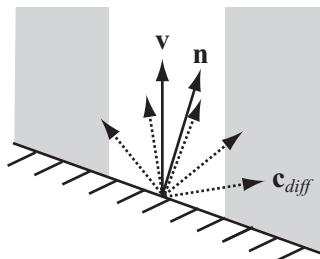


Figure 20.2

Calculating the diffuse component.

The characteristics of surface illumination allow you to develop a formula for the diffuse color component from a surface with normal \mathbf{n} due to a particular light. If the diffuse color of the material is \mathbf{d} , and it's illuminated by a light of color \mathbf{c} from the (unit) direction \mathbf{v} , then the Lambertian reflection c_{diff} is given by $\mathbf{cd} \max(\mathbf{u} \cdot \mathbf{v}, 0)$ where the multiplication of the colors is performed pairwise.

When considering the formula for Lambertian reflection, remember that color vectors are not the same as linear vectors in space. Among other things, pairwise multiplication modulates one color with another, making a blue vector more red. However, this is the correct method when dealing with a surface that absorbs some frequencies and reflects others. Note that since ambient light is normal to all surfaces, the diffuse component of an ambient light is just \mathbf{cd} .

The specular component has two elements, a color \mathbf{s} and an exponent m . These combine to create a light of a single color, called a *specular highlight*. What they don't create is a mirror reflection. To create a mirror reflection, you must model not just the direct light on the object due to the various lights in the scene but the light reflected off all the other objects. When you pursue this objective, you enter into a raytracing territory that is computationally expensive. Given this limitation, the illumination of an object in a real-time 3-D scene is usually affected only by the lights themselves, not by light emitted, absorbed, or reflected by other objects. This outcome affects not only the possibility of mirror-images but of real-time shadows. Both of these have to be modeled in different and not entirely satisfactory ways.

As discussed previously, a specular reflection is the light equivalent of an elastic collision. As shown in Figure 20.3, the result of this effect is that light bouncing off a surface is emitted at the same angle as it strikes the surface. The nearer the viewing vector is to this (unit) reflection vector \mathbf{r} , the brighter the specular light becomes. You modify how near the vector needs to be by means of the exponent m , which focuses the reflection in much the same way as the exponent p focuses spotlights.

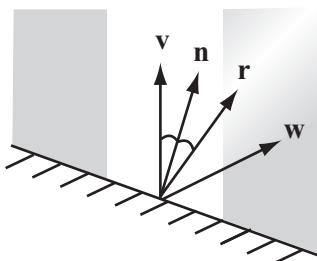


Figure 20.3
Calculating the specular component.

You can calculate the specular reflection due to a particular (non-ambient) light using different formulations. One in particular works especially well. With this approach, you assume that $\mathbf{n} \cdot \mathbf{v} > 0$ and that the observer is at a vector \mathbf{w} from the point on the surface. Given this understanding, you have $\mathbf{c}_{spec} = sc (\max(\mathbf{r} \cdot \mathbf{w}, 0))^m$. A specular color of white is usually the most appropriate. The exponent m can take any value. A value of 0 gives you a diffuse color, and an infinitely high value gives what would in theory be a mirrored surface. In this case, only a viewing angle exactly along \mathbf{r} will detect the light.

Each of these values needs to be calculated for each non-ambient light applied to each surface. The sum of the values due to all of the lights in the scene is the color seen by the viewer. To generate this color, you add the colors with a maximum of 1 for each primary color. You use this approach because you are combining the effects of several light sources. One color is not used to modulate another. You can sum all this up with the `surfaceColor()` function, which takes values corresponding to the light effects discussed in this and previous sections. This function is supplemented by others.

```

function surfaceColor(normal, position,
                      material, lights, observerPosition)
    set color to emissiveColor of material
    set observerVector to observerPosition - position

    repeat for each light in lights
        set lightColor to illumination(position, light)
        if light is not ambient then
            set v to the direction of light
            set diffuseAngle to max(-dotProd(normal, v),0)
            if diffuseAngle>0 then
                set diffuseComponent to
                    modulate(diffuseColor of material,
                            lightColor)
                add diffuseComponent*diffuseAngle to color

            set specularReflection to v + 2* dotProd(v, normal)
            set specularAngle to
                max(dotProd(observerVector,
                           specularReflection), 0)
            set brightness to power(specularAngle,
                                   specularFocus of material)

            set specularComponent to
                modulate(specularColor of material, lightColor)
            add specularComponent*brightness to color

```

```
    end if
otherwise
    add lightColor*diffuseColor of material to color
end if
end repeat
end function
```

The `modulate()` function attends to adjusting of colors relative to vector values:

```
function modulate(color1, color 2)
    return rgb(color1[1]*color2[1],
               color1[2]*color2[2], color1[3]*color2[3])
end function
```

The `illumination()` function takes parameters that define the source position of the light and the light itself.

```
function illumination(position, light)
    set color to the color of light
    if light is spot then
        set v to the position of light - position
        set brightnessAngle to max(-dotProd(v, direction of light), 0)
        if brightnessAngle=0 then return rgb(0,0,0)
        set brightness to power(brightnessAngle, angle factor of light)
        multiply color by brightness
    end if

    if light is spot or point then
        set d to mag(v)
        set denominator to the constant factor of light
        add the linear factor of light * d to denominator
        add the quadratic factor of light * d * d to denominator
        divide color by denominator
    end if
    return color
end function
```

There are additional components to materials, all of which are described by some kind of image map, which you'll look at next.

Image Maps and Textures

Not all objects are a solid color. Most have some kind of detail. Detail gives a pattern or texture to objects. In order to create details, you use an image called a *map*. A map gives information to the 3-D API about the surface of the shape at a higher resolution than just polygon by polygon. How maps are created and projected onto the surface is discussed farther on, but for now you only need to know that they allow you to specify values for various parameters of the surface at a pixel-by-pixel level. Further, because the pixels of the image map are converted to “texels” when they are applied to the surface, you work at a texel-by-texel level. In this way, maps can be applied to different surface areas depending on how they are projected.

Some examples of image maps are as follows:

- **Texture.** Texture maps are sometimes called textures. They modify the diffuse component of the surface.
- **Gloss.** Gloss maps modify the specular component.
- **Emission.** Emission maps modify the emission component.
- **Light.** Light maps modify the texture map and thus the diffuse component.
- **Reflection.** Reflection maps create a reflected image over the top of the main texture.
- **Bump and normal.** Bump and normal maps create the illusion of convoluted surfaces.

Note

Reflection maps are actually the same as texture maps; they are just applied differently to the surface.

Texture, gloss, and emission maps are fairly simple to understand. Each of them is an image that gives the value of the appropriate color of a material at particular points. By specifying how the image is mapped to the surface, you can alter the end result. The process is similar to clothing a paper doll. Textures can be combined to create more complex effects in essentially the same way that they can be combined to create a complex 2-D image in a program like Adobe Photoshop. This is in addition to global diffuse, specular, and emissive components applied to the whole surface.

Calculating all the lights in a scene is the most processor-hungry part of the operation. Most of the time, you’re recalculating exactly the same values every time. And yet the lights in a scene are basically static. Given this situation, it is important to pre-calculate the lighting in the scene and save it into the texture file. This is known as *baking*. Baking allows you to decrease the number of real-time lights. At the same time, it also requires a significant increase in the amount of texture information.

To get around this double bind, you can create a second map, *a light map*. A light map defines the lighting levels for each part of the scene, usually at much lower resolution than the texture map. By using the light map to modulate the texture map, you can reuse textures across an entire scene while still gaining most of the processing advantage of an image map. Figure 20.4 shows how this works. The texture map in the first picture has been combined with the lower resolution light map in the second picture to create an image with a shadow. Most 3-D modeling software includes the option to create baked textures, both as light maps and as new complete texture maps.

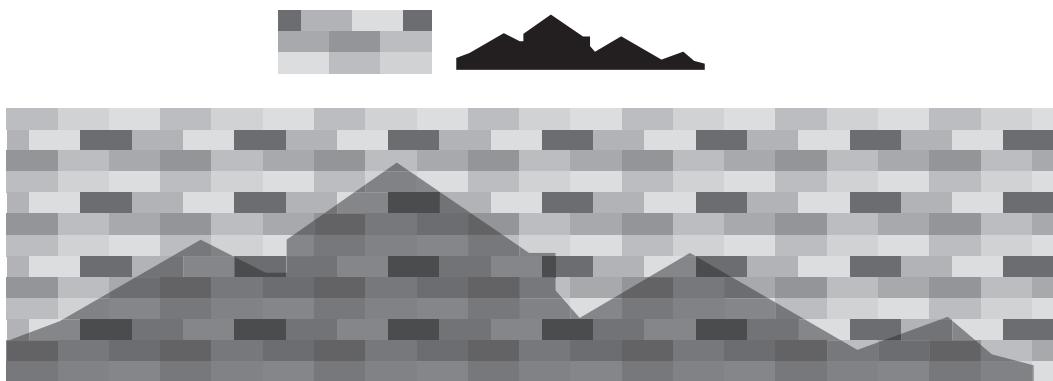


Figure 20.4

Using a light map to modulate a texture map.

A *bump map* is a way to model variations in height at a level of detail smaller than the polygon. Examples of its use include creating pockmarks, blisters, and embossed text. Such effects are created by using shadows and highlights. The bumps are faked. The process is similar to drawing a *trompe l'oeil* 3-D image on a piece of paper. When the surface is viewed head-on, it's very convincing, but you can see it's not really 3-D when your eye is almost level with the paper.

Bump maps are essentially height maps, where each point on the map represents a distance from the surface. Usually a grayscale image is employed to create the effect. An alternative is the *normal map*. With a normal map, by mapping the x -, y -, and z -coordinates of the (unit) vector to red, green, and blue values, each texel encodes a normal direction as a color. Ultimately, the bump map is equivalent to the normal map. The tradeoff is that with the normal map an increased amount of memory is used. With the bump map, there is a gain in performance. As shown in Figure 20.5, normal area maps are used to perturb the normal of the surface, changing the results of the directional and specular components of any lights.

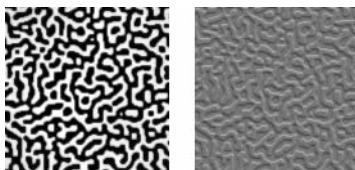


Figure 20.5

A bump map and its effect on the lighting of a surface.

A normal map can be derived from a bump map. The way to do this involves how the height maps are read. With a height map, essentially, you compare the heights of pixels in small neighborhoods of the bump map.

Fitting a Map to a Shape

To use your image maps correctly, you must tell the 3-D engine which part of the image corresponds to which part of the surface. To do this, you have to create a mapping from the image to the surface. In other words, for each point on the surface, you must match the point to a texel in the image map. For the sake of discussion, you can refer to this as a texture.

To match points and texels, you start by labeling your texels using standard coordinates. Textures are usually 2-D images, although for materials such as grainy wood, many people prefer to use 3-D texels, which map to the whole volume of an object and create more interesting effects. You label the texels using a separate coordinate system for clarity, such as s and t or u and v .

To map the texels to the surface, you can use a transform. You do so in a limited way, however, essentially restricting it to 2-D. However, when you do this so that you can perform scaling operations, you keep homogeneous coordinates intact. This allows you to rotate, translate, and scale the map before attaching it to each polygon. As for the mapping itself, you must still work out what to use. Some standard approaches are as follows:

- **Planar.** The texture is applied as if it passes the whole way through the object and emerges on the other side.
- **Cylindrical.** The texture is wrapped around the object like a roll of paper.
- **Spherical.** The texture is scrunched onto the object like a sphere.
- **Cubical.** Textures are combined to form a cube that is mapped to the outside. With this approach, six textures are used.
- **Bespoke.** For some complex meshes like characters, you must individually specify the texture coordinates for each triangle in the mesh.

Each of these options, apart from the last, offers a way to translate the 3-D information about the points on the surface of the object into 2-D form. With a planar map, you project the 3-D coordinates of each vertex to a plane whose xy -coordinates are used to map to the st -coordinates of the texture map. As Figure 20.6 illustrates, the x - and y -coordinates of the object are mapped directly to the s - and t -coordinates of the image.

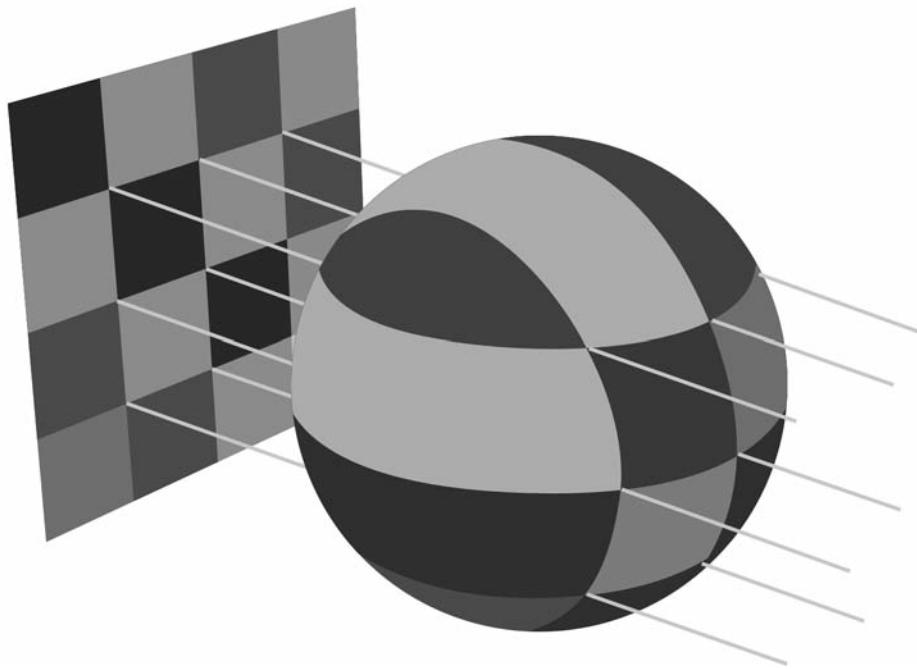


Figure 20.6
Planar mapping.

With a cylindrical map, as illustrated by Figure 20.7, you start by choosing an axis for the object. Then you calculate the distance of each vertex along the axis. This action scales the vertices to your t -coordinates. For the s -coordinate, you use the angle the vertex makes around the axis.

With a spherical map, you might use the latitude and longitude of each point as projected to a sphere. Using this approach, you discard the distance from the center. This approach is shown in Figure 20.8.

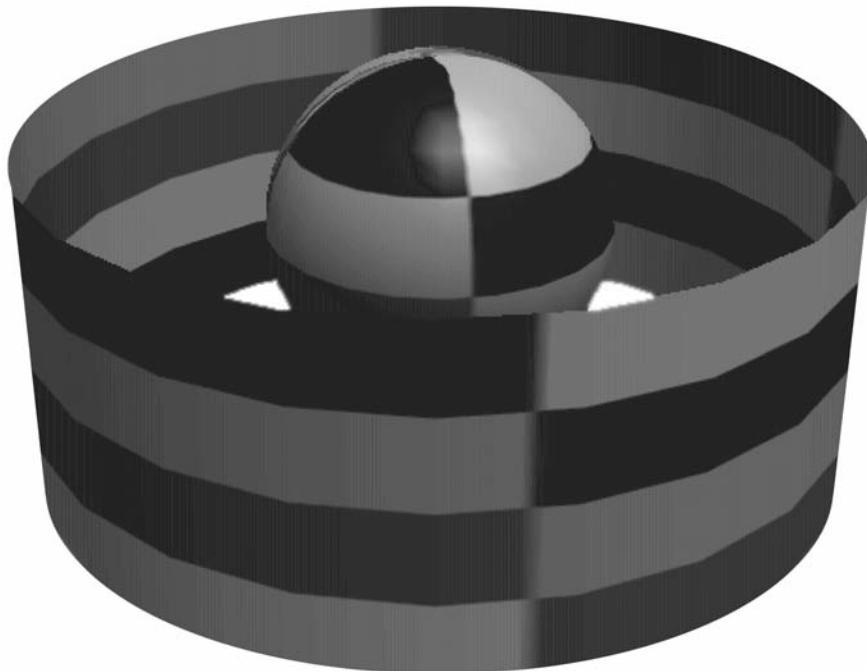


Figure 20.7
Cylindrical mapping.

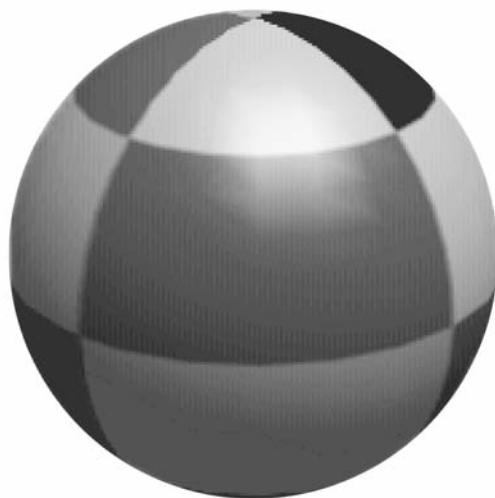


Figure 20.8
Spherical mapping.

In all the cases mentioned in this section, the positions of the vertices are directly related to the texture coordinates. There are different values for these positions. You can use the object's local geometry, or you can use its position in the world. The approach you use creates different effects. If you use the local geometry, then the texture remains the same regardless of how the model is transformed. You can rotate it, scale it, and so on, but the texture always looks the same. But if you use the world basis, the texture changes depending on how the object is placed. In particular, if you spin the object on its axis, the texture remains where it is, simulating the effect of a reflective surface. If you make a texture that represents the reflection, it will always be oriented the same way.

Reflection maps don't always have to be used for reflections. A light map applied the same way allows your object to simulate the effect of constant shadows, like an apple turning under the dappled sunlight of a tree. You can even apply it to stranger objects, such as bump maps, but if you do this, the effect is likely to be odd. For example, lumps are likely to move around under the surface, creating an effect similar to the flesh-eating scarabs from the movie *The Mummy* (1999).

Mip-Maps

One problem with image maps is that the image to which they are applied may be near to or far from the camera. If the image is far away, then you're using much more information than you need about the surface. If each pixel on the screen covers a hundred different texels, then you don't really need to know the color of each one. In fact, having more information than you need can do more harm than good. Conversely, if the image is close, then a single texel might cover a large amount of screen space, leading to *aliasing*, or jagged edges between texels.

You can deal with the problem of aliasing first. When working with an object that is close to the camera, you don't normally want to draw each texel as a solid plane of color. Instead, you want to interpolate smoothly from one to another. You can accomplish this using *bilinear filtering*, illustrated in Figure 20.9. Here, for each pixel, you determine the four nearest texels to a particular screen pixel and create a weighted average of all the colors at that point. In Figure 20.9, the two faces of the cube have the same 4×4 pixel texture. The one on the left has bilinear filtering turned on.

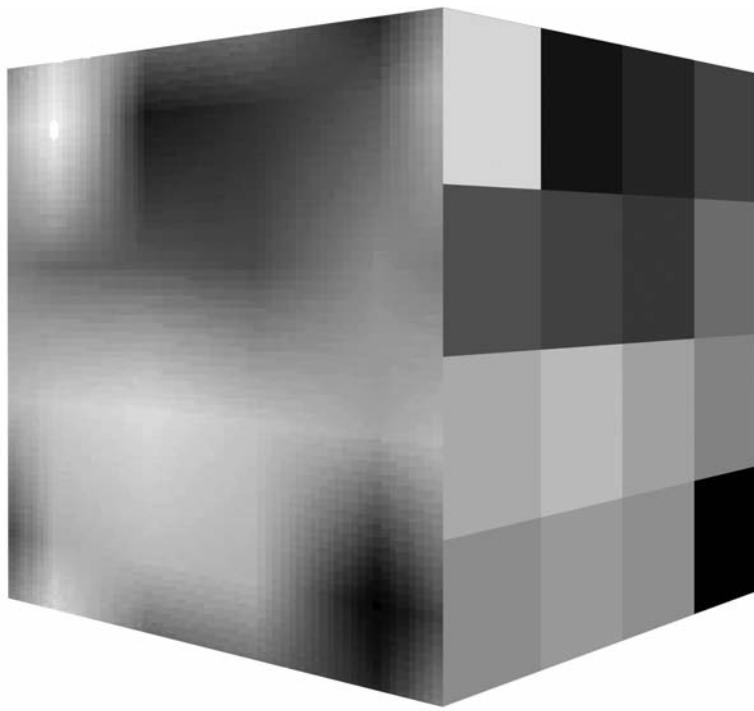


Figure 20.9
Bilinear filtering.

As you can see in Figure 20.9, bilinear filtering is not without its problems. It can blur the texture. One alternative technique is *oversampling*. With oversampling, instead of finding a single texel point under a pixel and then blurring with nearby texels, you find the texel points under a number of nearby pixels and blur them. The result is that near to the camera, you don't see serious blurring. Instead, you see strongly delineated areas of color with nicely antialiased lines between them.

For textures at a distance, one solution is to use a *mip-map*, which is a set of pre-calculated textures at different levels of detail. With a mip-map, you might have a texture that is 256×256 texels in size but also stored in lower-resolution versions of 128×128 texels, 64×64 texels, and so on, down to 1×1 texels. The last version is the equivalent of the average color of the whole texture. The 3-D engine can then choose which of these maps to use depending on the amount of screen space a particular polygon takes up. The amount of memory used by the mip-mapped texture is higher than before, but not much higher. It is usually less than 50% more. The gains in both processing speed and image quality more than make up for it.

Note

The term *mip* is an unusually intellectual piece of technical terminology. It stands for the Latin phrase “multim in parvo,” or “many in a small space.”

Having created your mip-maps, you still have a number of issues to deal with. For example, what happens at the transition point between different maps? If a large plane is being viewed, the nearest edge of it will be seen with the highest-quality texture map. The farther edge will be using the lowest quality. Between, there will be places where the engine switches from one map to the other, and this might be noticeable as a sudden increase in image quality. To avoid this irregularity, you can interpolate from one map to the other using *trilinear filtering*.

Trilinear filtering involves combining the effects of different resolutions at the boundaries. As illustrated by Figure 20.10, with such filtering, you calculate the color due to both nearby mip-maps. You then use a weighted average of the two to determine the appropriate color, smoothing the transition.

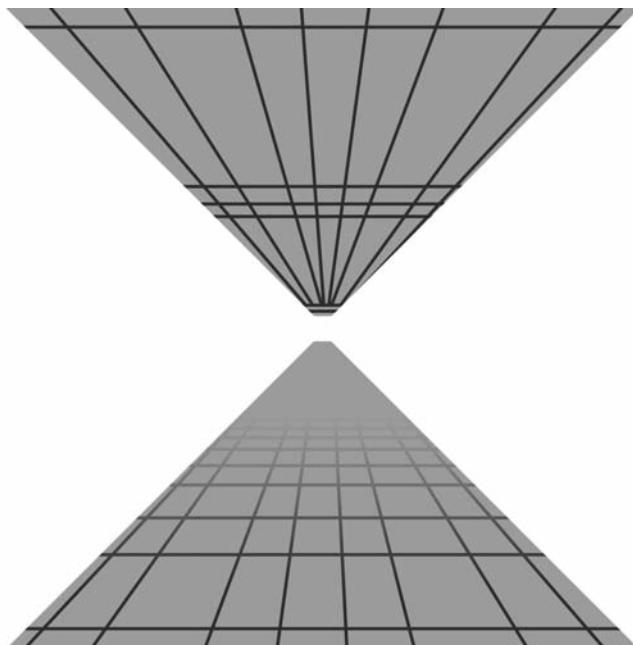


Figure 20.10

Using a mip-map, with and without trilinear filtering.

Some graphics cards use *bilinear filtering*. This form of filtering applies trilinear filtering selectively, usually near transition points. Some cards also use a technique called *anisotropic filtering*, which instead of merging square combinations of pixels, combines regions of pixels that are related to the angle of view. The result is that if the surface being viewed is heavily slanted away from the viewer, a more elongated section of it is used to create the combined pixel color. While more demanding of the processor, this approach produces a very realistic effect.

Shading

Another way to create detailed subpolygon contours on a model is to use *shading*. Shading is a way to interpolate the surface color according to the surrounding faces to create a smooth object.

Gouraud and Phong Shading

The simplest shading method is called *Gouraud shading*. This method of shading works by calculating the correct color at each of the three vertices of a triangle. It then interpolates them across the triangle. Gouraud shading affects only the constant components of a material. It is not affected by image maps.

The interpolation is achieved using *barycentric coordinates*. Barycentric coordinates are similar to homogeneous coordinates. As illustrated by the image on the left of Figure 20.11, the barycentric coordinates (w_1, w_2, w_3) of a point P in a triangle can be defined as a set of weights that can be placed at the vertices of the triangle to establish the center of gravity at P. Equivalently, they can be used to balance the triangle on a pin placed at P.

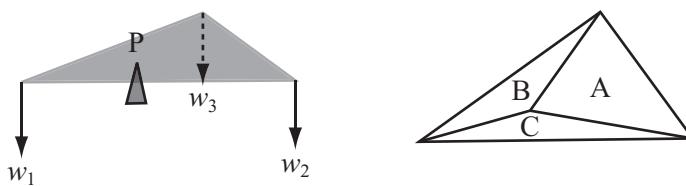


Figure 20.11

Barycentric coordinates.

If the first description of the use of barycentric coordinates in Figure 20.11 seems somewhat obscure, consider a second approach. With the second approach, picture the coordinates in terms of the three areas A, B, and C in the image on the right of Figure 20.11. You find that choosing $w_1 = A$, $w_2 = B$, and $w_3 = C$ gives a solution to the problem. As with homogeneous coordinates, there is only one possible solution, since barycentric coordinates are invariant under scaling. As discussed in Chapter 17, since the area of a triangle is half the magnitude of the cross product of two of its sides, this gives you a simple function to calculate the barycentric coordinates of a point. The `barycentric()` function encapsulates the logic and mathematics of this approach:

```
function barycentric(p, v1, v2, v3)
    set t1 to v1-p
    set t2 to v2-p
    set t3 to v3-p
    set a1 to t1[1]*t2[2]-t1[2]*t2[1]
    set a2 to t2[1]*t3[2]-t2[2]*t3[1]
    set a3 to t3[1]*t1[2]-t3[2]*t1[1]
    return norm(vector(a1, a2, a3))
end
```

You can use barycentric coordinates (scaled to unit length) to interpolate colors. For each point of a triangle, you multiply each vertex color by its appropriate weight and add them together. The `colorAtPoint()` function accomplishes this task:

```
function colorAtPoint(pos, vertex1, vertex2,
                      vertex3, color1, color2, color3)
    set coords to barycentric(pos, vertex1, vertex2, vertex3)
    return color1*coords[1] + color2*coords[2] + color3*coords[3]
end function
```

In graphical applications, this process is made much more efficient when you employ optimizations to allow integer calculations to be used. As an added benefit, when you know the barycentric coordinates of a point, if all three coordinates are between 0 and 1, you can tell that it's inside the triangle.

If you have a fast graphics card, you can do additional work and create a kind of global bump map for the triangle. Instead of calculating the colors and interpolating them, you interpolate the normals of the triangle and use these for pixel-by-pixel lighting calculations. This is called *Phong shading*. Since it is a great deal more difficult for the processor to use this approach, to save time, the 3-D engine usually interpolates, calculating the light intensity due to each light on a vertex-by-vertex basis and interpolating this across the triangle when calculating the contribution of each light.

The Normal at a Vertex

The shading methods described in the previous section rely on calculating the normal of the smoothed surface at each vertex. This is a problem due to the processing load required. To solve this problem, consider that the normal at each vertex can be calculated in two ways. The simplest is just to calculate the mean of the normals of all triangles that share that vertex. This works well for fairly regular shapes.

A more advanced method is to weight the normal according to the size of each triangle so that more prominent triangles have a greater influence. You can do this fairly simply by using the cross product as you did in calculating barycentric coordinates.

As you've seen before, you find the normal to a triangle whose vertices are ordered anti-clockwise as you look down on the triangle (conventionally) as $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ by finding the normalized cross product of $\mathbf{v}_2 - \mathbf{v}_1$ and $\mathbf{v}_3 - \mathbf{v}_1$. If instead you take the non-normalized cross product, you end up with a normal whose length is twice the area of the triangle. Averaging out these vectors before normalizing gives a weighted sum as required.

Note

If the model is not smooth, then there is no single answer to "what is the normal at this vertex?" There can be three or more of them.

Exercise

EXERCISE 20.1

Create a function that applies a cylindrical, spherical, or planar texture map to a surface. If you have a 3-D engine, you can try the results of this function. Even without a 3-D engine, you should be able to calculate the *st*-coordinates for any vertex in the mesh. You'll find the most difficult part of this problem involves dealing with the "singularities" where the texture meets itself. One such singularity is the top of the sphere.

Summary

In this chapter, you've taken a fairly detailed look at lighting, textures, and shading. You've learned how lighting in a 3-D simulation relates to real-world light, and how surfaces can react in different ways to the light that falls on them. You've also seen how to use materials, how to make image maps, and how to project them to a surface. Finally, you've examined shading and how it can be used to create the illusion of a smooth shape. In Chapter 21, the last chapter in Part IV of this book, you're going to take a look at some 3-D modeling techniques, including how to create surfaces from level maps and how to model water waves.

You Should Now Know

- How you use the *visible light* spectrum to see objects
- The different ways that objects can react to light
- How light is modeled in the computer
- The meanings of *ambient*, *diffuse*, *directional*, and *attenuated* as they apply to lights
- The meanings of *diffuse*, *specular*, and *emissive* as they apply to surfaces
- How to create an image map
- How to project an *image map* to create textured surfaces, shadows, and reflections
- How to use a *mip-map*, *bilinear*, and *trilinear* filtering and oversampling to remove aliasing effects
- How colors are interpolated across a triangle to create a smooth surface

This page intentionally left blank

CHAPTER 21

MODELING TECHNIQUES



In This Chapter

- Overview
- Mathematical 3-D Modeling
- Animated Surfaces
- Bone Animations

Overview

This is the final chapter on 3-D graphics. It examines techniques for creating complex objects at the mesh level. So far, you've been looking at how to move objects from place to place without considering how they are made. You have assumed that the objects are either primitives, such as spheres or boxes, or pre-existing polygonal meshes. At this point, you will examine underlying surfaces and how they can be defined. You're also going to explore surfaces, such as water and cloth, that can be animated in real time, and you'll finish up with a brief look at creating animated characters.

The discussion includes attention to inverse kinematics, a technique for animation that has become popular due to its use in different animation tools. Since you will be examining the mathematics behind the technology, most of the topics in this chapter are advanced. It is important to remember this. While you will gain familiarity with mathematical and technical topics discussed, whole books have been written to treat the topics introduced. In this respect, it is hoped that this chapter will leave you with enough knowledge to be curious about further research.

Mathematical 3-D Modeling

A good beginning for a discussion of 3-D modeling is *static* modeling. Static modeling concerns how you can build a realistic-looking surface from simple parts.

Surfaces of Rotation

Among other things, you can employ *lathe technique* to create a surface. The lathe technique involves creating a surface of rotation. To create a *surface of rotation*, you begin by defining a function f in one variable, usually one that does not have any roots in a particular interval. You then define your surface as the set of 3-D points whose distance from the x -axis at a particular value of x is $f(x)$ over some range of x -values. As illustrated by Figure 21.1, given the use of a particular cubic function, the resulting surface is shaped like a vase. The plot of the path is made on either side of an axis, and by revolving the function plot around the axis, a 3-D shape is generated. This, then, is a surface of revolution.

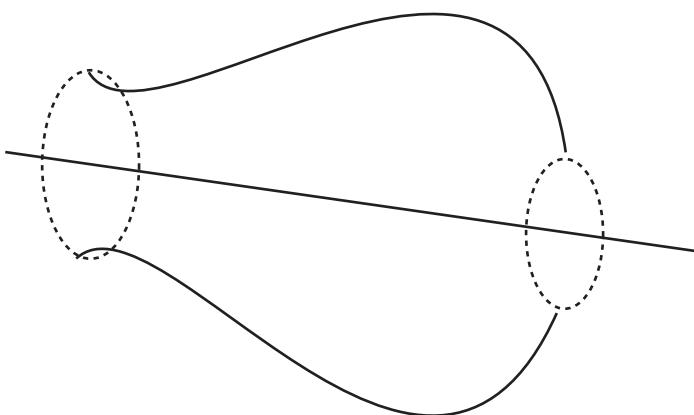


Figure 21.1

Creating a vase as a surface of revolution.

A surprising number of useful shapes can be created as surfaces of revolution. You can generate anything with complete rotational symmetry along one axis. This implies anything can be modeled on a potter's wheel. The big advantage of this approach is that many physical elements become reasonably easy to deal with. Among other things, the moment of inertia of a surface of revolution about its axis of symmetry is proportional to the integral of the function used for the revolution, $\int f(x)^2 dx$.

Similarly, collision detection with a surface of revolution, while not simple, is somewhat simplified. You can consider the surface to be a succession of frusta of cones, and you're interested only in collisions along the main surface, not with the top or bottom, which are the cause of most complications.

Splines in 3-D

To create more complex surfaces, your best bet is to extend the concept of a spline into three dimensions, creating a spline surface. It will be necessary to review several topics before getting the whole story on how this is accomplished, for the spline surface of choice in 3-D is the B-spline, or NURBS (which stands for *non-uniform rational B-spline*), and this proves to be a fairly complex mathematical topic. However, with reference to topics previously discussed, one beginning is to consider how you might extend the concept of a Bezier or Catmull-Rom spline into 3-D.

The simplest application is to create a single curve in space, giving it some depth by defining a normal at each control point. As shown in Figure 21.2, the result is something like a 3-D track curving and twisting through space. Here, the main curve follows a standard Catmull-Rom spline, which makes each control point a 3-D vector. However, a second "virtual" Catmull-Rom spline in one dimension defines a curve of angles from the vertical. To accomplish this you associate an angle with each control point and then create a normal vector at each point that is both perpendicular to the curve and at the correct angle to the vertical. The result is a smooth track in 3-D.

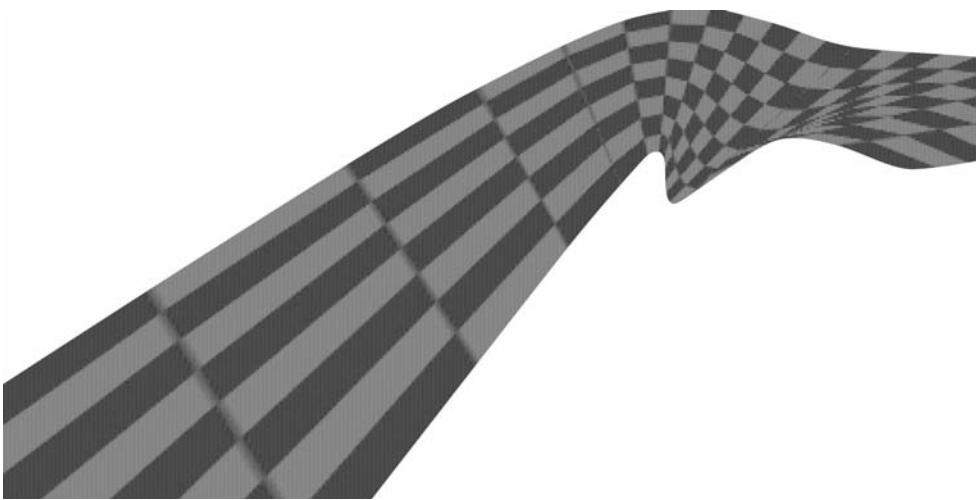


Figure 21.2

Using a spline to create a curved track.

Note

You'll return to this example in Chapter 23, when you look at tiled 3-D splines.

The problem with the method depicted in Figure 21.2 is that it doesn't enable you to create a complete surface. You end up with a surface that merely follows a specific line. However, you can overcome this limitation by creating a grid of curves instead. Toward this end, you define $n \times m$ control points and then use splines to interpolate a grid of n splines in one direction and m in the other. While it works fine for the purposes set, using this approach requires quite a lot of calculation, and the result offers no real integration of the two directions into one.

NURBS

While the approaches discussed in the previous sections offer excellent beginning strategies for dealing with surfaces, in industrial contexts, surfaces are dealt with using different, more advanced approaches. The most common utility for drawing surfaces in 3-D packages is NURBS. Encompassing both Bezier and Catmull-Rom curves, B-splines are an extremely versatile form of spline. They can readily represent circles, ellipses, spheres, and tori, among other familiar shapes.

Note

A *torus* is a doughnut shape. The plural of *torus* is *tori*.

Unlike the splines you've looked at up to now, B-splines are not defined segment by segment as individual cubic curves. Instead, they are created by a single knot vector defined using a set of values $\{t_0, t_1, \dots, t_m\}$ such that for each $i < m$, $0 \leq t_i \leq t_{i+1} \leq 1$. This creates a set of points on the curve called *knots*, each of which is the result of setting the parameter t to one of the knot vector values.

Because different kinds of knot vectors create curves with particular kinds of behavior, the knot vector is the principal way to classify B-spline curves. When the knots are evenly spaced, the curve is called *uniform*. This is the source of the “non-uniform” (NU) in NURBS. However, in addition to knots, the physical shape of the curve is created by a number of control points, $\{P_0, P_1, \dots, P_n\}$, where $n \leq m$. You define the degree of your spline to be equal to $k = m - n - 1$.

One additional element is needed. This is a set of functions called the *basis* or *blending functions*. They are defined by a recursive process.

First, for $p < j \leq k$, with $\frac{0}{0}$ defined as 0, you have the following equation:

$$N_{i,j}(t) = \frac{t - t_i}{t_{i+j} - t_i} N_{i,j-1}(t) + \frac{t_{i+j+1} - t}{t_{i+j+1} - t_{i+1}} N_{i+1,j-1}(t)$$

Then, for $t_i \leq t \leq t_{i+1}$ (and otherwise 0), you have

$$N_{i,0}(t) = 1$$

Note

Recall the definition *basis* from the world of vectors. As it happens, it's possible to extend the concept of a vector space into more abstract realms, particularly that of functions. Just as with more conventional vectors, a basis in an abstract vector space is a set of elements such that any element of the space can be written uniquely as a linear sum of multiples of the basis elements. As a result, a basis for the space of polynomial functions might be the functions $1, x, x^2, x^3$, and so on. This space has an infinite number of dimensions.

Since these functions are a little hard to visualize, it might help to introduce a simple example. Suppose that you have five knots at 0, 0.2, 0.5, 0.8, and 1 and a curve of degree 2 (so it has two control points) for which you want to find $N_{1,2}$. By the recursive definition, you have

$$N_{1,2}(t) = \frac{t - t_1}{t_3 - t_1} N_{1,1}(t) + \frac{t_4 - t}{t_4 - t_2} N_{2,1}(t)$$

Applying the definition again, you get

$$\begin{aligned} N_{1,2}(t) &= \\ &\frac{t - t_1}{t_3 - t_1} \left\langle \frac{t - t_1}{t_2 - t_1} N_{1,0}(t) + \frac{t_3 - t}{t_3 - t_2} N_{2,0}(t) \right\rangle \\ &+ \frac{t_4 - t}{t_4 - t_2} \left\langle \frac{t - t_2}{t_3 - t_2} N_{2,0}(t) + \frac{t_4 - t}{t_4 - t_3} N_{3,0}(t) \right\rangle \end{aligned}$$

Now you have reduced all the functions to $j = 0$, which means you can now apply the elementary definition, giving you several different behaviors enumerated in the following list:

- If $t < 0.2$, then all of the functions evaluate to 0, and you have $N_{1,2}(t) = 0$.
- If $0.2 \leq t < 0.5$, then you have $N_{1,0}(t) = 1$, so

$$N_{1,2}(t) = \frac{(t-t_1)^2}{(t_3-t_1)(t_2-t_1)} = \left(\frac{(t-0.2)^2}{0.6 \times 0.3} \right)$$

- If $0.5 \leq t < 0.8$, then you have $N_{2,0}(t) = 1$, so

$$N_{1,2}(t) = \frac{(t-t_1)(t_3-t)}{(t_3-t_1)(t_3-t_2)} + \frac{(t_4-t)(t-t_2)}{(t_4-t_2)(t_3-t_2)}$$

- If $0.8 \leq t < 1$, then you have $N_{3,0}(t) = 1$, so

$$N_{1,2}(t) = \frac{(t_4-t)^2}{(t_4-t_2)(t_4-t_3)}$$

Notice that if $j = k$, then the basis functions contain a reference to $k + 1$ of the function $N_{i,0}$. Notice also that the functions are entirely independent of the values of the control points. This is the reason that the knot vector has such a profound impact on the behavior of the curve. Further, although the definitions of the basis functions seem involved, computationally they are quite simple. The `NURBSbasisFunction()` function encapsulates the recursion involved in the computation in relatively few lines of code:

```
function NURBSbasisFunction( i, j, t, knotvector)
    if j=0 then // bottom out recursion
        if t<knotvector[i+1] then return 0
        if t>=knotvector[i+2] then return 0
        return 1
    end if
    // otherwise recurse
    if (knotvector[i+j+1]-knotvector[i+1])=0 then set a to 0
        otherwise set a to (t-knotvector[i+1]) /
            (knotvector[i+j+1]-knotvector[i+1])
    if (knotvector[i+j+2]-knotvector[i+2])=0 then set b to 0
        otherwise set b to (knotvector[i+j+2]-t) /
            (knotvector[i+j+2]-knotvector[i+2])
    return a*NURBSbasisFunction(i,j-1,t,knotvector)
        + b*NURBSbasisFunction(i+1,j-1,t,knotvector)
end function
```

Note

Remember that by the convention of this book, arrays begin with element 1.

Having defined the basis functions, you can at last arrive at the equation of the B-spline curve. It reads as follows:

$$C(t) = \sum_{i=0}^n P_i N_{i,k}(t)$$

While the function is easy to compute, notice that it is more abstract than the splines you've seen before. However, you still have a local behavior. Each curve segment is affected only by the nearby control points. In this case, each basis function, and thus each control point, contributes to exactly $k + 1$ of the curve segments.

A question remains, however. What about the *rational B-spline* (RBS) part of the NURBS acronym? Consider that if you use homogeneous coordinates to define your control points, you can think of the coordinates as a 3-D vector and a scalar w_i , which is the weight of the control point. This can be done for all B-splines by setting w_i to 1 for each control point. Allowing w to vary gives you a little more control. Curves in which w is always 1 are called non-rational. Curves in which w is allowed to vary are called rational. This leaves you with the general formula for a NURBS curve:

$$C(t) = \frac{\sum_{i=0}^n P_i w_i N_{i,k}(t)}{\sum_{i=0}^n w_i N_{i,k}(t)}$$

Note

NURBS is a singular noun, so you can speak of "a NURBS." However, it is usually used as an adjective. You typically say, for example, "a NURBS surface" or "a NURBS curve."

Rational B-splines provide one major advantage over NURBS. They are invariant under all transformations. If you transform space, objects on one side of the NURBS surface might not end up on the same side afterward. This is not so with a rational B-spline. With rational B-splines, they will always end up on the same side afterward. However, NURBS are invariant under affine transformations and all transforms.

One great advantage of the NURBS over other splines is the ease with which their transformational characteristics can be transferred to a surface instead of a line. This is communicated by adding a second sum:

$$S(s, t) = \frac{\sum_{i=0}^p \sum_{j=0}^q P_{i,j} w_{i,j} N_{i,k}(s) N_{j,l}(t)}{\sum_{i=0}^p \sum_{j=0}^q w_{i,j} N_{i,k}(s) N_{j,l}(t)}$$

Here, the control points are now arranged in a grid, with p in one direction and q in the other, and with two knot vectors of degree k and l , respectively. In Exercise 21.1, you are asked to translate this description into an explicit algorithm.

Surfaces Generated with Sine and Cosine Functions

NURBS surfaces provide an excellent tool for creating generic solid objects, but in some circumstances effective alternatives are available. One alternative arrives when creating an infinite ground or a height map. A height map in 3-D is a function in two variables. For each value of x and z , it gives a single value y . Since you can use it to specify an entire ground surface with just a few bits of information, it is useful to have a simple method for creating such a function.

You've already seen one method for creating a height map. Presented in Chapter 15, that method used combinations of trigonometric functions. When several sine waves are combined, they produce a complex pattern. As shown in Figure 21.3, if you combine these in two directions, you end up with a random-seeming mountainous landscape.

A useful side-effect of combining trigonometric functions for generating a random landscape is that you can guarantee the maximum and minimum height of any mountain or valley. If you combine a number of waves, the maximum possible height for any mountain is just the sum of their amplitudes. Another advantage is that you don't need to remember the height map for the whole landscape. The function is fixed in advance, so you can forget about distant areas and draw them at a later stage, without having to store them explicitly.

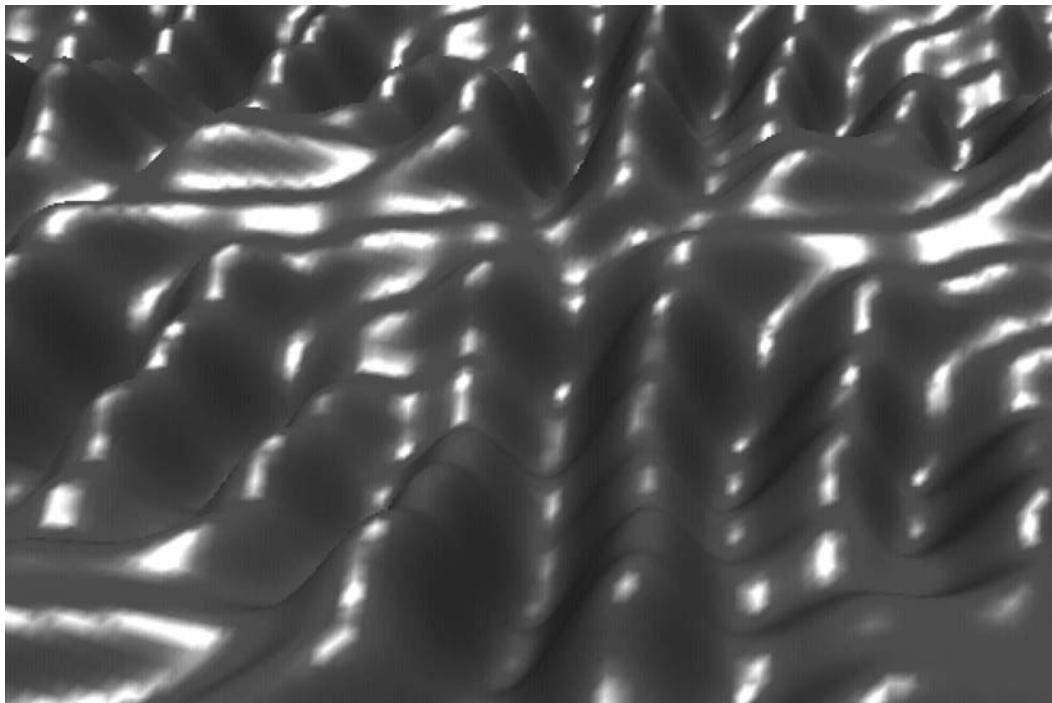


Figure 21.3

A landscape created using \sin and \cos .

Tessellation

When you specify a surface algorithmically, you can describe a smooth, richly contoured surface without using an infinite number of points. In practice, however, your 3-D engine does not display a surface in such a graceful way. It must translate the surface into a set of polygons. But if you've stored your surface as a precise, infinitely smooth curve, this enables you to create a polygonal mesh at run time. This is a process called *tessellation*.

To tessellate a surface, you begin by calculating the coordinates of a number of points on the mesh. Usually, the points are uniformly spaced out along the s and t values for a NURBS surface. Either that, or they are spaced out along the x and z values for a ground surface, such as the trigonometric surface described previously. These values are then converted to a mesh by joining them into triangles. Depending on the 3-D API you are using, you can either send them as individual triangles or as a set of vertices and a set of values defining which vertices are to be connected with triangles. For example, DirectX allows you to define triangles as lists of three points, or as a “strip” of triangles, each of which extends the strip by one point, or as a “fan” of triangles emanating from a single point.

Given that you're going to convert your surfaces to a mesh anyway, why not just store them as a mesh and be done with it? There are three principal reasons. The most obvious advantage is storage. Just as with the distinction between vector graphics and bitmaps, converting to a mesh generally requires less memory. In this case, you store a shape as a description rather than a predefined set of points. This is especially important when the shape is simple. If the shape is simple, then using NURBS is overkill, and you might be better off using an even simpler description. Consider, for example, an operation that can be described as "this is a sphere of radius 2." On the other hand, if the shape is complex, like a character, or if it is naturally composed of polygons, like a jewel, then a polygon-by-polygon description might be more appropriate. In the middle ground, while a NURBS or similar system is likely to save on memory, it does increase the load time, for the engine must build the model using the formula.

Another advantage is scaling. Scaling allows you to choose how detailed your mesh will be. The detail of a mesh is usually known by its *level of detail* (LOD), and the level of detail refers to the number of polygons used in the construction of the mesh. If your end user has a fast machine, you can create a highly detailed mesh. On the other hand, if your end user has a slower machine, you can create a less detailed mesh. Either way, the polygons are still calculated by means of the same underlying curves, and this means that they will always be good approximations to the true surface. You can even choose to vary the LOD of the mesh according to how far away it is from the camera, as is done with a mip-map. In fact, LOD is used to describe different levels of mip-maps.

Many 3-D engines can calculate LOD meshes automatically, and may even perform the switch between them according to distance. You can sometimes tell in a game the moment at which a model in the distance changes its mesh to the simpler form.

Scaling offers still another advantage. This advantage arises because scaling allows you to calculate the normal at a particular point on a surface generated algorithmically. You do this by using partial derivatives. (Partial derivatives were discussed in Chapter 6.) Use of this approach gives you two vectors tangential to the surface. As illustrated by Figure 21.4, you accomplish this by taking the cross product of these vectors. This gives you the normal.

In the case of a NURBS surface, the partial derivative of the basis functions in each direction can be described by a single polynomial of degree $k - 1$ (or $l - 1$). This gives you the constant for each knot span. You then can combine these partial derivatives into a partial derivative for the whole surface function. The outcome is that for each vertex of your mesh, in addition to calculating its position, you can calculate its normal. Similarly, you can calculate the normal for your trigonometric surface.

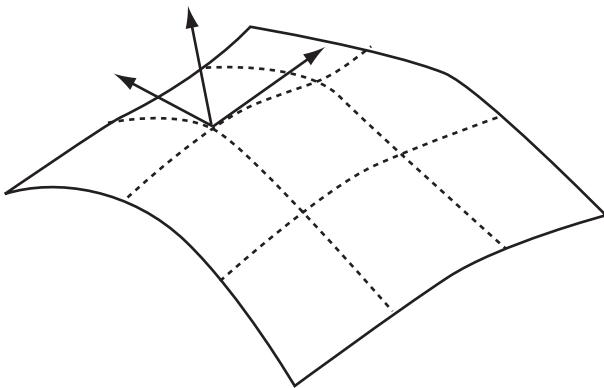


Figure 21.4
Finding the normal to a surface.

As with Bezier curves, mild advantages also accrue for these surfaces in collision detection, but the complications are large enough that using polygon-by-polygon collision detection is preferable.

Animated Surfaces

Given that it's reasonably simple to create it in real time, why not go the whole way and generate a complex surface from moment to moment, making a surface that can change over time? Several topics must be considered before an answer to this question can be given. These topics are the subject of the sections that follow.

Cloth and Hair

Although it is fair enough to conclude that erotic appeal is the sole reason that over time characters in some games have been clothed in figure-hugging bodysuits, other reasons also apply. If the garments the characters wear hang loosely, the movement of the fabric must be calculated, and such calculations are computationally intensive. As a result, during much of the history of games, an incentive has existed to use loose clothing sparingly. However, as computers have increased in speed, loose clothing has posed less of a problem, and it has become feasible to introduce a variety of loose clothing made of flowing cloth, even in real time.

Cloth, hair, and skin pose challenges to developers that are continuously the occasion for new innovations. However, on a rudimentary level, a basic trick is used in their construction. As discussed in Chapter 16, this trick involves creating a system of coupled oscillators. As illustrated by Figure 21.5, a piece of cloth is essentially a grid of particles mutually connected by springs. You've already seen how to calculate the motion of a particle attached to a system of springs. Most physics APIs allow you to create virtual springs. However, you still need to consider how best to set up such a system.

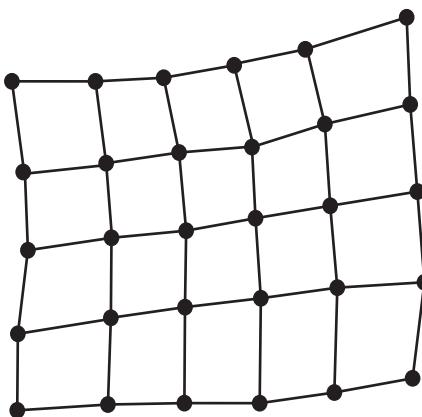
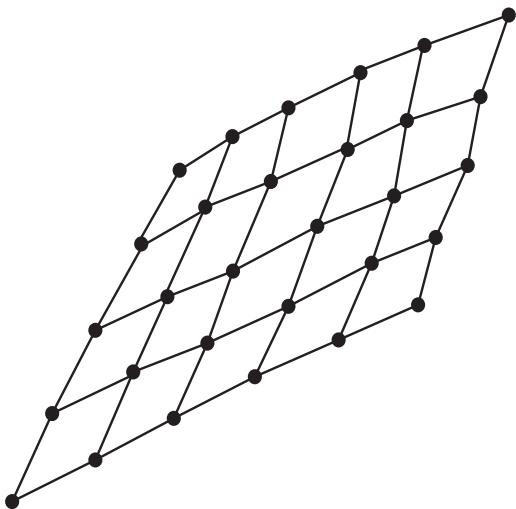


Figure 21.5
A cloth simulation.

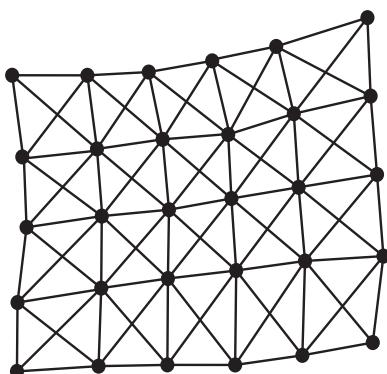
To set a system of springs, one key consideration is that real-life cloth is usually woven from a set of interlocking fibers. Interlocking makes it so that the fibers behave differently when stretched in different directions. When a force is applied along the direction of the fibers along either the *warp* or the *weft*, the cloth scarcely stretches at all. When a force is applied to stretch the fibers along the *bias* or *cross*, directions that are diagonal to the fibers, the cloth stretches easily. The behavior of the cloth ends up being analogous to a garden trellis, as shown in Figure 21.6.

Cloth as modeled for animation can be pictured as a set of inextensible springs arranged in a lattice. Such springs are actually harder to simulate than ordinary springs. An inextensible spring is essentially the same as an extensive spring with an infinite coefficient of elasticity. On the computer, setting up such a spring, especially in a coupled lattice, leads almost instantly to major feedback problems. The feedback problem occurs because, as small errors accumulate, the whole simulation spirals out of control. Adding damping doesn't help. In fact, damping tends to make the situation worse.

**Figure 21.6**

Stretching along the bias of a piece of cloth.

To overcome this problem, you must picture the cloth simulation as more like rubber. Rubber stretches equally in all directions. Using this analogy, as shown in Figure 21.7, you use springs to construct a lattice, as discussed previously, but this time you allow the springs a fairly high coefficient of elasticity. You also add cross-braces. The whole simulation is set up so that the natural length of the springs is the same as their length when the whole thing is flat.

**Figure 21.7**

A rubber cloth with cross-bracing.

To make the simulation more like skin rather than cloth, you can add an additional set of springs attaching each vertex to the underlying surface. These zero-length springs are sometimes called *dashpots*. In situations in which a surface is not the best medium, you join the springs in a chain. This approach is commonly used to simulate hair or a piece of rope.

Using the rubber surface method, you can create realistic cloth that hangs, flaps, and even drapes. For draping, you must include collision detection with the particles at the vertices. The only complication is user interaction. Because the mouse has no physical constraints in the simulated world, a user can easily create situations that are physically impossible. An example might be dragging a particle on a spring a long way past its elastic limit. Such problems can be solved fairly readily, however. In a situation in which the user can drag parts of the surface around, instead of having the user directly drag a particle, you can make is so that a force no greater than some set maximum on the particle is applied in a direction determined by the cursor. This will limit the amount by which the surface can move away from equilibrium.

Water

In computer graphics, no great divide separates water and cloth. One approach you can use to create a realistic water surface involves a series of coupled oscillators. With the use of coupled oscillations, however, the forces underlying the coupling of the oscillations of nearby points on the surface of the water are quite complicated. They involve a combination of gravitation, pressure, and surface tension.

An alternative method, involves directly modeling the waves on the surface. As with the trigonometric surfaces you looked at before, a wavy surface can be defined by a series of wave functions that can be calculated independently. The height of the surface at a particular point is just the sum of all the waves at that point.

Using wave functions or oscillations, the accuracy you achieve in your modeling depends on the approach you use. With the simplest approach, you set up each vertex of the surface as an independent oscillator under simple harmonic motion (SHM). With this model, the various vertices should all move with the same frequency, but they can vary in amplitude and phase. The variation might be specified by a function or a texture map of some kind. This is a common form of simulating moving wave surfaces in games. While this system is simple to use and reasonably fast to calculate, one disadvantage is that it lacks flexibility, particularly in relation to user interactions. It isn't affected by anything else that happens in the simulation.

If your goal is to make startlingly realistic waves, you can model them directly. The advantage of direct modeling is that the waves can then have different frequencies and can change over time. For example, simulating different weather conditions, you can make waves increase in strength over time. Waves on the surface can be of two kinds, simple parallel wavefronts moving in a straight line or circular wavefronts emanating from a point source, as happens when a pebble strikes the surface of a pond. Surface waves of either type allow you to create waves that respond to player actions.

All the methods mentioned so far create fake waves. They are fake because, among other things, they do not have crests or breakers. Crests and breakers occur when a wave moves from deep water to shallow water. When this happens, because it can't drop to its lowest negative amplitude, the wave can no longer move symmetrically, so its energy is transferred to the top of the wave. With its energy transferred to the top, the wave moves forward instead of just up and down. To model such actions involves performing many calculations, and game developers often conclude that the processing power is not available in a real-time game to support such calculations. In the area of fluid dynamics, however, such calculations are essential. Likewise, such calculations are common in the film industry, where rendering is not accomplished on a real-time basis.

Note

Still another area of computer graphics that is computationally intensive involves reflections and refractions on a water surface, and while there are a number of approaches to reflections and refraction, the topic lies beyond the scope of this chapter.

Bone Animations

The final topic of this chapter concerns using bones to create animated characters. While modeling bone tools remove the need for animators to be involved in the details of bone creation, it remains important to delve into fundamentals if mathematical and programming operations are to be understood. This section explores the essential elements only.

Working with Bones

The *bone* system has been around for a long while as a method for animating characters. It is all but universal. Using the bone system, a character is defined by a series of lines representing bones. As illustrated by Figure 21.8, these lines are arranged in a parent-child relationship. To create a model, the bones are fleshed out, and the model that results can be changed over time. Change involves rotating the bones. Each bone is rotated relative to its parent.

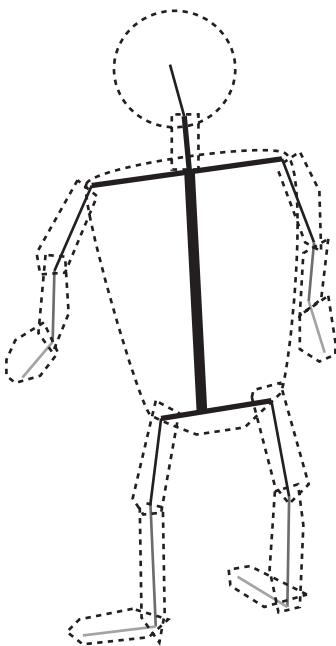


Figure 21.8

A bone system.

Since its surface must be modeled so that the skin doesn't distort at the joints when a bone is flexed particularly far, the details of how a model using bones is created are quite subtle. Modeling software can do this kind of thing automatically. In the current context, however, it's necessary only to think specifically about how bones work.

To start with, you create animation in a few basic ways. In one way, you use a pre-set series of motions, such as "run," "jump," "fall over," and so on. These are often recorded by means of a motion capture system and a live model. In another way, you directly animate each bone in real time. With the pre-set approach, the work is much easier and tends to be the most commonly used. With the direct animation approach, the work is more involved and is often used for *ragdoll animations*. With ragdoll animations, the body of the character moves as a set of connected rods under gravity, without any muscle actions.

Because the joints of a model are constrained in various ways, creating a realistic ragdoll figure is difficult. For example, the human body includes ball-and-socket joints, such as the shoulder. These joints can swivel freely in two dimensions, with limited motion in the third. Other joints, like the knee, involve simple hinges that can rotate only in one dimension. The twisting motion of the forearm is achieved by using two separate

bones rather than by motion in the elbow joint. Such constraints can be programmed using standard modeling packages, but to make them work in real time requires significant computational effort.

Making a ragdoll character move is essentially an exercise in *kinematics*. Kinematics is an area of physics that deals with the motions of systems without consideration of the physical laws accounting for the motions. With respect to character animation, kinematics deals with how the bones of a model are connected and move in relation to each other. To program kinematic actions, fairly extensive work is involved. The work combines rigid-body motion and 3-D collisions. The mathematical problems are usually solved numerically. With each time-frame, you can calculate the momentum and energy values and move the bones accordingly.

As an example of how kinematics works, consider a simple situation, illustrated by Figure 21.9, which provides a 2-D diagram. The system consists of two bones connected by a single pin. For each of the two bones, you know the current linear and angular velocity \mathbf{u}_i , ω_i and the mass and moment of inertia m_i , I_i . With 3-D animations, you would also need to know the moment of inertia about all three principal axes. For each bone, you assume that the pin is currently at a vector \mathbf{x}_i from the center of the limb. All the motion occurs in two dimensions, but without any collisions, the limbs pass over or under one another.

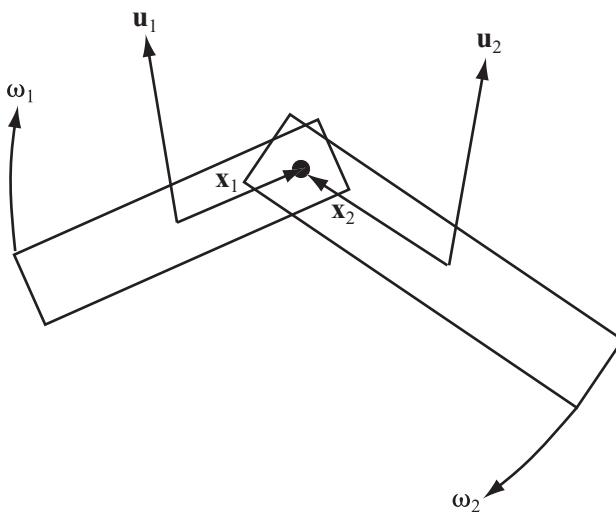


Figure 21.9

A simple ragdoll problem.

It's convenient to think in terms of the values $r_i = |\mathbf{x}_i|$ and $\mathbf{t}_i = \frac{(-x_{i2}x_{i1})^T}{r_i}$. The values establish, respectively, the distance of the pin from the center and the clockwise vector tangential to the center, through the pin. The pinning of the various limbs implies at all times that the local velocity of these points of contact must be equal. You can calculate the local velocity of a particular point as a function of the motion of the limb:

$$\mathbf{w}_i = \mathbf{u}_i + d_i \omega_i \mathbf{t}_i$$

At any time, this value must be equal for all pairs of bones at a particular pin. As presented in the following list, this reasoning gives you a set of equations for the new linear and angular velocities \mathbf{v}_i, φ_i :

- From conservation of energy, you get

$$m_1 |\mathbf{u}_1|^2 + I_1 \omega_1^2 + m_2 |\mathbf{u}_2|^2 + I_2 \omega_2^2 = m_1 |\mathbf{v}_1|^2 + I_1 \varphi_1^2 + m_2 |\mathbf{v}_2|^2 + I_2 \varphi_2^2$$

- From conservation of linear and angular momentum (as long as there are no external collisions), you have two equations:

$$m_1 \mathbf{u}_1 + m_2 \mathbf{u}_2 = m_1 \mathbf{v}_1 + m_2 \mathbf{v}_2$$

and

$$m_1 r_1 \mathbf{u}_1 \cdot \mathbf{t}_1 + m_2 r_2 \mathbf{u}_2 \cdot \mathbf{t}_2 = m_1 r_1 \mathbf{v}_1 \cdot \mathbf{t}_1 + m_2 r_2 \mathbf{v}_2 \cdot \mathbf{t}_2$$

- From the pinning, you get

$$\mathbf{v}_1 + d_1 \varphi_1 \mathbf{t}_1 = \mathbf{v}_2 + d_2 \varphi_2 \mathbf{t}_2$$

Between them, these equations give four unknowns, two of which are vectors. Although it is not easy, the four unknowns can be solved. One reason for the difficulty is that neat division does not exist for radial and tangential parts.

One disadvantage to this formulation is that there is a tendency for numerical errors to creep in, making the various bones drift apart. An alternative formulation takes advantage of the parent-child relationship of the bones. This formulation considers one bone to be the root, with the other bone(s) slaved to it. All you need to know is the angular velocity of each bone about the pivot point with its parent. While this is a simpler and appropriate formulation, it is slightly harder to set up. It will not be further explored in this context.

Inverse Kinematics

A far more difficult task is the one that our brains accomplish every second of the day. This task involves controlling each bone of your body as you engage in one or another activity. What applies to your actions applies to that of a model as it is animated. The task

of controlling the bones immediately leads to a major problem called *inverse kinematics* (IK). A question that someone working with inverse kinematics might ask is, “Which bones should I move to pick up a cup?”

Such questions are common and practical in the field of robotics, and what applies to robotics applies to animation. A wide variety of approaches have been made to solving such problems. As you will discover in Chapter 26, much interesting work has been done in this respect in the field of artificial intelligence (AI), especially as related to genetic algorithms. In this section, it is worthwhile to examine a few preliminary examples relating to inverse kinematics.

As a first example, consider the simplified bone illustrated in Figure 21.10. This model is similar to the one shown in Figure 21.9. It features a pair of jointed bones. The first bone, on the left, is fixed in place at its left end. The other end of the first bone can move freely. The second bone, which can move freely at both ends, is joined to the free end of the first bone. Suppose you want to reach the point P, touching it with the end point of the second bone.

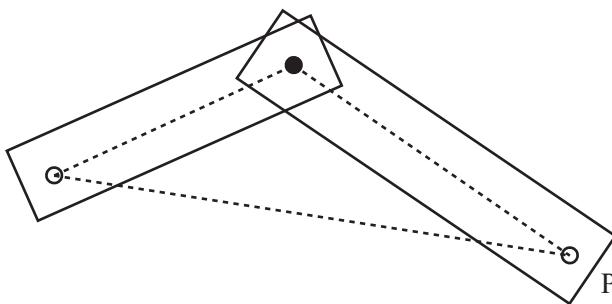


Figure 21.10
A simple IK problem.

In Figure 21.10, finding the correct final configuration to meet P is reasonably easy. It involves application of the cosine rule. You can apply the cosine rule because you know the lengths of all three sides of the triangle formed by the joint and the end points. Even with this simple scheme, however, a few complications arise. In addition to finding the end point, you must work out the best route to reach it. Ideally, you want to minimize the amount of movement required to get from the initial configuration to the end configuration. As an example of how motion can be wasted, consider the positions illustrated by Figure 21.11.

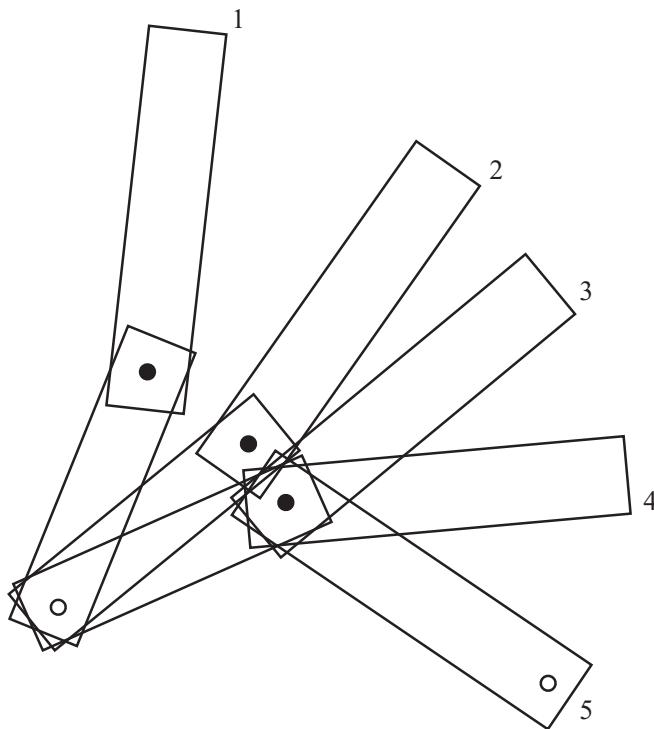


Figure 21.11

A poor solution to the IK problem.

One way to eliminate wasted motion is to create the complete motion path. To create a motion path, you attempt to move smoothly so that the free end of the second bone follows a simple line from its starting point to P, as shown in Figure 21.12. The advantage to this approach is that each movement is small and gets you progressively nearer to the goal. If the end-point is reachable, then this approach will always give a possible solution in the two-bone problem. However, it might not be physically possible if you are dealing with real characters with limited movement in the joints.

In case this seems to be overstressing the difficulty of working with inverse kinematics, remember that when there are more than two bones involved, typically there are an infinite number of possible configurations that solve the IK problem. Choosing the appropriate path for the bones is non-trivial. The best solution tends to result from an iterative approach to the solution, where the bones try to orient themselves toward the target a small amount at a time. One simple application of this method involves rotating each bone a little more toward the target at each time-step in proportion to how far away from the target it is currently. Exercise 21.2 poses this problem.

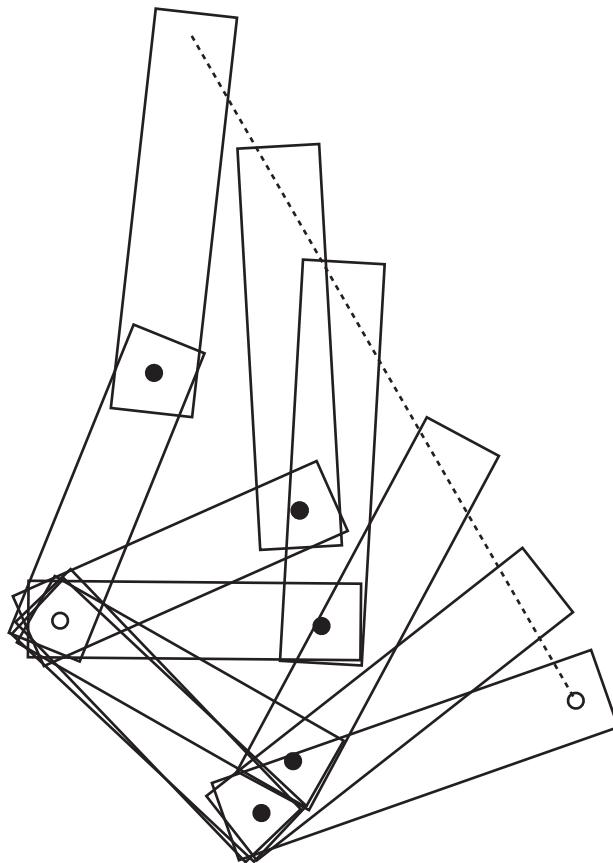


Figure 21.12

A better solution to the IK problem.

The same approach also works well in three dimensions, although it's harder to make it work realistically because there are more degrees of freedom. It's less successful when dealing with a system of bones that is not just a simple chain or when you need to worry about collision avoidance.

Exercises

EXERCISE 21.1

Create a program that will draw a 2-D NURBS. Your program should allow you to experiment with moving the control points around, changing the knot vector, and so on. If you're feeling brave, try using a 3-D NURBS surface.

EXERCISE 21.2

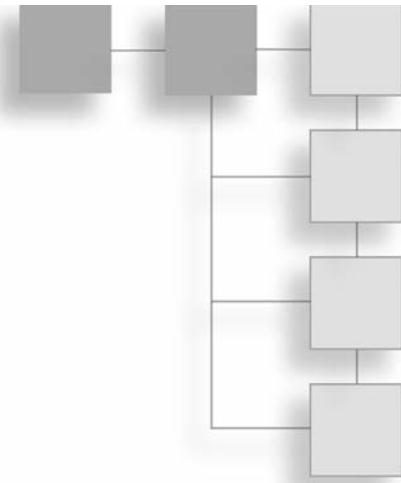
Create an iterative function `IKapproach(chain, target)` that adjusts a simple IK chain to hit a specified target. Your function should take a chain of bones specified in whatever way you prefer in a particular configuration and move it toward a particular target by some small amount. When applied successively, it should adjust the chain to hit the target.

Summary

To conclude the part of this book that discusses 3-D techniques, this chapter has led you through a brief discussion of methods dealing with surfaces. You have seen how you can use mathematical techniques to define a complex surface using various methods, especially B-splines, and how to make a surface with an adjustable level of detail. You've also looked at animated surfaces and bone systems. With the next chapter, you leave the subject of 3-D to spend the rest of the book looking at some algorithmic techniques, especially in the context of games.

You Should Now Know

- How to define an object in 3-D in terms of a *surface of revolution*, *NURBS*, or trigonometric functions
- How to use these mathematical descriptions to draw the object at different *levels of detail*
- How to animate a surface to simulate water or cloth
- How to create a bone system and use it to make a *ragdoll* simulation
- *Kinematics* and how to solve simple *inverse kinematic* problems in two dimensions



PART V

GAME ALGORITHMS

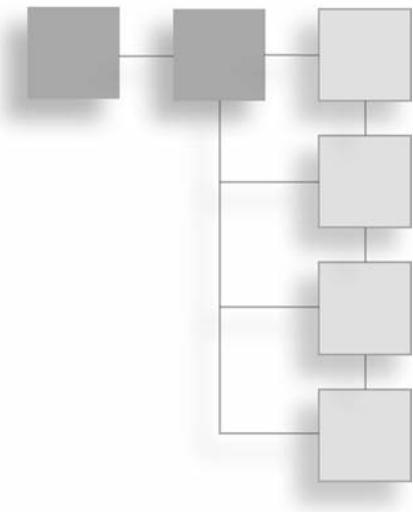
Most of the mathematics you have looked at up to now has been quite general. It has related to real-world physics and geometry and how to simulate it on the computer. But for the final part of the book, you will look at some mathematical ideas specific to computing, and games in particular. As before, I'm going to cover these concepts briefly, since each is a huge topic in itself. Still, this introduction should help you to know where to look when you encounter these issues in your work.

You'll begin by looking at some techniques for optimization and simplification of physics calculations, particularly collisions. Then you'll spend two chapters on game level design, looking at tile-based games and mazes. This leads into a discussion of pathfinding algorithms, which in turn takes you to the topic of artificial intelligence. Finally, you'll look at some techniques for using the computer as a problem-solving tool, for searching through data, and creating puzzles.

This page intentionally left blank

CHAPTER 22

SPEEDING THINGS UP



In This Chapter

- Overview
- Cheap and Expensive Calculations
- Pseudo-Physics
- Culling

Overview

Throughout this book, the emphasis has been on the mathematics and physics behind the code, and I've been stressing the fact that all code examples could be made to run faster by sensible optimizations. This chapter is the only one in which you'll be examining optimization. It offers techniques for speeding up your code by the use of pre-calculated values and segregation of space.

Cheap and Expensive Calculations

The principal key to speedy code is an understanding of which kinds of calculations are computationally *cheap* and which are more *expensive*. Cheap refers to code that takes few computer cycles. Expensive refers to code that takes many computer cycles. The more computer cycles required, the longer a process takes to complete. I'll start by discussing how the speed of a calculation can be measured, along the way looking at some ways to replace the expensive algorithms with cheaper look-up tables.

Computational Complexity

The length of time an algorithm takes is called its *computational complexity*. Computational complexity is expressed in terms of the size of the function arguments. For example, while the process of incrementing a number stored on the computer by 1 is essentially independent of the size of the number, it would be pointless to use this method to add two numbers together:

```
function sillyAdd(n1, n2)
    repeat for i=1 to n1
        add 1 to n2
    end repeat
    return n2
end function
```

The length of time this algorithm takes is roughly proportional to the size of $n1$. As this value becomes very large, the length of time for simple operations like storing the value $n2$ and returning the answer becomes increasingly irrelevant. In this respect, the algorithm is linear, and as a linear algorithm it has an *order* of 1. To express that an algorithm has a given order, you write $O(n)$, with n indicating the order.

A much more efficient algorithm for adding two numbers is the method usually presented in school. This algorithm takes the binary form discussed in Chapter 1. With this approach, if you assume that $n2$ is fixed, the time taken by the algorithm mostly depends on the number of digits in the binary representation of $n1$, which is roughly proportional to the logarithm of the number. You say, then, that the algorithm is $O(\log(n))$. The larger your numbers get, the faster this algorithm is in comparison to the `sillyAdd()` function.

The following list identifies a few of the most common kinds of algorithm:

- **Polynomial time calculations.** These have a time that is some power of n . You've already seen linear and quadratic calculations. Generally, all kinds of polynomial time algorithms are similar to them and worth aiming for. It's often the case that the rate of a polynomial calculation depends slightly on the grain size of the problem. A problem that's ostensibly quadratic might turn out to be cubic if you take into account machine-level operations.
- **Exponential time calculations.** These take a time proportional to e^n and are to be avoided at all costs since they get slow very rapidly as n increases. An example is a recursive algorithm that fills in a crossword from a list of words. It might do so by placing one word and then filling the remaining crossword. Chapter 26 looks further at algorithms of this kind.

- **Logarithmic time calculations.** There are also combinations of logarithmic and polynomial time, such as the long-multiplication algorithm, which has an order of $n\log(n)$. Logarithmic time is much faster than polynomial time and generally worth striving for.
- **Constant-time calculations.** This is the Holy Grail of algorithm creators. Such calculations are few and far between, but one example is the concatenation of two linked lists. Linked lists are chains of data in which each member of the chain contains information about itself and a pointer to the next link. To join the list together, you simply link the last link of the first chain to the first link of the second. This operation is theoretically independent of the lengths of the chain.

A few notes might be added to the preceding discussion. The base of the logarithm might be important for small values in a function, but it might also become irrelevant when dealing with very large numbers. What applies to mathematics alone might not apply to dealing with practical computations. Practical computations are just as likely to involve small numbers as large ones. If your function is proportional to $1,000,000n$, for example, and you are comparing it to another that is proportional to n^2 , then computational efficiency is important. For smaller numbers, however, the efficiency of the algorithm might not be important at all.

What applies to computational complexity also applies to *benchmark tests*. With a benchmark test, a particular calculation is run a large number of times with different methods in order to evaluate the speed of the calculation. Benchmarks need to be considered carefully in order to put them into context. After all, how often do you need to perform the same calculation a million times in succession? Usually, a calculation is a part of a larger process, and this can affect the value of the different algorithms. For example, one process might be significantly faster but involve higher memory usage. Such situations arise commonly since one of the principal ways to speed up calculations is to use look-up tables. Another process might take slightly longer but produce a number of other intermediate values that are useful later on and reduce the time taken by another process.

Using Look-Up Tables

When dealing with computationally expensive calculations, one solution is to use a *look-up table*. A look-up table is a list of values of a given function pre-calculated over a given range of inputs. A common example is the trigonometric functions. Instead of calculating $\sin(x)$ directly, you can look it up from the table by finding the nearest entries to x in the table and interpolating between them. For the trigonometric functions, you can use some

optimizations to limit the number of entries, since you only need the values of $\sin(x)$ from 0 to $\pi/2$ to calculate the complete list of entries for sine, cosine, and tangent. You can also use the table in reverse to find the inverse functions.

How many entries a table needs to contain depends on how accurate the look-up needs to be. It also depends on the interpolation method. A linear interpolation is quicker but requires more memory while a cubic interpolation, such as one based on the same principle as the Catmull-Rom splines mentioned in Chapter 10, uses fewer points but requires a little more processing. The difference is fairly significant. With no more than 15 control points, you can use cubic interpolation to calculate the values of $\sin(x)$ to an accuracy of four decimal places. (To save on calculation, you can store $15 \times 4 = 60$ cubic coefficients.) Using linear interpolation, you need well over 200 points to get the same degree of accuracy. Two hundred points with cubic interpolation is accurate to seven decimal places.

Although look-up tables can save some time, it's worth considering that the engine powering your programming language almost certainly uses a look-up table of its own to calculate such values. If you want to calculate the trigonometric, logarithmic, or exponential functions to a high degree of accuracy, it's almost certainly going to be quicker just to use the inbuilt calculations. Look-up tables are only useful if you're willing to drop some information. If you're willing to drop a great deal of information, you can ignore the interpolation step altogether and simply find the nearest point in your table.

In addition to calculating standard values, you can use a *bespoke look-up table* for your own special purposes. A common example of a bespoke look-up table is one that is used for a character and some kind of jump function. Consider, for example, the actions of a character in a platform game. Instead of calculating the motion through the air using ballistic physics each time the character jumps, you can pre-calculate the list of heights. In addition to the speed advantages, this also helps to standardize the motion across different times and machines.

Integer Calculations

As was discussed in Chapter 1, it's significantly faster to use integer values for calculations than floating-point values. Although for simplicity you've mostly ignored this issue, it's often possible to improve the speed of an algorithm by replacing floats with integers. In the simplest example, instead of finding a value as a float between 0 and 1, you find the value as an integer between, say, 1 and 1000. You can then scale all other calculations accordingly.

However, there is a problem with the integer-scaling approach. The problem arises with rounding errors. The errors that accumulate when performing multiple calculations with floats are even more prevalent when you are restricted to integers. As a result, you have to be quite clever in order to ensure that accuracy is maintained. Achieving accuracy generally means finding algorithms that rely only on initial data and not building calculations on intermediate values.

Achieving accuracy by using initial data and avoiding intermediate values can be explained most readily by means of a concrete example. Figure 22.1 illustrates a common problem scenario. In this illustration, a board is set up with a set of squares arranged uniformly. Suppose that you want to move from A to B along a grid of squares. You want to find the set of squares that most closely approximates a straight line. This is the problem that painting software faces every time it draws a line as a sequence of pixels with no gaps.

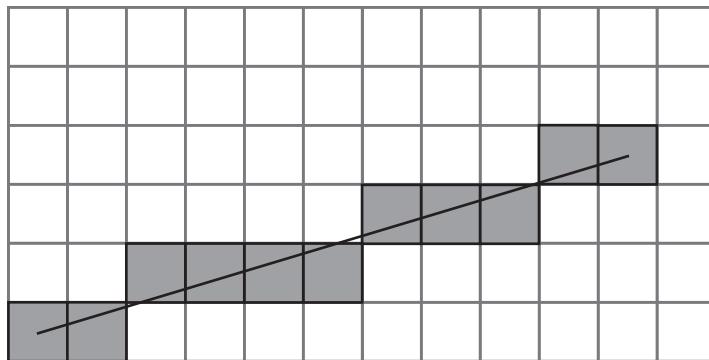


Figure 22.1

Approximating a straight line on a square grid.

The best method known for addressing the grid problem involves *Bresenham's Algorithm*. Figure 22.2 illustrates how Bresenham's Algorithm works. Suppose you are drawing a line between two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. Say for the sake of argument that $x_2 > x_1$, that all the values are integers, and that the gradient m of the line, measuring y downward, is between 0 and -1 . (In the end, adjustments will be made to these assumptions.)

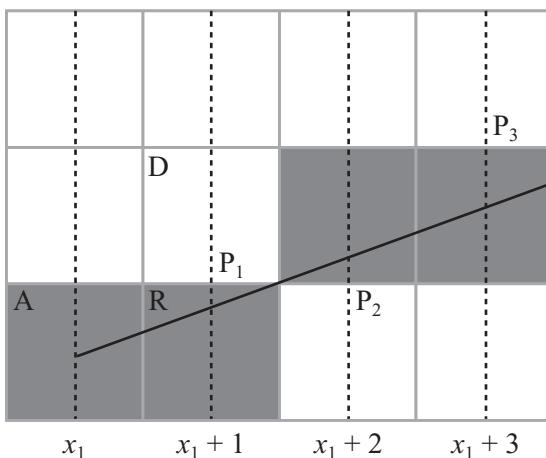


Figure 22.2

Bresenham's Algorithm.

Figure 22.2 focuses on the start of the line. The pixel at point A is filled in, and because of your assumptions about the gradient of the line, you know that the next point you're interested in is either the next pixel to the right (R) or the pixel diagonally upward (D). Which one you want to fill in depends on the slope of the line. In particular, you are interested in whether the point marked P_1 , above the point $x_1 + 1$, lies above or below the line between R and D.

You can continue this process of determining which square to fill in on a step-by-step basis, moving either right or diagonally at each point, depending on whether your line at the midpoint is above or below the half-way line. The squares you will fill in are lightly shaded in the figure. All you need is an efficient way to calculate whether the line is above or below your test position at each step such that only integer values are involved.

You can rewrite m as $\frac{a}{b}$, where $a = y_2 - y_1$ and $b = x_2 - x_1$. Note that a and b are both integers. Then your line equation $y = mx + c$ becomes $by - ax - bc = 0$, with bc found by substituting the coordinates of A into this equation, so $bc = by_1 - ax_1$. Given this start, you can define a function $L(x, y) = by - ax - bc$. For any point (x, y) above the line, $L(x, y) < 0$, and for any point below the line, $L(x, y) > 0$.

You are now in a position to create an *iterative method* of solution, which is the opposite of a recursive algorithm. An iterative algorithm works by defining two elements. The first element is what you do on the first step. The second element is how you get from any particular step to a subsequent step. For example, an iterative algorithm for climbing a flight of stairs might be “go to the bottom of the first step; then for each step, walk onto the next step until you reach the top.” In this case, you create a series of midpoints given by P_1, P_2, P_3, \dots , each of which is determined based on the previous one. For each midpoint, you calculate the value of $L(P_{i+1})$ based on the value of $L(P_i)$.

Now consider the first step, where you are moving from (x_1, y_1) . In this case, the midpoint you’re interested in is $P_1 = (x_1 + 1, y_1 - \frac{1}{2})$, and you want to know if $L(P_1) = by_1 - \frac{b}{2} - ax_1 - a - bc < 0$. Substituting in the value of bc , you get $L(P_1) = by_1 - \frac{b}{2} - a$. To keep everything in integers, you can multiply the inequality by 2 (which does not affect the result). If the inequality is true, then you move diagonally; otherwise, you move right.

Now that you know this, you can think of what you do at each further step. With reference to Figure 22.2, the situation breaks down into two cases. If at the previous step you moved right, then the midpoint you’re interested in has the same y -coordinate as before but an x -coordinate that is one greater. If you moved diagonally, then the y -coordinate must decrease by 1.

In the first case, if the last midpoint (P_i) you checked had coordinates (x_i, y_i) and you moved right, then you know that $P_{i+1} = (x_i + 1, y_i)$, so you move right again if $L(x_i + 1, y_i) < 0$. Otherwise, you move diagonally. This gives the inequality $by_i - ax_i - a - bc < 0$. Notice that this is the same as $L(x_i, y_i) - a < 0$. In the second case, having moved diagonally, $P_{i+1} = (x_i + 1, y_i - 1)$, you have $by_i - b - ax_i - a - bc < 0$, or $L(x_i, y_i) - a - b < 0$. Again, you multiply everything by 2 to ensure that you are still in integer territory.

The `drawBresenham()` function encapsulates this reasoning:

```
function drawBresenham(startCoords, endCoords)
  drawPixel(startCoords)
  set x to startCoords[1]
  set y to startCoords[2]
  set a to endCoords[1]-x
  set b to endCoords[2]-y
  set d to 2*(a+b)
  set e to 2*a
  // calculate 2L(P1)
  set linefn to -2*a-b
```

```
// perform iteration
repeat for x=x+1 to endcoords[1]
    if linefn<0 then // move diagonally
        subtract 1 from y
        subtract d from linefn
    otherwise // move right
        add e to linefn
    end if
    drawPixel(x,y)
end repeat
end function
```

The code given for the `drawBresenham()` function must be adapted to deal with the other cases, where the absolute value of the gradient is greater than 1. In such a case, you must switch x and y in the algorithm. In another case, the value of the gradient might be positive, and you must add 1 to y at each stage instead of subtracting and switch the sign of a . If the x -coordinate of the start point is greater than that of the end point, you can just switch the two points around. Exercise 22.1 challenges you to deal with such changes.

Exploration of Bresenham's Algorithm and the `drawBresenham()` function provides a good approach to understanding the principles of integer calculations. In particular, it shows you the process of approximating to integers efficiently so that you do not lose original detail. The Bresenham Algorithm is also useful in its own right, not just in drawing, but also in pathfinding, motion, and collision detection. It allows you to digitalize a world with complex lines and angles into a more tractable and faster form.

Pseudo-Physics

Having looked at simplifying motion and creating faster paths, it is natural to pursue optimization further by looking at what might be called *pseudo-physics*. Pseudo-physics involves motion that looks realistic but is actually simplified or faked. In general, the approaches offered by pseudo-physics for solving problems of simulation can be referred to as *approximate solutions*.

Simplifying Collisions

As you've already seen, calculating collisions can get fairly unpleasant fairly quickly. Even a simple problem like finding the point of collision between two ellipses or a circle and a rotating line leads to situations that require a numerical solution. Numerical solutions are usually computationally expensive.

If your time slices are small enough, you can fake collisions to a reasonable degree of accuracy using a simple process. The faking involves two steps. First, you find out whether a collision has occurred during a particular time slice. Next, you make a guess as to when it happened. In most cases, this process is not too hard to implement and produces a fairly believable effect. While there are a few potential pitfalls, you can mostly get around them.

To examine a simple example, consider the collision of two ellipses. You can reduce the collision of two ellipses to the collision of one ellipse aligned along the x -axis with a unit circle. Suppose, then, that you have an ellipse $E(\mathbf{p}, (1 \ 0)^T, a, b)$ moving along a displacement vector \mathbf{v} , and you want to see if it collides with the circle $C(0,1)$.

If your time slice is small enough, then for the ellipse to collide during this time period, it must intersect the circle at the end of the time designated. If it doesn't, then the only way it can collide is if it glances off the circle at a very shallow angle. The angle can be ignored since such collisions don't have much effect.

Deciding if there is a (pseudo-)collision is essentially the same as calculating if the ellipse $E'(\mathbf{p} + \mathbf{v}, (1 \ 0)^T, a, b)$ intersects with C . In other words, you are looking for a point on E' at a distance less than 1 from the origin:

$$(a \sin \theta + q_1)^2 + (b \cos \theta + q_2)^2 < 1$$

where $\mathbf{q} = \mathbf{p} + \mathbf{v}$. You can carry out this operation by searching for the minimum value of this expression. You find the minimum value by differentiating and setting the derivative to 0:

$$\begin{aligned} 2a(a \sin \theta + q_1) \cos \theta - 2b(b \cos \theta + q_2) \sin \theta &= 0 \\ a \sin \theta + q_1 &= \frac{b}{a}(b \cos \theta + q_2) \tan \theta \end{aligned}$$

Substituting this back into the inequality, you are looking for

$$\begin{aligned} \left(\frac{b}{a}(b \cos \theta + q_2) \tan \theta \right)^2 + (b \cos \theta + q_2)^2 &< 1 \\ \left(\frac{b^2}{a^2} \tan^2 \theta + 1 \right) (b \cos \theta + q_2)^2 &< 1 \end{aligned}$$

Given that $\tan^2 \theta = \frac{1 - \cos^2 \theta}{\cos^2 \theta}$, with a little algebraic manipulation, you end up with the following quartic inequality:

$$b^2 d^2 c^4 + 2bd^2 q_2 c^3 + (b^4 + q_2 d^2 - a^2)c^2 + 2b^3 q_2 c + b^2 q_2^2 < 0$$

Where $c = \cos\theta$. You can solve this inequality algebraically. If it yields a value of c between -1 and 1 , you know that a collision occurs.

You can make similar simplifications when dealing with rotational motion. For example, collisions between two moving objects that are spinning can be calculated by ignoring the spin. At each stage, you calculate a linear collision between the objects in their current orientation. This will not work, however, when neither object is moving linearly. Likewise, you will miss some collisions on the fringes, where they are less noticeable.

Simplifying Motion

Pseudo-physics can be used to deal with several other types of motion. One of these is friction. In the pool game explored in previous chapters, you implemented a simple system of pseudo-friction in which the balls slowed down at a constant rate. As a general rule, you can simplify things roughly in proportion to how attuned people are to them. If your simulation is for people who possess a very accurate intuition about objects in flight, for example, then the ballistics of your simulation must be implemented accurately. On the other hand, the intuitive understanding most people have of oscillations is usually not very strong, so you are not likely to encounter strong objections if you present a fairly rough simulation of objects on springs. An example would be applying a uniform backward acceleration whenever a spring is extended, combined, perhaps, with a uniform frictional deceleration applied at all times.

Similarly, most people have a natural understanding of linear momentum and energy but tend to be weak with respect to their understanding of angular motion. For this reason, it is predictable that many people find some of the phenomena associated with spin strange or magical. For this reason, you are taking few risks if you significantly simplify angular motion. An example in this respect is resolving a spinning collision by applying an additional impulse in the direction of spin instead of accurately resolving the angular and linear components.

In most game contexts, people are often more comfortable with pseudo-physics than with highly precise physical simulations. Consider an automobile racing game. If such a game uses realistic physics for driving a racing car or a four-by-four, it is likely the user will find the vehicles much harder to control than if they are governed by simplified, discrete motion. Such motion applies to forward, backward, left, and right movements. Due in part to gameplay satisfaction, game designers have, as a rule, differed on whether it is essential to make real-time physics a genuine part of the game. What happens, for example, if the difficulty of your game were to make it impossible for the majority of your potential customers to win? Exercise 22.2 asks you to explore this topic further.

Culling

Culling has been mentioned a few times in this text, in conjunction with collision and visibility, but it has not yet been treated in detail. Culling is the process of quickly eliminating obvious non-candidates in a particular problem, such as the objects behind the camera when determining what parts of a scene are visible. There are a number of useful techniques that can be used to improve culling. Most of these fall under the category of *partitioning trees*.

Segregating Space

Recall that in Chapter 11 you explored techniques for partitioning or segregating the game world into smaller chunks. This technique can be generalized in two ways. One is to make the chunks smaller and tie them in to the actual game world design, which leads to the tile-based games you'll examine in the next chapter. The other is to create a recursive process that organizes the world into regions within regions, creating a partitioning tree. In this section, a number of methods for developing partitioning trees are discussed. All of these methods work equally well in two or three dimensions, but the discussion concentrates on 2-D examples because you rarely need a 3-D implementation of them. Three-dimensional games tend to take place on a ground of some kind, making them essentially 2-D (or 2.5-D). Only a few games, such as space battles, fully use three dimensions.

A partitioning tree is a data structure that stores information about the world in some form of hierarchy. The form of the tree is essentially the same as the system used in Chapter 18 to deal with relative parent-child transforms. It consists of a number of *nodes*, all but one of which has a single *parent* and zero or more *children*. A *childless node* is called a *leaf*, and the topmost, *parentless node* is called the *root*. As you might infer from the discussion presented here, trees are generally drawn upside down. A tree contains no loops. No node can be its own parent, grandparent, or other relative. Information used to make calculations is associated with each node, and in this realm, you require an understanding of object-oriented programming.

Note

That a tree is a special case of a graph is a notion that will be explored more extensively in Chapter 24.

In a partitioning tree, the root node represents the whole world, which is then split into smaller (generally non-overlapping) parts based on some criterion. These parts are then split again, and again, and so on, until some pre-set condition is reached, such as a minimum size. The partition is usually performed in such a way that calculations such as raycasting or collision detection are simplified. In particular, if something is not true of a parent node (visibility, proximity, and so on), then it is not true of any of the parent's children, which means that you need not bother checking any further along the branch of the tree associated with the parent node.

As was pointed out in Chapter 11, partitioning trees can't solve all ills and shouldn't be expected to. For example, checking each parent node in itself adds an additional calculation. If you happen to be looking toward every object in the world, then you're going to be performing all your visibility culling calculations for nothing. Similarly, if you're performing a break in a game of pool, you can't avoid the fact that every ball on the table is involved and has to be checked. Nevertheless, in many circumstances partitioning space is helpful, and in some cases, such as working on a vast outdoor terrain, it is essential.

Quadtrees and Octrees

Probably the simplest and most common form of partitioning tree is the *quadtree*, or its 3-D equivalent, the *octree*. A quadtree is the most natural extension of the system you looked at in the pool game, in which the world is broken up into square regions. Suppose you start with one large square that encompasses the whole 2-D plane. This is the root node. You can break up the root node into four equal squares, which are its children. Each of the children in turn, can be broken into four smaller squares. As shown in the tartan in Figure 22.3, the process of breaking up child nodes can continue until your squares are a pre-set minimum size.

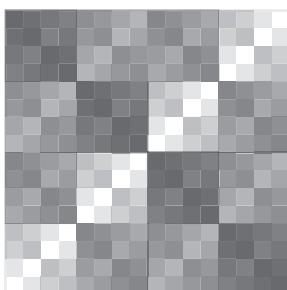


Figure 22.3

A 2-D plane broken into a quadtree.

Associated with each leaf node is the set of objects contained in the node's area, which might be either complete geometrical objects, such as balls, or individual line segments of a polygon. (With a 3-D mesh, multiple polygons would be used.) Objects in more than one square are associated with all these leaves. Associated with each parent node is its set of vertices on the plane and, possibly, a duplicate set of all the objects it contains.

Various algorithms for searching a tree have been developed, dealing with a variety of problems. To examine one example of searching a tree, consider collision detection. When detecting objects that are colliding, for each object you cull out any parent nodes whose partition does not contain any part of the object's trajectory during a given time slice. As shown in Figure 22.4, you begin by removing the partitions shaded with the darkest gray. Next, you remove the children shaded with the next darkest. The process continues, until you end up with a set of leaves that intersect the motion. You can then quickly check for collisions with any objects in these leaves.

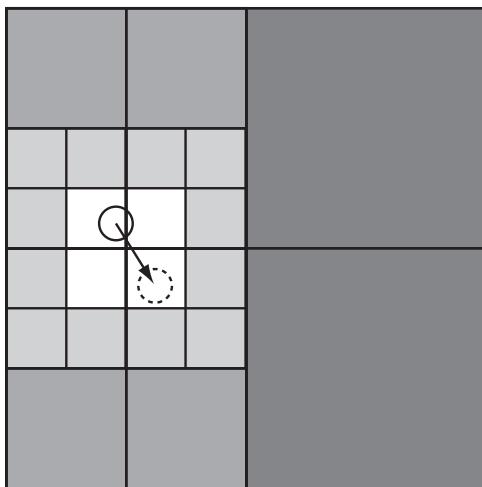


Figure 22.4
Culling for collision detection with a quadtree.

Despite the simplicity of this algorithm, it is not significantly quicker than just partitioning the space into a flat grid. In some circumstances, it can even be slower. However, to speed things up, you can pre-compute sets of nodes that can be culled automatically for an object in a given leaf. If the amount of motion is small relative to the size of the grid, you can assume that an object in the middle of a first-level partition can't ever get to one of the other first-level nodes. While effective generally, this approach won't work for nodes on the edges of a particular parent.

Quadtrees are useful for visibility determination because they allow you to reduce the number of leaves you need to consider in your calculations. They also can be useful for raycasting. With raycasting, you can determine the set of parent nodes intersecting with a ray. If a parent node is known to be empty, then you can ignore it; otherwise, you recurse over its children. One project worth attempting is to adapt Bresenham's Algorithm to perform this operation using a fast integer-only algorithm.

At the object level, quadtrees are most useful for storing a collision map. Figure 22.5 illustrates a collision map that has been divided into a quadtree using the following algorithm: If a particular partition is either completely empty or completely full (partitions A and B, for example), you stop subdividing the tree at that stage; otherwise, you recurse over its children. You can further optimize this algorithm by stopping whenever a particular node contains a flat edge. When detecting collisions between such shapes, you work down through the nodes of the trees until either a collision is detected (a black leaf from each shape intersects) or all nodes at that level in one or another shape are empty.

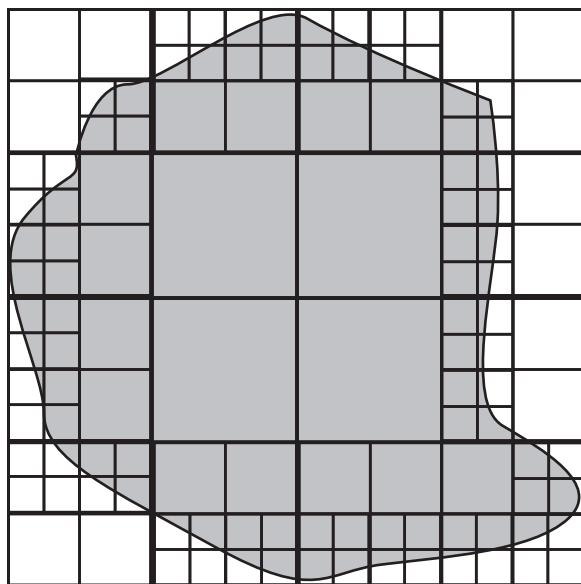


Figure 22.5
A collision map organized as a quadtree.

Binary-Space Partitioning

A second type of partition is the *binary space partitioning tree*, or *BSP tree*. Unlike quadtrees, BSP trees use two children for each non-leaf node, so each partition is divided into two parts at each stage, either by a line in 2-D or a plane in 3-D. You can choose various approaches to making the partition. One method of accomplishing this involves making each leaf node contain exactly one object, as is illustrated by Figure 22.6. The object might be one line segment or polygon. A common technique is to use a split line oriented along a particular line segment or polygon of the partitioned region. Using this approach means that your BSP tree has to be recalculated every time an object moves out of its current partition. For this reason, this kind of tree is best suited for a scene that is mostly static. Each node of the BSP tree keeps a record of all objects that are entirely contained within it.

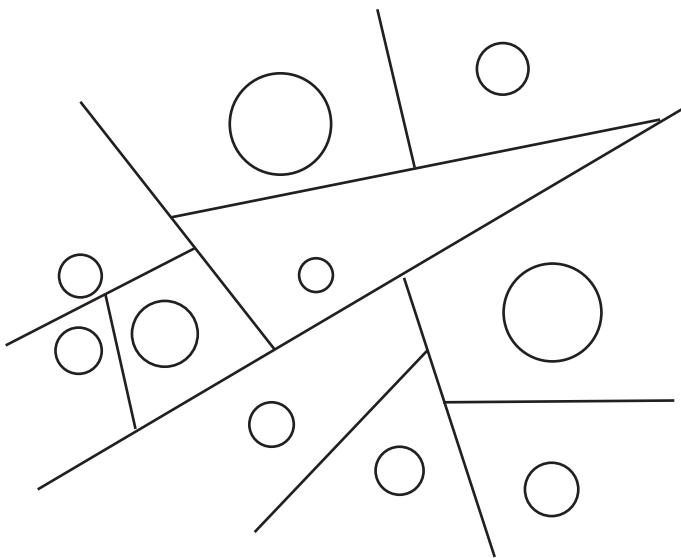


Figure 22.6

A binary partition of a plane.

One useful feature of a BSP tree is that all the regions are convex. For this reason, a BSP tree provides a neat way to divide a concave object into convex parts. As with quadtrees, BSP trees are best suited at the world level and to working with visibility and raycasting. At the object level, they can also be helpful for detecting object-to-object collisions.

Bounding Volume Hierarchies

A *bounding volume hierarchy*, or BVH, is in some ways the opposite of the space partitions examined so far. Instead of concentrating on space, a BVH focuses on the objects and organizes them according to *bounding shapes*. The BVH is most commonly used in complex 3-D worlds, but it is equally applicable to 2-D. The bounding shapes might be spheres, AABBs, OABBs, or any other useful armature.

Note

OABB is the abbreviation for *object-aligned bounding box*. AABB is the abbreviation for *axis-aligned bounding box*. AABB is said to be faster but less accurate.

The crucial difference between this approach and those discussed previously is that a BVH does not in general cover the whole game world. It does not do so because large regions of empty space are likely to be outside the bounding volumes and, on the other hand, its various volumes can overlap. Figure 22.7 shows an example of a 2-D bounding area hierarchy in which the bounding shapes are circles.

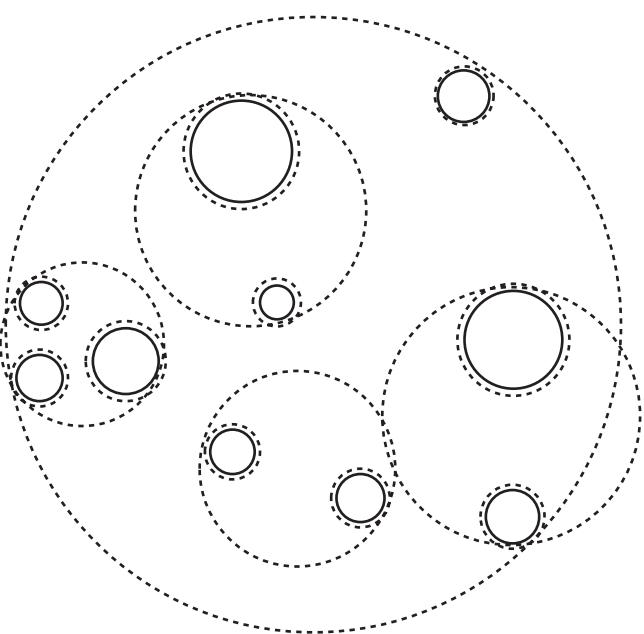


Figure 22.7

A 2-D bounding area hierarchy for segregating a space.

BVHs are an extremely useful tool for culling and have a number of advantages over other systems, particularly in a game world made up of many moving objects. They can also be combined with space partitioning systems in various ways.

The most difficult part of working with a BVH is constructing the tree in the first place. Since you want it to be as efficient as possible, at each level of the tree, you make the bounding volumes as small as possible. Depending on the way your game is constructed, you can choose a number of different systems. In particular, objects that generally appear together should be grouped together. Objects that are moving relative to one another should not be grouped together. Within an object, you can also create a hierarchy of bounding volumes that gradually approaches the shape of the object. Figure 22.8 shows an object surrounded by a hierarchy of smaller bounding OABBS.

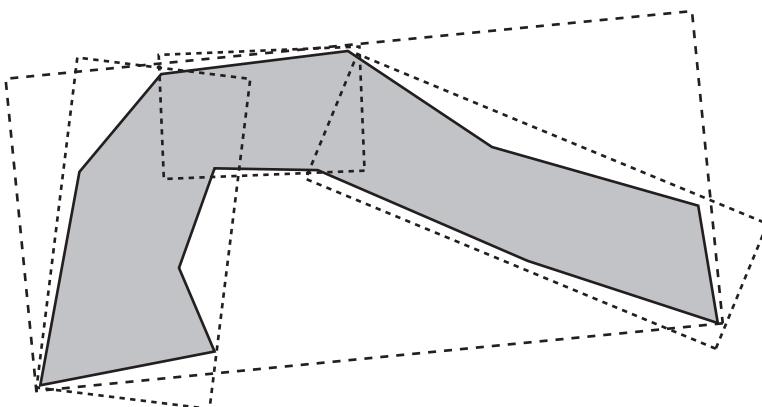


Figure 22.8

Using a BVH to whittle down the geometry of a shape.

When objects are moving significantly relative to one another, you must create a more dynamic BVH. Creating a more dynamic BVH involves quickly splitting, merging, and inserting new nodes in the tree. This is a complex process, but a great deal of work has been done on the subject, and it is worth looking into if you want to explore partitioning tree systems further, particularly if you are dealing with a large, complex world.

In addition to the techniques mentioned in this chapter, keep in mind that there are other systems of culling. Among many are those using visibility graphs, which store details for visible areas. Investing the available technologies amounts to an extended task that can end up leading you into a very specialized line of work. Hopefully, this taster gives you some insight into the tricks of this very interesting trade. Further references can be found in Appendix D.

Exercises

EXERCISE 22.1

Complete the `drawBresenham()` function by extending it to deal with all possible gradients. Some hints as to how to complete this exercise are given in the description of the function.

EXERCISE 22.2

Write a function that controls a car with the cursor keys using simplified physics. The standard car-control system used in most games employs the up and down arrows to accelerate and brake and the left and right arrows to turn. The orientation is in the forward direction, so a left turn makes you point right if you’re moving backward. If you have the urge, you might want to extend this with a skid option. If the momentum perpendicular to the forward direction is above some threshold, then the turn function ceases to be effective.

Summary

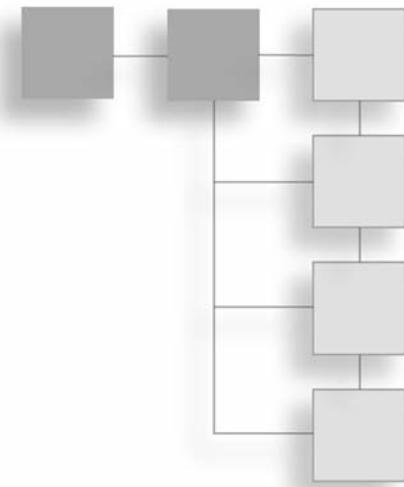
It’s unfortunate that so little space is available to discuss what amounts to one of the major areas of contemporary game programming, but as it is, the topics presented in this chapter are peripheral to the central concerns of this book. It is hoped, however, that if the topics have tantalized you, you will make use of some of the listings in Appendix D to extend your explorations. At the very least, these few pages should have given you some insights. You’ve seen how particular calculations can be simplified and optimized, and you’ve looked briefly at some of the techniques for culling and optimizing game world data. In the next chapter, you’ll explore the other side of the partitioning coin by creating the partition first and making the game to fit it.

You Should Now Know

- How you measure complexity using $O(n)$ notation
- How to use *Bresenham’s Algorithm* to calculate a path
- How to use *approximate solutions* to create faked motion and collisions
- How to use a *quadtree*, *BSP tree*, or *BVH* to speed up visibility and collision determination

CHAPTER 23

TILE-BASED GAMES



In This Chapter

- Overview
- Generating a Game from Bits
- Basic Movements and Camera Control
- Advanced Tiling

Overview

One of the most common genres of games is the *platform game*. In a platform game, a character must negotiate around a world made up of greater and lesser obstacles, usually placed on several levels, to reach an ultimate goal. This genre encompasses 2-D games, such as *Super Mario World*, and less obvious 3-D adventures, such as *Tomb Raider*, *Sonic*, and the *Sim* series. In their initial versions, all of these games were based on the same basic design. If you visit a site like Kongregate.com, you can see that this tradition is still very much alive in online games. Thousands of games built with Flash populate the site, and more are added each day by individual developers and game companies. With many platform games, the game worlds are made of *tiles*, or small, identical pieces. This system has many advantages, including ease of designing levels, lower memory costs due to reuse of graphics, the possibility of creating a generic interaction and navigation system, and simplification of collision detection. While some of these topics are outside the scope of this book, the last two issues are relevant and are discussed in this chapter.

Generating a Game from Bits

The topic of tiling is extensive, but one way to begin to examine it is to look at the essentials of a system and how it can be used to generate, store, display, and run games. This activity involves constructing the game world and populating it with characters and other features.

Creating a Tile-Based World

A *tile-based game* (TBG) begins life in a simple program called a *level editor*, where individual tiles from a graphics gallery can be dragged into a grid of squares. Here, the level designer can create the level, designing the various puzzles. The level editor varies in complexity according to the details of the game, but the end result is much the same. Each square is assigned to a particular tile. This grid data is then transferred into memory when the level is loaded, in the form of an array.

Each tile can have several properties. For example, a piece of ground might be able to melt, have a low friction value, be deadly to the touch, or undulate like water. As interesting as the visible presentation might be, such details are not as important as the fact that you can store data about collisions in tiles.

On top of the game world, you create a number of moving characters or enemies. You also create scenery, some of it moving, as with floating platforms. Such objects enhance the game environment. To complement them, you place discrete objects such as obstacles or tokens, and while these do not need to correspond to tiles, they nevertheless add complexity that must be taken into consideration as you work with tiles. You also might create some kind of background image. This is made from tiles and involves work with camera control.

With respect to the general project of developing a tile-based game, all of these elements can be viewed as comprising the game engine. From a summary perspective, then, the game engine has four main ingredients: the tile data, the fixed objects, the moving environment objects, and the characters, including the player. As the game is played, the engine has to composite these images into a single picture on the screen.

Basic Movement and Camera Control

Movement in the game world takes place in its own coordinate system, which is usually called *game space*. The game space houses the game world. Each moving object has a position relative to the origin of the game space. Usually, this origin lies at the top left of the grid. All calculations of collisions take place in this grid, or coordinate system. Your

screen can be imagined as a “viewport” into the game world, and an independent or semi-independent camera object follows the player character as it moves about the game world. Whether the camera is independent or not depends on such factors as whether the player directly controls it. In a simple 2-D view, you can consider the camera to be at a particular distance from the main plane of the game. One pixel of the game world might correspond to one pixel on the screen.

The `drawWorld()` function does what its name implies. Given a particular camera position, it draws a selective portion of the game world in the viewport. The assumption is made that the camera position in the game world is set at the coordinates of the pixel at the middle of the screen:

```
function drawWorld(tileList, cameraX, cameraY, w, h)
    set x to cameraX-w/2
    set y to cameraY-h/2
    set hoffset to (x mod 16) // assuming 16-pixel tiles
    set voffset to (y mod 16)
    set leftmost to 1+(x-hoffset)/16
    set topmost to 1+(y-voffset)/16
    // add error checks to ensure you aren't out of range
    set largeImage to an empty buffer
    repeat for i=0 to w/16
        repeat for j=0 to h/16
            draw tileList[i+leftmost, j+topmost] to
                square(i*16,j*16,(i+1)*16,(j+1)*16) of largeImage
        end repeat
    end repeat
    copy rectangle(hoffset,voffset,hoffset+w,voffset+h)
        of largeImage to screen
end function
```

The `drawWorld()` function is generalized to the point that it is too inefficient to use as given, but it still provides the basic features of a function that creates a game world. Sixteen-pixel tiles are used, for example, and a 2-D world is spawned. In practice, there are different ways by which such a function might be optimized. For a game that scrolls in only one direction (a *side-scroller* or a *top-scroller*), it makes sense to draw the world column by column or row by row, adding a new chunk to the image each time another chunk goes out of range. This saves compositing time but is slow when dealing with an image that scrolls in all directions. It is also inefficient, since the chunks of the image must be repeatedly calculated if the player chooses to go backward. Depending on the size of the level, an alternative is to draw the whole level at the start into a single image held in memory. You then copy portions of this image to the screen as needed.

At the expense of some processing speed, you make the world image more interesting if you add image layers to create *parallax scrolling*. Parallax scrolling involves using background or foreground images. These images are placed at a distance from the camera that differs from the focal plane. The parallax scenery then scrolls at a different speed. As an example of how to implement parallax scrolling, you might create a background image for which one pixel of the image represents two pixels of world space. With this approach, as perceived the image is twice as far away as the focal plane.

Having drawn your tiled planes, you then have to draw all the movable objects on top of them. This is a matter of translating their positions from world coordinates to screen coordinates. Given that their positions are based on the focal plane, you can subtract the position of the top-left of the viewport from their position to work out where they are placed onscreen. In some cases, they might not be placed onscreen.

The final question you face is where to place the camera. One starting rule with respect to this question is that it's advisable not to have the camera linked directly to the character position. A reason for this is that when the character is at the edges of the game world, images with a lot of blank space at the sides are produced. At the very least, you want to offset the camera whenever the character approaches an edge.

More generally, you want to have a camera that floats behind the character and generally acts like a real camera, following the character around by interpolation. For example, when the character moves in a given direction, the camera can lag slightly behind. You can also choose to focus to one side of the character to take into account the direction in which the character is looking. In a side-scroller, for example, you're usually much more interested in what's in front of the character than what's behind. When the character stops, you don't want it to be in the center of the screen. On the other hand, if the character is looking right, you want it to be somewhere to the left of center.

Basic Collisions

After the game world is in place, your next task is to create the functionality that allows you to detect collisions. Apart from the advantages in terms of memory and speed, the biggest advantage of using tiles is that it dramatically simplifies collision detection. How you approach collision detection depends on whether the landscape is defined as solid or permeable. You can choose one or the other or combine the two. If you combine the two, you might make tiles that are solid only from the top. This is a common convention for platform games. With such tiles, when the character is moving, you need to check for collisions only when some part of the character is entering a new tile. This approach is used with the `detectCollisionWithWorld()` function:

```
detectCollisionWithWorld(c, w, h, displacement, tiles)
  // w and h are the width and height of the box
  // c is the top-left corner

  // determine the colliding edges
  if displacement[1]=0 then set edge1 to "none"
  else if displacement[1]>0 then set edge1 to c[1]+w
  else set edge1 to c[1]

  if displacement[2]=0 then set edge2 to "none"
  else if displacement[2]>0 then set edge2 to c[2]+h
  else set edge2 to c[2]

  // calculate first collision
  set t1 to 2 // time to collision along vertical edge
  if edge1<>"none" then
    set currTileX to ceil(edge1/16.0)
    set newTileX to ceil((edge1+displacement[1])/16.0)
    if currTileX>newTileX then
      set t1 to ((currTileX-1)*16.0-edge1)/displacement[1]
    otherwise if currTileX<newTileX then
      set t1 to (currTileX*16.0-edge1)/displacement[1]
    end if
  end if

  set t2 to 2 // time to collision along horizontal edge
  if edge2<>"none" then
    set currTileY to ceil(edge2/16.0)
    set newTileY to ceil((edge2+displacement[2])/16.0)
    if currTileY>newTileY then
      set t2 to ((currTileY-1)*16.0-edge2)/displacement[2]
    otherwise if currTileY<newTileY then
      set t2 to (currTileY*16.0-edge2)/displacement[2]
    end if
  end if

  if min(t1,t2)=2 then return "none" // no change of tile
```

```
if t2<t1 then // first collision is along horizontal
    set newTile to newTileY
    set currTile to currTileY
    set checktile to [ceil(c[1]/16.0), ceil((c[1]+w)/16.0)]
    set mx to the number of columns of tiles
    if displacement[2]>0 then set dir to "bottom"
    otherwise set dir to "top"
otherwise
    set newTile to newTileX
    set currTile to currTileX
    set checktile to [ceil(c[2]/16.0), ceil((c[2]+h)/16.0)]
    set mx to the number of rows of tiles
    if displacement[1]>0 then set dir to "right"
    otherwise set dir to "left"
end if
// at the end of the above process, newtile and currtile give the
// changed row or column, checktile gives the start and finish
// of the tiles containing the player

set t to min(t1, t2)

if newTile<1 or newTile>mx then
    // edge of map
    return [c+t*displacement, (0,0),dir]
end if

// check whether any new tiles entered are solid
repeat with i=checktile[1] to checktile[2]
    if dir="bottom" or dir="top"
        then set tile to ptiles[newTile][i]
    otherwise set tile to ptiles[i][newTile]

    if tile is not empty then
        // potential collision
        if tile.solidity="solid" or
            (tile.solidity="top" and dir="bottom") then
                // tile collision (there could be more options here)
                // move to collision point
```

```

        return [c+t*displacement, (0,0),dir]
    end if
end if
end repeat

// no collision: recurse
if t=0 then set t to 0.001
return detectCollisionWithWorld(
    c+t*displacement, w, h, (1-t)*displacement)

end function

```

Note

It's conventional and convenient to consider the moving character to be an axis-aligned bounding box. You can also apply a similar approach to other objects in the game.

To augment the `detectCollisionWithWorld()` function, you can make use of techniques introduced in Chapter 21. For example, you might pre-calculate the heights of a jumping character over time and store them in a table. You can also save on calculations. Consider, for example, a character that is known to be currently walking on the ground. In a profile game (*Mario*, for instance), this character is necessarily at the junction between two tiles in the vertical direction. When the character is in this position, the first thing to check is whether it is still standing on solid ground, since this is always the immediate collision. Only then do you need to worry about collisions in the horizontal direction.

With respect to how you govern movements and the results of collisions, it is important to recognize that the physics in platform games is seldom realistic. Consider, for example, that it is sometimes possible to change direction in mid-air while jumping, and objects do not as a rule bounce when they hit the ground. As mentioned in Chapter 22, such behavior often makes the game more appealing to the player.

Complex Tiles

As discussed in the previous section, most TBGs use solid tiles. However, other options are available. Among other things, you might employ a collision map to subdivide a tile into smaller pieces. If you use this approach, as Figure 23.1 illustrates, instead of calculating collisions according to whether a tile is wholly solid or empty, you can employ the collision map so that collisions can be detected on the basis that the tile is solid or empty only in places.

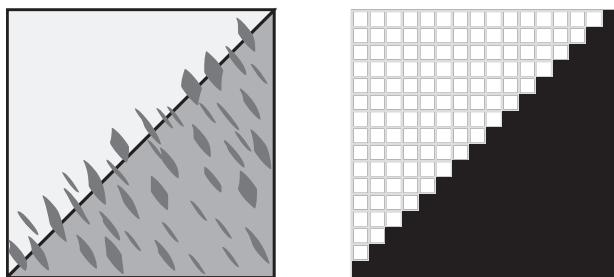


Figure 23.1

A tile with its associated collision map.

To use the collision map as shown in Figure 23.1, you store the normal details along with the collision information. Since this approach makes collision detection more complicated, it's sensible to limit the number of complex tiles in use. You can also consider using a vector approach instead of a collision map bitmap. If you use the vector approach, the tile illustrated in Figure 23.1 might be thought of as "a wall from top-right to bottom-left." While less versatile, this approach increases the speed of collision detection.

Another extension of the tile concept involves tiles that change over time. Tiles change in different ways. They can change continuously, as with a conveyor belt, or they can change when triggered by some action, as when a cube of ice melts when stood on. In principle, such changes aren't complicated, but they are awkward to deal with in the compositing process. To save time, you don't want to redraw the whole level each time a particular tile changes. Instead, you need only to change the tiles that are currently in view. To accomplish this, you can draw such tiles over the existing level image.

As a further extension, tiles that move from place to place, such as moving platforms, can be dealt with as characters rather than level tiles. Such tiles can be drawn on top of the level image after the fixed world has been copied across. It's a judgment call regarding other examples. Consider, for example, growing trees, exploding barrels, or whatever else your imagination comes up with. Whether you want to see them as discrete objects drawn on top of the world or tiles in their own right depends on how much work you want to do and how much you feel your game must be optimized. In general, if something can be a tile, to increase performance, it's better to explore optimization but not necessarily to assume that it is absolutely essential.

Advanced Tiling

So far in this chapter, the discussion has centered on 2-D TBGs, but the techniques that apply to 2-D TBGs apply just as well in a 3-D or a “2.5-D” world in which most of the action takes place on a flat plane. The techniques can even be used for games in which the action takes place on a spline surface.

The Isometric View

The most traditional form of tile-based 3-D game features an isometric world, which you encountered in Chapter 17. Isometric worlds do not differ greatly from 2-D TBG worlds. As illustrated by Figure 23.2, the only significant difference is that the tiles of isometric worlds, representing areas of ground, are drawn as rhombuses instead of squares. However, this is not the whole story. Since isometric worlds are 3-D worlds, the tiles of isometric worlds have height as well as length and width. Generally, the height of a tile is stored along with the tile. When you draw the tile at the appropriate position, it is offset by its stored height.

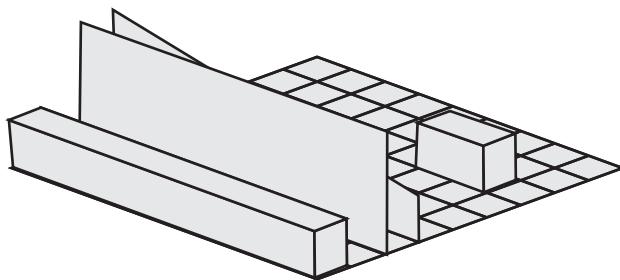


Figure 23.2

The world of an isometric game.

Many developers today take advantage of 3-D acceleration to display the features of isometric worlds. However, the worlds of the original games of this type were drawn in real time using an approach similar to the one discussed in the previous section for the 2-D TBG.

One of the hardest parts of creating a good isometric game involves control of the character. The principal problem is that movements corresponding to left/right and up/down on the grid don't correspond to equivalent movements on the screen. Various solutions address this problem. One is to use a point-and-click interface. With a point-and-click interface, you instruct a character to move to a particular spot by clicking on the spot on which the character is to move. Another solution entails using diagonal keys, as in the classic *Qbert*.

A more fundamental problem arises in situations in which the character can move in dynamic ways. Consider a character that flies over an isometric landscape. If you are working with 3-D, no immediate way exists to distinguish between an object near the camera but high up and an object further away but low down. Because the world is isometric, you can't even use relative size to help you out. Different solutions to this problem have been presented. An early approach involved using a shadow. The shadow showed the flying character's position relative to the tile plane.

Platform Games in 3-D

No rule exists requiring you to use isometric 3-D. Mainly, it is a tradition. The same tile-based technique works just as well for any 3-D world. The original *Tomb Raider*, for example, clearly shows its origin in tiles. Most of the world is made up of square-based blocks of various sizes and shapes. In fact, there was little essential difference between the original *Tomb Raider* and *Super Mario World*. At its heart, each was a simple game in which you had to find your way to a goal in a tile-based environment, dodging or defeating enemies and collecting items along the way.

Note

Not all 3-D games are tile-based. The *Doom* and *Quake* engines, for example, employ an extrusion system. In this system, walls are drawn to follow lines on a 2-D map.

Many of the issues associated with 3-D isometric games come down to camera control, and camera control is a topic reserved for Chapter 24. Short of camera control, you face the same issues that surfaced in the discussion of TBG games. With respect to these issues, the techniques already discussed still apply. However, the techniques can be enhanced due to the options afforded by the added dimension. Consider, for example, what happens to tiles. Since they no longer have to conform to the isometric grid, with 3-D worlds, tiling can involve more interesting shapes. Among other things, they can have slanted tops. Additionally, you can also apply textures and lights to the surface independently of the tile shape, which gives you much more leeway for design. Collision and visual elements of the tile are separated.

With respect to techniques for collision detection, as with the complex tiles discussed in the previous section, 3-D provides some interesting options. Drawing in part from previous explorations, to consider two such options, you can define tiles with a height map or you can use a vector description such as “sloping down from left to right at an angle of 30° from a height of 50.” If your tiles are reasonably small, using the vector description is almost certainly a better choice. Among other things, this approach leaves you plenty of leeway to create rich surfaces, especially with the addition of textures and bump maps.

To apply some of these observations to code, the `box3DTileTopCollision()` function makes use of the vector approach to test for collisions between a 3-D AABB and the top of a single tile. To construct this function, you make several assumptions. One is that the tile has a flat top whose normal is known. Two more assumptions are the height of the center point and that the tile has a size of 16. A final assumption is that other faces are all aligned to the coordinate axes. The function is a long one, and Exercise 23.1 challenges you to make it so that the function detects collisions with an increased number of faces.

```
function box3DTileTopCollision(center, width, length,
                                height, displacement,
                                tileCenter, tileHeight, tileNormal)
  // width, length and height are the
  // half-lengths of the sides of the box
  // calculate tile top edges
  set edge1 to crossProduct(tileNormal,
                            (1,0,0)) // edge vector parallel to z-axis
  set edge2 to crossProduct(tileNormal,
                            (0,0,1)) // edge vector parallel to x-axis
  multiply edge1 by 16/edge1[3]
  multiply edge2 by 16/edge2[1]
  // find vertices
  set c to tileCenter+(0,tileHeight,0)
  set v1 to c-edge1/2-edge2/2
  set v2 to v1+edge1
  set v3 to v2+edge2
  set v4 to v1+edge2

  set t to 2
```

```
// find collision time with base of box
if displacement[2]<0 then // box is going down
    set planec to center+(0,-height,0) // start height of base
    set vtop to the one of v1, v2,
                    v3, v4 with the highest y-coordinate
    set t1 to (vtop[2]-planec[2])/displacement[2]
// possible collision during time period
if t1<1 and t1>=0 then
    // check for intersection within box base
    set p to vtop[2]-t1*displacement-planec
    // vector from box center to intersection point
    if abs(p[1])<width and abs(p[3])<length then
        set t to t1 // top vertex collides
    end if
end if
// NB: if there is a collision with this vertex,
// it has to be the first collision
end if

if t=2 then // try other collisions with base
    if dotProduct(displacement, tileNormal)<0 then
        // find the vertex of the box that
        // would collide with the tile face
        set basevertex to planec
        if tileNormal[1]<0 then add (width,0,0) to basevertex
        otherwise add (-width,0,0) to basevertex
        if tileNormal[3]<0 then add (0,0,length) to basevertex
        otherwise add (0,0,-length) to basevertex
        // basevertex is the leading vertex
        // on the box with respect to the tile top

        set t2 to dotProduct(basevertex-c, tileNormal) /
                    dotProduct(displacement, tileNormal)
        // leading vertex intersects tile plane
        if t2<1 and t2>=0 then
            // check for intersection within tile
            set p to basevertex +t2*displacement-c
            if abs(p[1])<8 and abs(p[3])<8 then
                set t to t2 // leading vertex collides
```

```

        end if
    end if
end if
// NB: again, this collision will always
// come before any other potential collision
end if

if t=2 then // try edge-to-edge collision
    set n1 to crossProduct((1,0,0), norm(displacement))
    // n1 is the normal of the plane swept
    // out by x-aligned edge of box
    set s1 to dotProduct(basevertex-vtop,n1)
    if s1>0 then
        set s1 to -s1
        set n1 to -n1
    end if
    // check for intersection with z-aligned axis of tile
    divide s1 by dotProduct(edge1,n1)
    if dotProduct(c-vtop,edge1)<0 then set s to -s1
    otherwise set s to s1

    if s>=0 and s<1 then
        set p1 to vtop+s1*edge1 // intersection point with edge
        set t3 to magnitude(p1-basevertex)/ magnitude(displacement)
        if t3<1 and t3>=0 then set t to t3
    end if
    || repeat for other edge pair
end if
if t<1 and t>=0 then return t
end function

```

This function takes advantage of several optimizations that result from the alignment of the boxes. For example, since the edges of both boxes are aligned with the axes, as shown in Figure 23.3, only two possible edge-edge collisions can occur. In this respect, note that the vertices marked *vt* and *bv* correspond to the variables *vtop* and *basevertex* in the code for the *box3DTileTopCollision()* function.

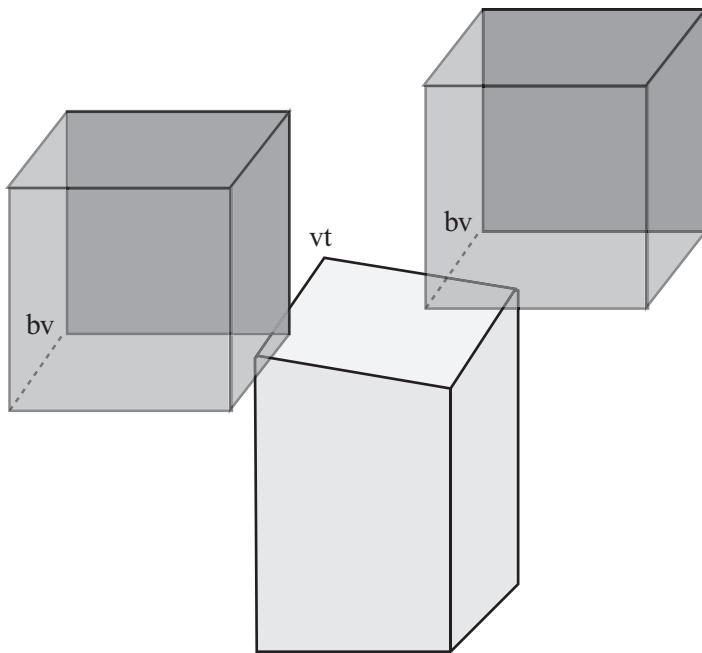


Figure 23.3

Possible edge-edge collisions with a 3-D tile.

Spline-Based Tiles

A final example of a TBG is a game set on one or more curving paths. Since this is a complex scenario, it is not possible discuss it at great depth. However, a few central points can be made. For one, although nominally constructed in 3-D environments, this game is limited to a single path that curves through the game world, with a small range of movement in either the *x* or *y* directions.

As it is, you can't be sure how particular games are programmed, but it's interesting to speculate how a game using spline-based tiles might be created. One way involves describing the main path in terms of a 3-D spline, which is then mapped to a simple 2-D tiled map. To review the context of this notion, see Figure 21.2.

When a game using spline-based tiles is loaded, you translate the 2-D map into 3-D, just as you do with a normal platform game, but you also curve the map to fit onto the spline. Such methods can be used to create racing games. You transform a straight line into a curved track, either on a plane or curving through space as in more space-age racing games.

The most useful aspect of using spline-based tiles is that you can perform all your collision detection routines in the simple tile space, using actual 3-D space for display purposes only. However, working this way requires more than a little fiddling to achieve effects that are robust and feel natural. This is especially so when you deal with speeds along different curves of the track. In the end, however, the approach offers many payoffs.

Exercise

EXERCISE 23.1

Complete the `box3DTileTopCollision()` function and extend it to take into account collisions with the other faces of the tile. The version given in the chapter leaves some of the work unfinished. See if you can complete it. Then finish it off by doing some of the easier work of detecting collisions with the other faces. Note that there are only two other possible face-face collisions. Attending to them should be reasonably straightforward, but don't forget that the top edge is not parallel to the ground.

Summary

This chapter focuses more on programming techniques than mathematics. Much of the programming is broadly discussed, but the topics presented are important in gaming and deserve some discussion at even the most cursory level. This material also provides you with a good introduction to Chapter 24, which explores the quintessential TBG, the grid-based maze.

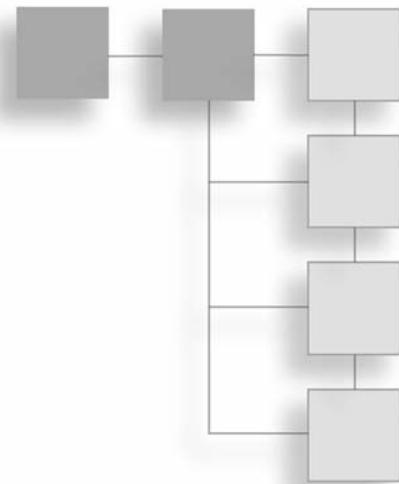
You Should Now Know

- How to create and draw a 2-D *tile-based game*
- How to scroll through it either by copying from a *pre-rendered image* or drawing on the fly
- How to control the camera to create natural movement
- How to calculate collisions in a 2-D game or a 3-D game with *vector-based tiles*
- How to create a game based on a *spline* by mapping from a simpler 2-D description

This page intentionally left blank

CHAPTER 24

MAZES



In This Chapter

- Overview
- Classifying Mazes
- Creating Mazes
- Navigating Within Mazes

Overview

A particularly common type of game world can be loosely described as a *maze*. The variety of games that fit into this category in one way or another is large. Consider such games as the classical *Pac-Man*, which conspicuously uses a maze, to sophisticated driving games that unfold within a vast city grid. Even tile-based games and exploratory 3-D platform games make use of mazes. In this chapter, you'll look at how to create, work with, and navigate through mazes. You will also explore how to create simple artificial intelligence (AI) routines that search through the maze for a quarry.

Classifying Mazes

Before you look at how to work with mazes, it's helpful to understand what the term *maze* formally means and how different mazes can be classified. There are two general ways to think of a maze. One is to think of its physical properties, which include the shape of the underlying grid (rectangular, triangular, circular, and so on), how convoluted the paths of

the maze are, and the dimensionality of the maze. The other way is to think about a maze that involves giving attention to its *topology*. Formally, topology is a fairly important area of mathematics that studies objects as a set of properties that are preserved as the objects are *deformed* (twisted, warped, bent, and so on). Topology allows you to examine the mathematical form of the maze stripped of its physical factors. Using topology, you can examine a maze as a set of branching points joined by paths.

Graphs and Connectivity

Mathematically, a maze is a kind of *graph*. However, the term *graph* is used in a different sense here than you use it elsewhere. As used in this context, a graph is a diagram consisting of a number of points joined by lines, or *edges*, as shown in Figure 24.1. The vertices formed by the edges are referred to as *nodes*. Observe in Figure 24.1 that the maze on the left has been converted to a graph on the right. In the graph, each node represents a fork, crossroads, or dead-end in the maze, and each edge represents a path that can be taken from one node to another. There are also two special points, those at the start and the finish. To make things simpler, these are given their own nodes (S and F), although they could just as easily be placed at the nearest branching point or dead end. Although a maze isn't required to have a start and finish, it usually does.

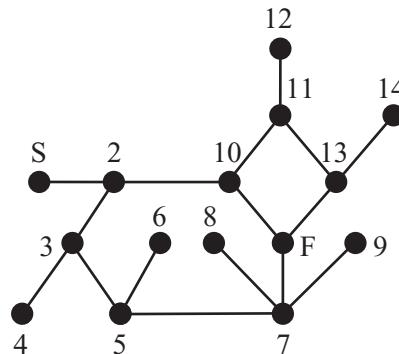
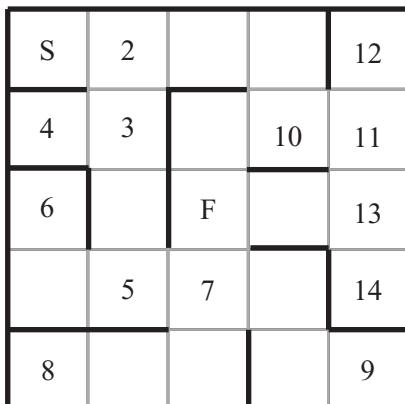


Figure 24.1

A maze and its associated graph.

A graph is, in most cases, considered independently of how it's drawn on the page. When you examine a graph mathematically, you are interested only in its topological properties. In other words, you are concerned about which points are connected to which. However, it is possible to provide a *label* for each of the edges of a graph. The label can show a number that represents the length of the edge, among other things. You can also label the vertices (nodes) with values that represent some kind of cost or distance. A graph labeled in this way is known as a *network*. Alternatively, if you do not want to define a formal network, the edge and node labeling can simply be a way to index the graph for reference.

Conventional *true graphs* can have at most one edge between any pair of vertices, and no vertex may be joined to itself. However, such strictures don't apply to mazes. In light of this, you need to consider what are sometimes called *pseudo-graphs*. Mazes sometimes translate to pseudo-graphs. It's possible to convert any pseudo-graph to a true graph simply by removing offending edges, with no effect on the main properties of the maze, especially with respect to finding a path from one point to another. Since a graph or network is affected by the different choices of edge, it's important to allow multiple connections when searching for optimal paths. In such cases, you can convert a pseudo-graph to a true graph by inserting additional vertices, which in a network are connected by edges of length zero.

There are a number of other ways to classify graphs that apply naturally to mazes. Figure 24.2 illustrates the different classifications. The following list reviews the classifications illustrated in Figure 24.2:

- A graph is said to be *connected* if there is a path from each node to every other node. Figure 24.2a is connected, while Figure 24.2b is *disconnected*.
- If there is exactly one path from any node to any other node, the graph is called a *tree*. Any connected graph with n nodes and $n - 1$ edges is a tree. Figure 24.2c is a tree.
- A graph is called *planar* if there is some way to draw it on paper so that no two edges cross over. Figure 24.2d shows how the graph in Figure 24.2a can be drawn in this way, proving that it is planar. All trees are planar.

A maze whose graph is a tree is defined as *simply connected*. Unless you count backtracking, such a maze has exactly one correct solution. If the graph is connected but not a tree, then the maze is *multiply connected*. Such a maze contains loops. If the graph is not connected, then there is some set of nodes that can't be reached from some other set. In other words, the graph can be divided into a number of connected *subgraphs* that are not connected to one another. For a maze, this means either that it has no solution (if the start and end nodes are in different regions) or that some part of the maze is irrelevant since it can't be reached.

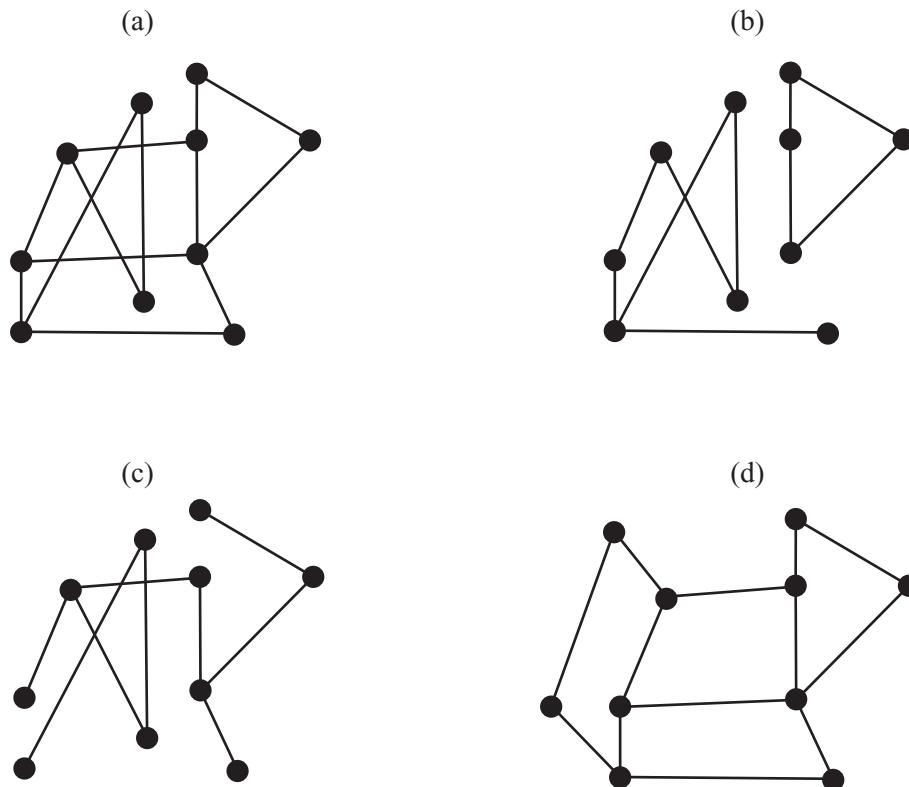


Figure 24.2
Examples of connectivity and planarity.

Graph theory is a large and complex mathematical topic, and it is not possible to explore it in this context beyond what has already been presented. However, you'll encounter a few graph theory issues in the next two chapters when you look at search strategies.

Twists and Turns

Although the topology of a maze is the only thing apparently affecting finding a path through it, in reality the physical properties of a maze are also significant. After all, real-life labyrinths and traditional hedge mazes are not generally based on particularly complicated graphs. The classical labyrinth that contained the Minotaur did not have any branching points at all. Instead, it was simply a convoluted path to the center, like the decorative labyrinths used in many ancient mosaics. This understanding is generally considered to underlie the technical distinction between a maze and a labyrinth.

Psychological factors complicate a maze. A game that features a maze that the player looks down on depends for optical complexity on the player's tendency to immediately seek the goal. If the game is a first-person maze, the difficulty of remembering sequences of twists and turns plays heavily into the success of the player.

Given the extent to which psychological factors are predicated on physical factors, it is useful to provide an inventory of the physical features of a maze. Here is a list of some of the most salient physical features.

- **Dimensionality.** Does the maze have one, two, three or more dimensions? You're most used to mazes of two dimensions (2-D), but it's perfectly possible to create a maze in a cube. Or you might create a sequence of cubes with portals between them. You can also create a kind of "2.5-D" maze by allowing bridges, tunnels, or teleports to connect different parts of a 2-D maze. More complicated mazes can be created by allowing the maze itself to mutate over time, as in the movie *Cube* (1997).
- **Geometry.** A 2-D maze can be based on a plane, but it can also be allowed to wrap in various directions, as with the *Pac-Man* maze, which wraps from left to right as if it is drawn on a cylindrical surface. Such behavior characterizes the geometry of the maze. The geometry of a 2.5-D maze can be described as a 2-D maze that curves through 3-D space.
- **Underlying grid.** A maze can be based on a grid of squares, as in Figure 24.1. It can also be based on triangles, hexagons, or concentric circles. All of these are grids. The grid can be distorted in various ways, such as when perspective transforms are applied. A maze can also have no underlying grid at all. In this case, it can be characterized as a bunch of walls running in various directions.
- **Texture.** Texture is characterized in several ways. One approach is how far you normally travel before encountering a branching point. Another approach involves how far you travel before encountering a (non-branching) turn in a corridor. A third approach involves the proportion of dead ends or, equivalently, the average length of a dead-end passage. Different maze-generation methods tend to produce mazes of different textures.

Of the salient physical features of a maze, texture is the most subtle. You can explore texture in depth by applying terminology specific to textures. Much of this terminology originated with Walter Pullen, creator of a freeware maze software package called *Daedalus*. (For more information, see Appendix D.)

Figure 24.3 shows two mazes with different degrees of what, in relation to texture, Pullen calls *river*. The maze on the left consists of one long passageway with a few very short dead-end passageways leading off it. Such a maze is described as having a *low river*. In contrast, the maze on the right is a *high-river* maze. With this maze, while fewer dead ends occur, the dead ends are longer and more convoluted. Notice, however, that in both mazes the longest path is much the same. You can assign a numerical value to the river by calculating the percentage of dead ends in the maze. In terms of determining how difficult it is to solve a maze, low-river mazes are most difficult. High-river mazes are easier to solve because they offer fewer choices.

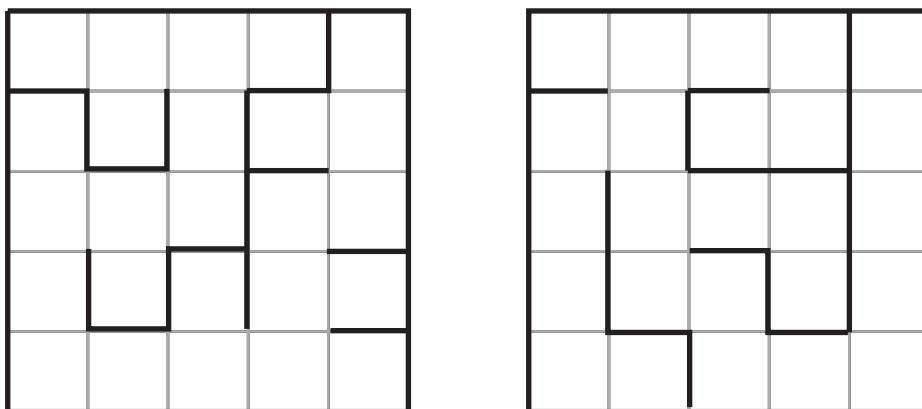


Figure 24.3

A low-river maze (left) and high-river maze (right).

Another term related to texture is *convolution*. The convolution of a maze can be measured by the *standard deviation* of the length of a path from any end-node to any other end-node of the underlying tree. Standard deviation is a common statistical tool that measures the average distance from the *mean*. If the mean of a set of numbers x_1, x_2, \dots, x_n is μ , then the standard deviation σ is the square root of the mean squared difference of each number from μ . In other words, the mean is the square root of the variance (which was discussed in Chapter 10). The formula for standard deviation is as follows:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2}$$

While an extended discussion of statistics lies beyond the scope of this chapter, with relation to the topic at hand, the higher the standard deviation, the lower the value of convolution. The highest convolution is be obtained by a maze that looks like a spider, with a number of paths of equal length radiating out from a central branch point. In this case, the standard deviation of the path length is zero. An alternative measure is the mean distance from one node in the graph to another. A maze with higher convolution tends to be easier to solve although the path to the goal is usually longer.

Figure 24.4 illustrates two mazes. The mazes possess the same degree of river, but different degrees of *run*. Run is a measure of the *twistiness* of the maze. Twistiness is an indicator of how far you can travel in a straight line. The run can also be biased in particular directions, resulting in a maze that has a long run in one direction and a short run in another. A maze based on concentric squares, for example, has differently biased run values at different points.

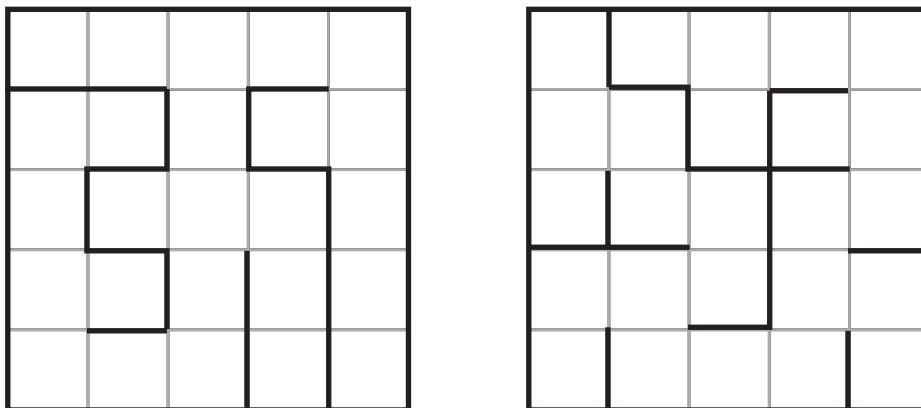


Figure 24.4
Mazes with different degrees of run.

Creating Mazes

Now that you have a general understanding of how mazes can be discussed, you can take a look at how a maze might be handled on a computer and some techniques for generating a maze at random. To approach such topics, it is best to work with mazes based on a grid. Such mazes are typically generated automatically. Most first-person shooters can be considered mazes of this kind. In this respect, they contrast with non-grid-based mazes. Non-grid-based mazes are rarely generated automatically. Instead they are created by hand using level editors and navigated using the collision detection techniques discussed in previous chapters.

Handling Maze Data

A tile-based maze is based on a grid. Such a maze can be defined as a list of cells with walls between them. For example, a square grid consists of an $n \times m$ array of cells, and each cell can have a wall to the north and to the east. To determine if there is a wall to the west of cell (i, j) , you look to see if there is an east wall in cell $(i - 1, j)$. A similar technique is used to determine if there is a south wall. You might recall that in Chapter 2 you set up a function for initializing a grid of this kind using the `modulo()` function.

What applies to a square grid can be applied to other grids. One in particular is the triangular grid. Figure 24.5 illustrates a triangular grid. Grids of this type are best considered as two interlocking sets of cells with different properties or one set of rhombuses (equivalent to squares) that might or might not be split in half.

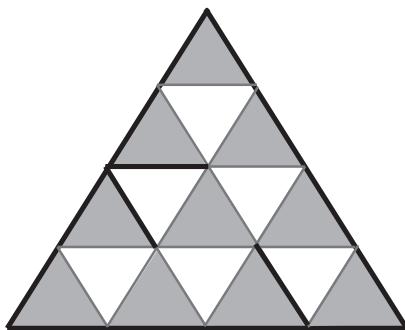


Figure 24.5

Storing maze data in a triangular grid.

Automatically Generating a Maze

There are quite a few standard algorithms for generating mazes. In general, generating a maze can be done either by growing the walls out from the border or by starting from a (tile-based) maze with all the walls in place and removing walls until the maze is complete. You'll only consider the first kind here, since it's more convenient when the maze is set up as suggested in previous sections.

The simplest method for generating a maze is called the *recursive backtracker*. In this algorithm, you maintain a memory of the path you have taken, and at each stage you look from a neighboring cell to the current cell to see if it has yet been visited. If it has been visited, you try another cell. If it has not been visited, you remove the wall and move into the new cell. If all cells have been visited, you backtrack along your path until you find another cell with an unvisited neighbor. If you get back to the first cell with no more

neighbors to reach, then the maze is complete. The `recursiveBacktrack()` function uses this approach:

```

function recursiveBacktrack(maze, startcell, endcell, path)
    if path is empty then add startcell to path
    set currentcell to the last cell in path
    set neighborList to the neighbors of currentcell in maze
    randomize neighborList
    repeat for each cell in neighborList
        if cell is not in path then
            set found to 1
            add cell to path
            remove wall between cell and currentCell in maze
            recursiveBacktrack(maze, startcell, endcell, path)
        end if
    end repeat
end function

```

A slight variation on the recursive backtracker method is *Prim's Algorithm*, which instead of backtracking, picks a random unvisited neighbor of any cell on the path. This algorithm requires more computational effort, since you must maintain a complete list of all unvisited neighbors. However, it runs somewhat faster and produces a maze with more dead ends. Both the recursive backtracker algorithm and Prim's Algorithm can be used to provide flood-fill for a paint program. This is so because they completely explore any space designated for them. For this reason, both are convenient methods for creating mazes of irregular shapes or working with any underlying grid. Figure 24.6 shows two mazes created using these two algorithms.

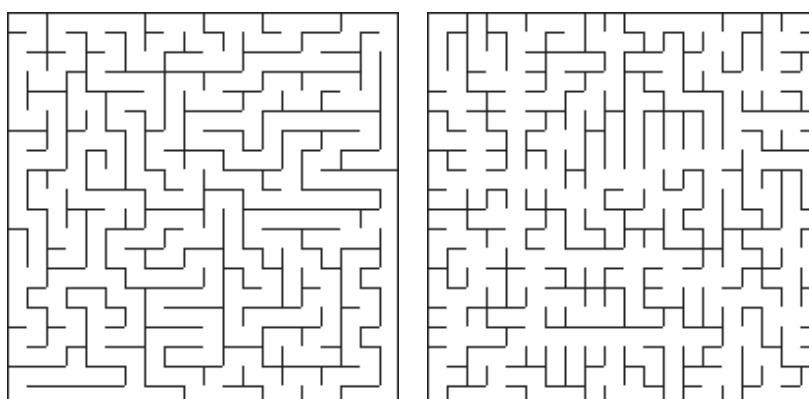


Figure 24.6

Mazes created using recursive backtracking (left) and Prim's Algorithm (right).

A third algorithm is called Kruskal's Algorithm. This algorithm works differently from the two already reviewed. Instead of working with a path, it starts by numbering each cell in the maze. Then it picks walls at random. If the cells on each side have different numbers, it removes the wall and renames all the walls on each side with the same number. This activity gradually builds connected regions in the maze, until all cells have been numbered the same. At this point, the maze is complete. Prim's and Kruskal's Algorithms are examples of what is called a *genetic algorithm*, which you'll encounter again in Chapter 26. The `kruskal()` function encapsulates the actions just described:

```
function kruskal(maze)
    set wallList to a list of all walls in maze
    randomize wallList
    set idList to an empty array
    repeat for i=1 to the number of cells in maze
        id maze[i] with i
        add [i] to idList
    end repeat
    repeat for wall in wallList
        set cell1 and cell2 to the neighbors of wall
        set id1 to cell1's id
        set id2 to cell2's id
        if id1 <> id2 then
            remove wall from maze
            repeat for each cell in idList[id2]
                add cell to idList[id1]
                id cell with id1
            end repeat
            set idList[id2] to an empty array
        end if
    end repeat
end function
```

The `kruskal()` function as presented here can be optimized, but even with optimization, it is harder work for the computer than the `recursiveBacktrack()` function. Still, it works well and is suitable for a grid of any type or shape.

A fourth function is based on *Eller's Algorithm*. Eller's Algorithm has a few minor disadvantages when compared to those already discussed. Two issues are that it works only for rectangular mazes and tends to have a bias in one direction. Still it's quick and memory-efficient. With this method, you work through the whole maze a row at a time, using an

approach similar to Kruskal's Algorithm. Each cell in a particular row is given an ID number according to how it is connected to other cells in previous rows. There must be at least one path from the previous to the current row for each remaining ID number, and you can have at most one path with the same ID from the previous row to any connected set in the new row. This is to avoid creating a loop. Generally, implementing Eller's Algorithm is a more complex task than those discussed previously. Notice that you start with a maze whose walls are all down. The `eller()` function encapsulates Eller's Algorithm:

```
function eller(maze, hfactor)
    set w to the length of a maze row
    set h to the number of rows

    // create first row
    set currentRow to ellerRow(maze, w, 1, 1, hfactor)

    // create body of maze
    repeat for j=2 to h
        set newRow to ellerRow(maze, w, j, w*(j-1)+1, hfactor)
        set testList to a list of elements from 1 to w
        randomize testList
        repeat for each i in testList
            set id1 to newRow[i]
            set id2 to currentRow[i]
            if id1 <> id2 then
                repeat for each element in newRow
                    if element=id1 then set element to id2
                end repeat

                repeat for each element in currentRow
                    if element=id1 then set element to id2
                end repeat
            otherwise
                add a wall between cell (i, j) and (i, j-1) in maze
            end if
        end repeat
        set currentRow to newRow
    end repeat
```

```
// adjust final row to ensure span
set idlist to an empty array
repeat for i=1 to w
    if currentRow[i] is not in idList then
        add currentRow[i] to idList
        remove the wall between cell(i, h) and (i-1, h) in maze
    end if
end repeat
end function

function ellerRow (maze, w, row, id, hfactor)
    set r to an empty array
    repeat for i=1 to w-1
        add id to r
        if random(1000)<hfactor then
            add 1 to id
            add a wall between cell (i, row) and (i+1, row) in maze
        end if
    end repeat
    add id to r
    return r
end function
```

Figure 24.7 shows two mazes generated by an implementation of the `eller()` function using different values assigned to the `hfactor` variable. A number from 1 to 1000, representing probability, should be assigned to this variable. As Figure 24.7 reveals, this value affects the run factor in each direction and needs to be approximately 50% to create a successful maze.

The greatest advantage of Eller's Algorithm is that it can be used for a maze of any size in a particular direction. Another advantage is that you don't need to worry about memory requirements because only one row of the maze at a time needs to be held in memory. It can also be very easily adapted to create a 3-D maze, working one layer at a time instead of one row at a time.

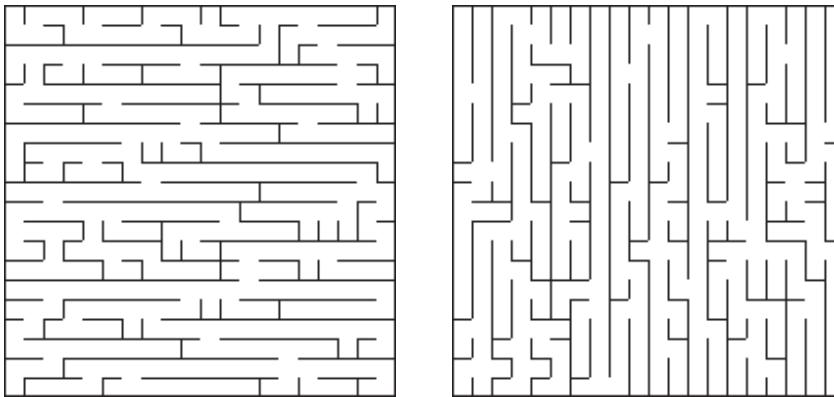


Figure 24.7

Adjusting the random factor for horizontal walls in Eller's Algorithm (left: 20%, right: 80%).

Multiply-Connected Mazes

Strange as it may seem, a maze does not become easier to solve by removing dead ends. In fact, it generally becomes significantly harder to solve, not so much computationally as psychologically. Computationally, the faster algorithms work much the same way whether or not the maze has loops, although the simpler algorithms such as wall-following may fail. With a sufficiently large maze, any algorithm based on storing the whole path is going to have memory problems.

Psychologically, the existence of loops means that it is much easier to get lost. If this is a desired property of a maze, however, it invites difficulties, for loops make creating a random multiply-connected maze a difficult proposition. With loops, it is hard to create a random algorithm. Random algorithms, however, generate psychologically interesting mazes. What arises, then, is a tradeoff.

The simplest way to generate a multiply-connected maze is to create a variant of the recursive backtracker algorithm. Using this approach, at the end of each pathway, you open up a wall back into the existing path. While this works well, it also creates mazes with primitive loops. You end up with four cells in a square, for example, or a 3×2 group with a single wall in the center.

An alternative method that is much more computationally expensive but more effective begins with a simply-connected maze and then removes walls. To remove the walls, you first choose a few walls at random. For each chosen wall, you calculate the length of the shortest path from one side of the wall to the other. To create a maze with long loops, you can remove the wall with the longest path. For shorter loops, you choose the median path—the one in the middle of the sample. This approach does not lead to a complete multiply-connected maze with no dead ends (a braid maze), but it does produce mazes that are generally interesting. The `multiplyConnected()` function uses this approach to generating mazes:

```
function multiplyConnected(maze, connections)
    // assume maze is already created and simply connected
    repeat with i=1 to connections
        set wallList to an empty array
        repeat for 11 walls in maze
            set s1, s2 to squares on either side of the wall
            set dist to path distance from s1 to s2
            add wall to wallList, sorted by dist
        end repeat
        remove the last wall in wallList

    end repeat
    return maze
end function
```

For multiply-connected mazes even more than for simply-connected mazes, human-generated patterns tend to be better than anything produced by a random algorithm. Random processes are best used for filling the gaps around the principal areas of the maze with additional convolutions and blind alleys.

More Complex Mazes

Although the grid-based maze is the most common, other maze types can also be used. Topologically, there is no difference between a maze based on a grid and one based on another substructure or no structure at all, but the difference does have implications both psychologically and in terms of computation.

One common maze type that is not grid-based is the *circular maze*, represented in Figure 24.8. Topologically, this maze is identical to the maze in Figure 24.1, but its substructure is very different.

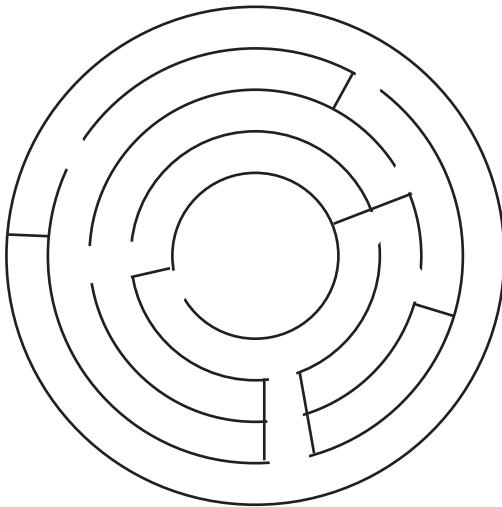


Figure 24.8

A maze based on concentric circles.

Creating a maze of the form shown in Figure 24.8 can be done in a number of ways, but a particularly effective method entails using a variant on Eller's Algorithm. With this variant, instead of working row by row, you work circle by circle from the inside out. To work in circles instead of rows is simple. The only change is that regions now wrap from one side of the row to the other. The process is equivalent to creating a maze on a cylinder.

Having created a cylindrical maze, you give it an exit at the top and another at the bottom, and you map the whole thing to a circle. As shown in Figure 24.9, the exit at the bottom leads to the center of the maze, and the exit at the top leads to the outside.

A maze such as the one illustrated by Figure 24.9 won't be entirely regular in the sense that the walls on the inner circles are more closely packed than the walls on the outer circles. The reason for this is that there are more grid cells in a smaller space. You can offset this problem by varying the horizontal wall density corresponding to the `hfactor` variable in the previous discussion. Increasing the value assigned to this variable can make more walls as you travel outward.

One further consideration when creating these mazes is how to draw them on the computer. Such drawing is much harder to effectively accomplish than drawing involving a grid of straight lines. In general, making rounded mazes look good requires tweaking by hand. Still, that the automated approach can be effective is demonstrated by the maze in Figure 24.10, which was created using a computer.

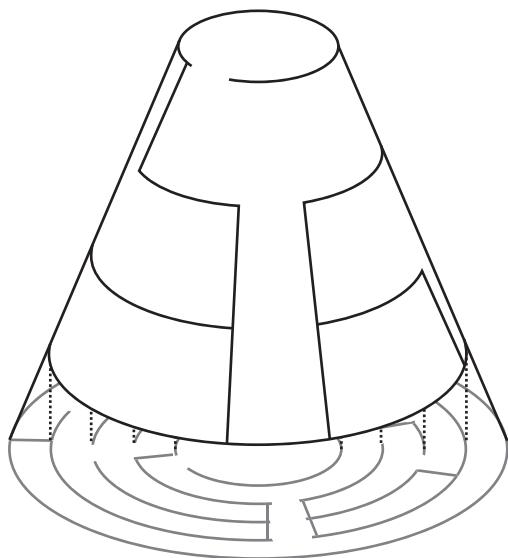


Figure 24.9

Converting a cylindrical maze to a circular one.

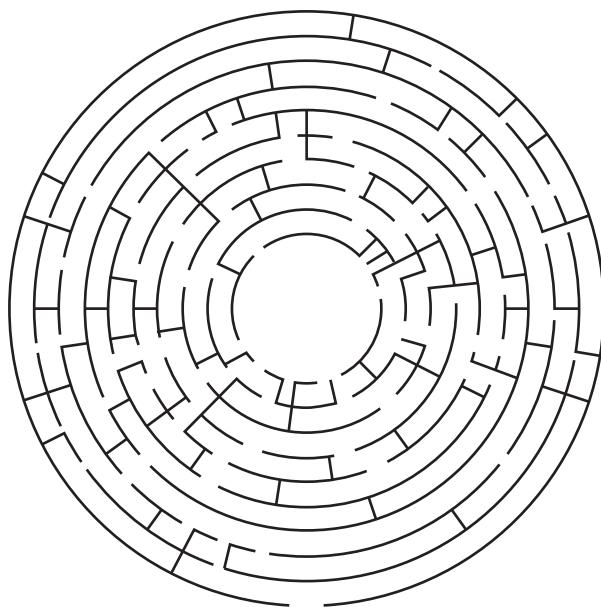


Figure 24.10

A circular maze generated by a variant of Eller's Algorithm.

Navigating Within Mazes

Having generated a maze, you must then move through it. To move through a maze, the size of your character becomes important. The character must be small enough to move through the maze. This point in many respects is a primary criteria for distinguishing a maze game from the tile-based games examined in the previous chapter.

The simplest kind of maze movement involves stepping from one cell to another, but for most applications, stepping from one cell to another is not enough. The character must be able to move smoothly and navigate around in ways that are not merely seen as steps from one cell to another. In addition, navigation requires that you be able to move in specific ways, from A to B. This, after all, is the primary motivation for implementing the maze.

Collision Detection and Camera Control

The main trick to maze navigation requires that you consider each square as a separate room. The situation in many ways resembles working with quadtrees. The only difference is that you have the added advantage that, rather than being arbitrary, the trees constitute an essential part of the landscape. As long as a character is moving within a room, you know that collision is occurring. When the character leaves the room, all you need to do is find which room it is trying to enter and check if a wall is in the way. It couldn't be simpler.

The details of constructing the code for this section are left as an extended exercise, but one issue that arises is worth mentioning here. This issue concerns a convention of navigation that has become standard in gaming. This convention is referred to as *wall-sliding*. With wall-sliding, when following a path that naturally takes you into a wall, rather than either stopping dead or rebounding, you remove the normal component of motion but retain the tangential component. The result is that you appear to approach the wall and then slide along it. It's a very strange concept, but it feels quite natural and is computationally simple.

When working with wall-sliding scenarios in a first-person 3-D view, you must keep the camera a little away from the wall. Otherwise, you'll get partial views of the room next door. To keep the camera away from the wall, it's best to treat the observer as a sphere rather than a point.

In a third-person 3-D view, things get a little more complicated because you have different sets of collisions to consider. Imagine a situation in which you have worked out where the character is but still need to determine where the camera will go. Generally, you consider the camera to have a natural resting point, usually somewhere behind and a little above the character. If possible, when the character moves, you want the camera to

follow, but often this isn't possible, as shown in Figure 24.11. As shown in this figure, the character at A has turned, so the camera that was previously at B is now likely to be at C, which is, unfortunately, inside a wall.

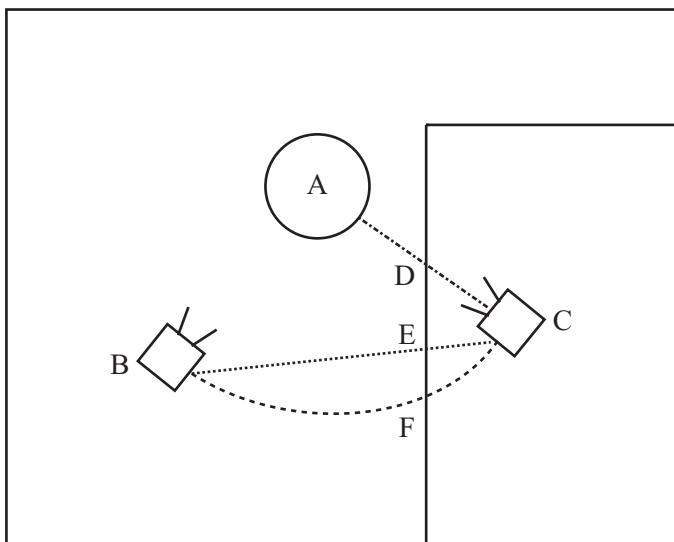


Figure 24.11

A camera movement hitting problems with a wall.

To deal with the situation shown in Figure 24.11, you can treat the camera as another observer following its own path and deal with its collisions exactly as you do with the first observer. But you still have to decide what happens when the collision occurs. The best solution might be for the camera to stop at the point D, the nearest available point along the line AC. This might be a little too close to the observer, however, and you must then lift the camera up into the air so that it looks slightly downward at the observer.

Two alternative positions are the points E and F. With E, the camera first hits the wall on a straight-line path from B to C. On the other hand, F is on a curved path that does not help things. Both of these alternative positions suffer from the same problem. You cannot see everything the character sees.

Meanwhile, for a situation like that shown in Figure 24.12, it would be foolish to take any of these options when there is a perfectly good camera position at C. In this case, you might be better off to cut to C in a jump instead of interpolating the camera from B to C and passing through the wall.

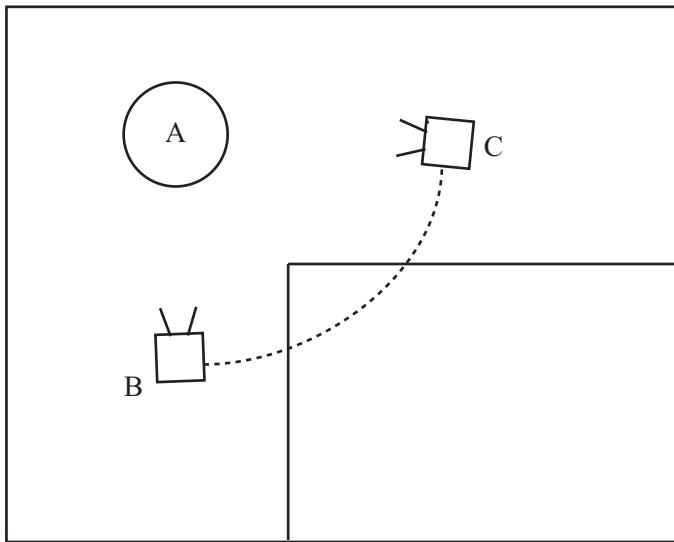


Figure 24.12

A perfectly reasonable camera movement running into problems on the way.

As you've seen before, these decisions about camera control are very much a matter of preference and require careful handling. If in doubt, make sure your user can override whatever unacceptable thing your particular system chooses to do in some unanticipated situation.

Line of Sight

Another reason that grid-based mazes are convenient to work with is that you can calculate relatively easily from any particular point which areas are visible. This, then, is an issue that concerns line of sight. Working with line of sight is useful both for visibility culling in a 3-D FPS and when calculating enemy AI behavior. Picture, for example, a *Pac-Man*-style ghost or a sentry in a stealth game. To explore this topic, square-based mazes are addressed. The techniques discussed can be used with other mazes, as well.

The simplest way to demonstrate a reason for developing a line-of-sight algorithm is to take a look at a diagram, such as the one illustrated by Figure 24.13. Here, a character is situated at the point marked with a cross. All the places the character can see are shaded. Notice that in addition to the cells along the principal rows, a number of neighboring cells are also partially visible.

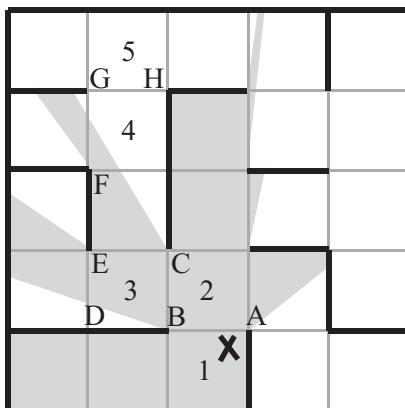


Figure 24.13

Visibility determination from a particular point.

It should be immediately clear that even in a grid, the situation depicted by Figure 24.13 requires calculations that are by no means simple. However, there are some useful shortcuts that you can take, particularly in a simply-connected maze. The trick is to make use of recursion. Each time you pass through a doorway, you’re entering a new simply-connected area illuminated by a beam of light defined by the doorway and the beam of light you began with. In other words, you’re splitting the 360° sweep around the observer into smaller and smaller pieces each time a doorway is passed.

Look at Figure 24.13 again. From the point X, there are four possible doorways through which your imaginary light can shine. Two of them are closed, and two are open. With respect to these, the line AB subtends the largest angle at X. (The two points A and B are said to subtend the angle AXB at X.) So you are now looking at the neighbors of the cell marked 2. You see that all three of the doorways are open. At this point, you recurse over each of these. You can go through the door BC into cell 3, splitting the beam further into the angle BXC. Notice that since the vertex D is outside the illuminated angle, you can ignore it. However, since vertex E is illuminated, some light still gets through the doorway DE—specifically, the beam BXE. You also know already that C is illuminated, which means that the whole beam EXC passes through CE.

You continue recursing in this way until you reach either a blank wall or a doorway such as GH. Cell 4 is partially visible, illuminated by the beam FXC, but both G and H lie outside this beam. This means that none of the light from X passes through GH. What’s more, since the maze is simply-connected, you can be sure that none of the light will reach any cell beyond that point.

The `visibleSquares()` function provides a rough implementation of the line-of-sight algorithm as discussed in this section. One observation is that the function could be much faster, since you're calculating lots of the values several times over:

```
function visibleSquares(observerPoint, beamStartAngle,
                        beamEndAngle, fromSquare, thisSquare)
  if beamStartAngle is not defined,
    then set beamStartAngle to -pi;
         beamEndAngle to pi;
  if thisSquare is not defined then
    set thisSquare to observerPoint's square

  set ret to an empty array
  append thisSquare to ret

  repeat for each neighbor of thisSquare except fromSquare
    if there is a doorway from thisSquare to neighbor then
      set v1 to the first vertex of
        the doorway clockwise from the observer
      set v2 to the other vertex of the doorway
      // find angles of these vertices (in the range -pi, pi)
      set a1 to the angle of v1 with observerPoint
      set a2 to the angle of v2 with observerPoint
      if both a1 and a2 are between beamStartAngle
          and beamEndAngle then
        if a1<a2 then
          // vertices are on the same side of the angle range
          set start to max(a1, beamStartAngle)+0.02 --
            the increment is added to
            rule out 'just visible' squares
          set finish to min(a2, beamEndAngle)-0.02
          // recurse over all visible squares
          set s to visibleSquares(maze, observerPoint,
                                start, finish, thisSquare, n)
          append all elements of s to ret
        otherwise
          // the doorway 'straddles' the angle range
          set start to min(a1, beamEndAngle)+0.02
          set finish to max(a2, beamStartAngle)-0.02
          // split the angle range into two parts
          // and recurse along both paths
```

```
    set s1 to visibleSquares(maze, observerPoint,
                            finish, pi, thisSquare, n)
    set s2 to visibleSquares(maze, observerPoint, -pi,
                            start, thisSquare, n)
    add each element of s1 and s2 to ret

    end if
end repeat
return ret
end function
```

Since this version of the `visibleSquares()` function requires somewhat extended recalculation of values, you might want to think about alternative methods for dealing with the situation in which the doorway straddles the start and end angles. In general, however, there are a number of ways to speed up this algorithm. One is by storing doorway vertex values. Likewise, you might apply Bresenham's Algorithm as presented in Chapter 22 to avoid the need for floats and trigonometry.

As it happens, the algorithm used in the `visibleSquares()` function works just as well for multiply-connected mazes; however, you are likely to find that you are sometimes investigating the same square twice from different directions. In fact, it's more than likely that two parts of the same square might be visible from different sides. The result is that the recursion will often check illumination of particular vertices under several different beams. For a simply-connected maze, the greatest advantage is that as soon as you have determined that a particular cell is out of sight, all passages beyond that cell can be culled from a 3-D scene. This possibility is much less applicable in a multiply-connected maze.

Maze-Threading

When trying to solve a maze, you face two typical scenarios. The first is when you are placed inside the maze at a particular point and need to find your way to the goal or the exit without knowing anything about the maze. The second is when you are looking down at the maze and trying to find a path from one point to another, often the shortest. You'll look at the second situation in the next section. In this section, the focus is on the first scenario, which is sometimes called *threading a maze*. Implementing the algorithm for threading a maze is left to you in Exercise 24.1. The methods that apply to this problem are similar to those you have already encountered when generating mazes.

Exploring the algorithm, threading a maze begins with a classic method for solving a maze that is somewhat strange. It involves turning in the same direction every time you reach a junction. This is analogous to walking along while keeping one hand on the same wall throughout. For any simply-connected maze, this is guaranteed to work. For a multiply-connected maze, it will also work if you are trying to get from an entrance to an exit on the outer walls. However, if there is a loop surrounding the center, it won't enable you to find a way to the center of a multiply-connected maze. Likewise, it will not find the shortest path. Still, it is computationally simple, requiring no memory of the path traveled.

Another method that is faster than the classic method guarantees a solution in even a multiply-connected maze. This is a simple recursive search. It is the equivalent of the recursive backtracker maze generator. It will follow every path as far as it will go, backtracking and trying alternative paths each time you reach a dead end or a cell you have already visited. It can even be used *blind*. *Blind* refers to walking in particular directions and banging into walls. A wall can be viewed as a dead end of length 0.

If you know how to find a path to a goal, then it's possible to improve on the recursive backtracker approach by preferentially trying to form a path that divides the maze into smaller regions. You might make a path that splits the maze in half. You then know that there is no need to try any path on the wrong side of the maze. This can significantly reduce your search time. This can sometimes also mean that specifically searching for paths that don't lead to the goal can be more useful in the long term.

Another method that has the advantage of being easy to apply is to use a system of marking. With this approach, imagine walking through an actual maze, making marks on the walls to indicate passages that you have explored. In computational terms, this approach requires that you maintain the whole maze grid in working memory, since you have to be able to annotate the cell data in some way. In all other respects, the algorithm is identical to the recursive backtracker.

To implement the marker method, you make a mark at each junction along the paths by which you enter and leave. A single mark behind you means that you are moving forward. When you turn around, you leave a second mark, meaning that the current path has been completely explored and was unsuccessful. To ensure that each path is explored, you preferentially enter a passage with no marks. As with the recursive method, you also treat all loops as dead ends. When moving forward, if you encounter a junction that you have already visited (as indicated by a marked path), then you turn around. When moving backward, you expect to encounter marks, but there should be only one passageway with a single mark, which is the one by which you leave.

As an added bonus with the marker approach, when you reach the goal, you also have a marked path back to the entrance. You can take the passageways marked once. You can think of this algorithm literally as *threading*. The marked paths are like a thread that forms loops that indicate dead ends. When you reach the goal, pulling the end draws all the loops of thread in, leaving a single path back to the start.

Pathfinding and the A* Algorithm

The flip side of the maze-threading problem is the *pathfinding problem*. With the pathfinding problem, you deal with two concerns. On the one hand, you know a given maze in advance. On the other hand, you want to find a path, usually the shortest, from A to B.

There are a number of approaches to the pathfinding problem. One approach finds a path at the same average time as the recursive backtracker, but by the opposite method. Instead of searching an individual path to the full before abandoning it (a *depth-first search* in the terminology of Chapter 26), it searches all paths simultaneously (*a breadth-first search*). The breadth-first method is somewhat equivalent to Prim's Algorithm, spreading along a frontier to fill the maze. At each step, you travel one step farther in all possible directions, creating a frontier consisting of all squares at the same distance from the start point. Each cell remembers the neighbor that spread to it, which means that when you find the solution, you can follow the path back to the start. The start is by definition the shortest possible path.

As effective as the breadth-first method is, there is another. This method combines the best of both depth-first and breadth-first searches. It is referred to as the A* (“A-star”) method. It is implemented in the `aStar()` function, and it is easiest to explain how its algorithm works if you first examine the code. In code for the function, `maze` could be anything, as long as you have some way of describing a distance between two nodes. Here is the code for the function:

```
function aStar(maze, start, goal)
    set pathList to an empty array
    set d to distance from start to goal in maze
    append [d, 0, start] to pathList
    sort pathList on element 1 of paths
    // by estimated distance to goal
    repeat until break
        set path to pathList[1]
        // extend each path to all possible neighbors
        delete pathList[1]
        set currentSquare to the last square of path
        set previousSquare to the second-to-last square of path if any
```

```

repeat for each neighbor of currentSquare except previousSquare
    set p to a copy of path
    // no loops
    if neighbor is not an element of p then
        append neighbor to p
        add 1 to p[2] // length of path
        set p[1] to p[2]+distance(neighbor, goal)
            // distance underestimate
        append p to pathList // retaining the sort on element 1
    end if
end repeat
if pathList is empty then return "No path to goal"
if pathList[1] ends with goal then return pathList[1]
end repeat
end function

```

One of the primary features of the A* Algorithm is that it sorts the found paths according to an *underestimate* of the distance from the goal. In other words, it sorts according to the length of the path added to the linear distance from the last square to the goal. This technique gives precedence to searches approaching the goal. It is somewhat similar to the “collision halo” approach to collision detection discussed in Chapter 10.

As a further point, it should be fairly clear that although the A* Algorithm is the optimal maze-search algorithm known to date, it still does not give the solution very quickly in a well-designed maze in which paths are specifically designed to lead away from the goal. However, when applied to a random maze, it is by far the fastest algorithm, and it is particularly good at finding optimal local paths—paths to a nearby target. Such a path might be to an AI enemy that is trying to follow a moving target.

To examine the A* Algorithm more in detail, you start with the goal square. At each stage, you take the current best path and extend it to all possible neighbors. If you’ve made a loop, you ignore that path. Otherwise, you calculate the underestimate for the new path and add it back to the list of paths, sorting on the underestimate.

What does this sorting on the underestimate entail? Consider the maze shown in Figure 24.14. In this maze, the path marked by a dotted line is found first by A*, since as it begins it is well aligned. However, along the way, it encounters a few twists and turns, and so its length increases while the distance from the goal also increases. At the moment pictured, the algorithm’s underestimate has for the last time increased above that of the initially less promising path marked with a jagged line. The result is that the algorithm’s attention now switches entirely to the new path, which eventually achieves the goal faster, even though there is a perfectly reasonable route to the goal along the original path.

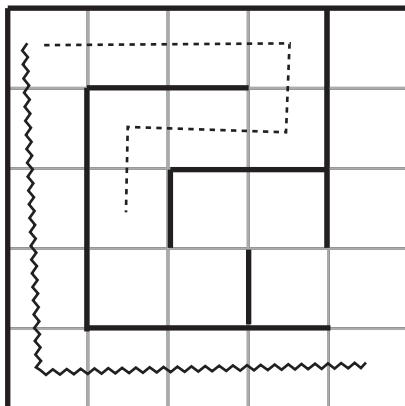


Figure 24.14

The A* Algorithm in action.

One final observation is that with the A* Algorithm, it's perfectly possible for a path to get to the goal but be placed somewhere else in the list because another path of maximal underestimate has come into sight.

Exercise

EXERCISE 24.1

Create a program that navigates through a maze by either wall-following or recursive backtracking. Draw from the algorithms presented in this chapter.

Summary

This chapter has gone into a fairly deep level of detail during certain passages. The details have been justified for at least one reason. While knowing how to generate and navigate mazes are not the most important things to know, they provide a very good introduction to searches, AI, and game theory. These are the topics presented in Chapters 25 and 26, the last two chapters of this book. In Chapter 25, in particular, you'll be making good use of the language of graph theory introduced in this chapter.

You Should Now Know

- How to classify a maze according to its physical or topological properties
- The meaning of certain fundamental graph theory terms such as *node*, *edge*, *tree*, *network*, and *connected*
- How to store the details of a grid-based maze on the computer and use it to walk through and control a camera
- How to generate a maze using a variety of different methods, including *Prim's*, *Kruskal's*, and *Eller's Algorithms*
- How to navigate through a maze either blind or with the whole maze in front of you, especially using the *A* Algorithm*

This page intentionally left blank

CHAPTER 25

GAME THEORY AND ARTIFICIAL INTELLIGENCE



In This Chapter

- Overview
- Introduction to Game Theory
- Tactical AI
- Top-Down AI
- Bottom-UP AI

Overview

This chapter concerns artificial intelligence (AI), and attention is given to how AI is used in board games. AI refers to the study of whether or how machines can be made to “think.” The field is almost as old as computing itself. One of the key moments in the history of the field came when Alan Turing (1912–1954), a mathematician and computer scientist, wrote a seminal paper on the concept of AI. Computers begin to think, Turing contended, when people communicating with computers cannot distinguish the communications of computers from the communications of human beings. This goal has been achieved in only limited ways, if at all. Today, AI is an important part of cognitive science, with researchers trying to create models of different parts of thinking.

AI has always been a fundamental part of game design, but game designers seldom try to create programs that really think. Cognitive scientists deal with what is sometimes referred to as *strong AI*. Strong AI concerns making machines that can think like humans. Game designers usually work with *weak AI*. Weak AI strives to program responses into games that enable games to simulate thinking or respond in realistic ways to selected input. Both areas share a common goal of trying to create sets of rules that produce flexible behavior and sustain some level of learning.

Introduction to Game Theory

In addition to Turing, other significant figures in the history of games and computers are John von Neumann (1903–1957) and Oskar Morgenstern (1902–1977). Von Neumann and Morgenstern, in addition to Émile Borel (1871–1956), are recognized as leading figures in the creation of *game theory*. Game theory is the study of the effects of making choices. A game as defined by game theorists is a context in which one person makes decisions based on what he or she thinks one or more other people are thinking or will do. Questions of this nature have become so significant that in 1994, along with others, John F. Nash, whose life was the topic of the movie *A Beautiful Mind* (2001), received the Nobel Prize for his work in game theory.

Zero-Sum Games

The most basic type of game in the context of game theory is the *zero-sum game*. In this type of game, two or more players compete by making a choice of some kind. With the choice, a certain sum of money is given by the losing player(s) to the winning player(s). The aim is to try to maximize winnings (or minimize losses). The term *zero-sum* means that the total amount of money in the game is constant. Whenever one player gains, another player must lose.

To keep things simple, the games discussed in this chapter will be two-person games. What applies to these simple games applies to games involving more than two players. Figure 25.1 shows a simple example of a two-person zero-sum game. The game is the familiar one of Rock-Paper-Scissors. The game has two players, Andy and Beth. Both players reveal their hands at the same time and have a choice of three options, rock (R), paper (P), or scissors (S). If they match, then no money passes hands. If they do not match, the winner is determined by the system $R > S > P > R$, where $>$ is used to stand for “beats.” In Figure 25.1, a matrix represents the outcomes. In this figure, the three rows correspond to the three possible actions of Beth. The three columns relate the three possible actions of Andy. The values given in the cells of the table are the payoffs of the players with relation to the profile of possible actions.

		Andy		
		R	P	S
Beth	R	0	-1	1
	P	1	0	-1
	S	-1	1	0

Figure 25.1

A two-person zero-sum game.

Another example of a game is shown in Figure 25.2. This is a game called Undercut. It is examined by the mathematician Douglas Hofstadter in a book titled *Metamagical Themas: Questing for the Essence of Mind and Pattern* (Basic Books, 1985). With this game, Andy and Beth have to choose a number from 1 to 5, and the player who chooses the higher number wins the difference of the two numbers unless the numbers differ by one, in which case the other player wins the sum.

		Andy				
		1	2	3	4	5
Beth	1	0	3	-2	-3	-4
	2	-3	0	5	-2	-3
	3	2	-5	0	7	-2
	4	3	2	-7	0	9
	5	4	3	2	-9	0

Figure 25.2

Undercut.

Not all games are zero-sum. Figure 25.3 shows the matrix corresponding to what is perhaps the most famous game discussed by game theorists. This is the Prisoner's Dilemma. In the Prisoner's Dilemma, players don't compete with each other. Instead, they try to maximize a reward received from a third party. In the usual formulation, they try to minimize the amount of time they have to spend in jail. In Figure 25.3, the payout is listed as a payout vector, with both Andy's and Beth's payouts.

		Andy	
		C	D
Beth	C	(3,3)	(5,0)
	D	(0,5)	(1,1)

Figure 25.3

The Prisoner's Dilemma.

Note

Prisoner's Dilemma was first formulated by game theorists Merrill Flood and Melvin Dresher in 1950. The game can be presented in any number of ways, but one version might go like this. Two people are arrested and charged with a crime they are suspected of having committed together. During an interrogation following their arrest, the two people are placed in separate rooms, and rewards are offered to them if they confess. The conditions they are offered stipulate the possible outcomes. First, if one confesses and the other does not, then the person who confesses will be let free and the other will be punished fully. Second, if both confess, then both will still be punished, but the sentences will be reduced. Third, if neither confesses, there is the chance that neither will be convicted, but at the same time, if evidence exists to gain a conviction, both will suffer the maximum penalty. The objective for both prisoners is to gain the lightest penalty. The dilemma arises because each prisoner must trust the other either not to confess or to confess, but as it stands, both will clearly benefit if both confess. As it turns out, when people play this game, it is by no means a given that they will opt to confess.

Solving a Game

Theoretically, any zero-sum game can be solved, for each player has an ideal strategy for play that maximizes winnings. Consider, for example, the game of Rube, a matrix for which is shown in Figure 25.4. One difference between Rube and, say, Prisoner's Dilemma is that Rube clearly makes a fool of anyone who plays it.

For the game of Rube illustrated by Figure 25.4, imagine that Beth is a carnival cardsharp who offers a maximum payout of \$100. Andy is a gullible victim. Andy and Beth choose one of the cards J, Q, or K. The payout matrix is as shown. It should be immediately obvious that Beth's best strategy is to always choose the King, and Andy's best strategy is to choose the Jack, meaning that Andy will be paying Beth \$1 each time. But why is this the best?

		Andy			
		J	Q	K	Min
Beth	J	-100	2	-100	-100
	Q	-50	-100	2	-100
	K	1	2	5	1
	Max	1	2	8	

Figure 25.4

Rube.

The matrix includes Min and Max values. These represent, for Andy, the maximum amount he would pay to Beth given a particular choice of card, and for Beth, the minimum amount Andy would pay to her (or equivalently, the maximum she would pay to him). In both cases, the values represent the worst-case scenario for a particular card.

Each player's preferred option is to choose the card that minimizes the maximum payout to the other player. This is known as the *minimax strategy*. Using this strategy, Andy will choose J and Beth will choose K. Notice that in the payoff matrix, the value of \$1 is minimax for both Andy and Beth. This is a stable strategy for each player. If Andy chooses J, then Beth's best choice of card is K. Any other card will produce a worse outcome for her. Beth is minimaxing the negative value of the table entries.

If Beth chooses K, then Andy can't do better than to choose J. As becomes evident, then, as long as either player is using this strategy, the other player must also do so. That the strategies of the two players can be paired in this way is called the *Nash Equilibrium*. One theory derived from the Nash Equilibrium states that every finite, two-person zero-sum game must have either one or infinitely many pairs of strategies with this property. Many games involving more than two players also sustain this dynamic.

With the playing of Rube, the Nash Equilibrium strategy involves choosing a single move every time. The payoff of \$1 is called the *value* of the game, and the game is strictly determined because the minimax for each player is the same. Stated differently, the value \$1 in the payoff matrix is both the minimum value in its column and the maximum value in its row. This outcome is called a *saddle point* of the matrix.

Rock-Paper-Scissors is not determined in the same way. If you look at the payoff matrix for this game, you will see that there is no saddle point. In fact, there is not even a single minimax for either player. The absence of a saddle point means that no single strategy will work for either player. For any choice Andy makes, Beth can make a choice that will beat it. Both players must use a mixed strategy. Specifically, they must choose each option precisely one third of the time and at random. The game is made interesting by the fact that human beings are not very good at picking numbers at random. Given this reality, as you play the game, you can try to notice patterns in the other player's actions and anticipate them.

When dealing with games with no saddle point, you need a way to determine the appropriate mixed strategy. This requires *probability theory*. A *probability* is a number between 0 and 1 representing the likelihood that one event out of a set of events will occur. A *fair coin* is a coin balanced so that an equal probability exists that it will land either on heads or tails. The probability of throwing heads with a fair coin, then, is 0.5. With a six-sided *fair die*, the probability of rolling a 6 is 0.167. In general, the probability of a discrete event occurring is given by the following equation:

$$P(\text{event}) = \frac{\text{Number of ways event can occur}}{\text{Total number of possible events}}$$

The chance of throwing a number less than 3 on a single die is $\frac{2}{6} \left(= \frac{1}{3} \right)$, since there are two ways to throw a number less than 3, and six possible events altogether.

Suppose Mary offers that if you pay her \$3 per throw, she'll pay you \$1 for each pip that shows on a die. You can calculate the expected amount of money you will receive when throwing the die. This is the average amount you will win per game if you were to play it for long time, and it's essentially a weighted average:

$$\text{Expected payout} = \sum_{\text{events}} P(\text{event}) \times \text{event payout}$$

For Mary's offer, the expected payout is the sum

$$\frac{1}{6} \times 1 + \frac{1}{6} \times 2 + \frac{1}{6} \times 3 + \frac{1}{6} \times 4 + \frac{1}{6} \times 5 + \frac{1}{6} \times 6 = \frac{21}{6} = \$3.5$$

Taking off your initial payment of \$3, since your expected profit is \$0.50, the game is worth playing.

An *optimal strategy* for a game is a set of probabilities of choosing each of the options such that the expected payout is the same whatever choice the other player makes. For a matrix with a saddle point, the strategy is simple. Choose one option with probability 1. For others, you have to determine the probabilities algebraically. Consider the game of Undercut, played by Beth and Andy, the matrix for which is shown in Figure 25.2. Since the Undercut matrix is symmetrical, you know that the expected payout must be zero. If he picks the number i with a probability p_i , this gives you five equations for Andy's expected winnings based on a particular strategy:

$$\begin{pmatrix} 0 & -3 & 2 & 3 & 4 \\ 3 & 0 & -5 & 2 & 3 \\ -2 & 5 & 0 & -7 & 2 \\ -3 & -2 & 7 & 0 & -9 \\ -4 & -3 & -2 & 9 & 0 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Because the determinant of the matrix for Andy's expected winning is zero, there are infinitely many solutions to this set of equations, but you also have an additional piece of information. You know that the sum of the probabilities is 1. This sum gives you a strategy for Andy that always has an expected payout of 0, regardless of what Beth does. The same strategy also works for Beth. If you didn't have a symmetrical matrix and therefore didn't know the value of the expected payout, you'd solve separately for Beth and include one more unknown, the value of the payout.

Note

Solving the Undercut game is left as an exercise. If you work this problem, recall that the method for solving a set of simultaneous linear equations is discussed in Chapter 3.

The approach applied to the Undercut game can be used to analyze any zero-sum game in which all information about the game is known to both players. For a non-zero-sum game or a game with incomplete information, things get more complicated. If you are interested in pursuing this topic, there is plenty of theoretical work on such problems. An example of a more complex game of this description is Poker.

The information available to players distinguishes different games. A *simultaneous* game is one in which players choose actions without knowing the actions of other players. Prisoner's Dilemma provides an example of a simultaneous game. Other games are *sequential*. In sequential games, each player has some information about the actions of other players. The players take turns choosing an option, until one wins. While fully analyzing sequential games involves complicated work with strings of matrices called *Markov Chains*, from a computational point of view, a different, less mathematically involved approach can be used.

A Game Theory Approach to Tic-Tac-Toe

Like Rock-Paper-Scissors, a game that most people know well is Tic-Tac-Toe. In addition to being familiar, Tic-Tac-Toe happens to be an excellent game on which to apply AI theories. Figure 25.5 illustrates a game of Tic-Tac-Toe in which one player, Beth, has won by placing three Os in the second column. Since this is a sequential game, the two players, Andy and Beth, alternate as they play. Each places a symbol in a square of the grid until either the grid is full or one player has three symbols in a row, column, or diagonal.

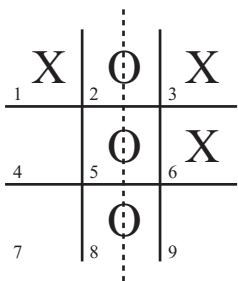


Figure 25.5

Tic-Tac-Toe.

In Figure 25.5, the squares have been labeled for ease of reference. To explore how game theory applies to the game of Tic-Tac-Toe, consider the *minimax* concept first. The minimax concept corresponds to the *min-max algorithm*. To represent the playing of the game, you create a search tree. The search tree works in a fashion similar to the octrees, quadtrees, or the maze networks. Each possible game position in the tree is represented by a node, and nodes are connected by the moves that lead from one to the next. Figure 25.6 illustrates the top of the search tree. Each layer, or level, from the top down represents a move by a particular player.

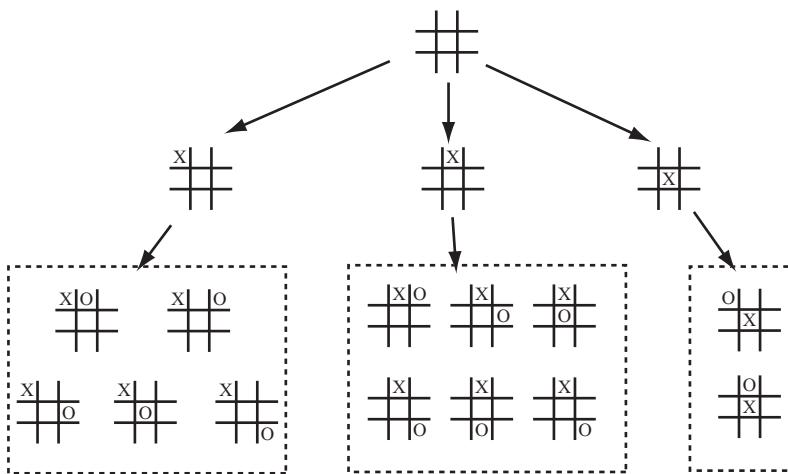


Figure 25.6

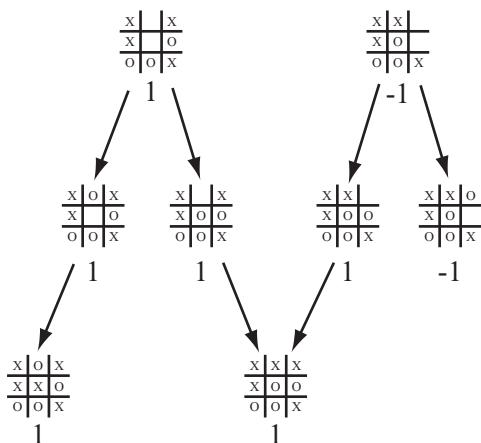
The start of the search tree for Tic-Tac-Toe.

Note

The term *search tree* as shown in this context is not strictly a tree, for two or more positions at one level can lead to the same position at the next.

With the representation of a game given in Figure 25.6, notice that some optimizations have been made. Since the game is symmetrical, many of the game positions are equivalent. Andy, the player using x's, has only three distinct first moves: corner, edge, or center.

To apply the min-max algorithm, you first go through the tree and find all end positions. End positions are those with three symbols in a row. Positions with a win for player 1 are given a score of 1. Wins for player 2 are given a score of -1. Drawn positions get a score of 0. Next, you follow all the links up the tree from these end positions. These links will tell you the possible positions that could lead to your end position. If these occur on Beth's layer, then you label each node with the minimum value of the nodes beneath. This tells you if Beth can win from this position. If they occur on Andy's layer, then you label it with the maximum value of the nodes. Figure 25.7 shows a portion of the search tree with these values calculated.

**Figure 25.7**

Labeling the search nodes.

Having completed a pass along the tree, you repeat the process just described, working up the tree until you reach the root node. This will tell you, for each player, the best possible move(s) for any given position. In the case of Tic-Tac-Toe, since both Andy and Beth can force a draw from any of the start positions, all the early nodes have a value of 0.

Applying a Search to Tic-Tac-Toe

Given the discussion in the previous section, it becomes possible to lay out a complete set of functionality for solving Tic-Tac-Toe games. The functionality involves a group of five functions that together find a perfect strategy for playing the game. The `makeTTTList()` and `makeTTTtree()` functions create the search tree. The other three functions follow up with analysis. Because this is only theoretical and you're not worrying about speed or memory usage, you'll save work by not eliminating symmetrical game positions. Here is the `makeTTTList()` function:

```

function makeTTTList()
  // creates a list of all possible game boards
  set blist to an empty array
  repeat for i=0 to 5
    set blank to an array of 9 0's
    if i=0 then add blank to boardlist
  
```

```

otherwise
  set bl to boardlist(blank, 1, i, 1)
  // list of all boards with i 1's
  repeat for j=i-1 to i
    if j=0 then next repeat
    repeat for each b in bl
      set bl2 to boardlist(b, 2, j, 1) // i 1's and j 2's
      add all of bl2 to bl
    end repeat
  end repeat
  end if
end repeat
return blist
end function

```

Here is the makeTTTtree() function:

```

function makeTTTtree(blist)
  // creates a tree of all possible moves in blist:
  // for each node, creates a list of all possible parents and children

  set tree to an empty array
  set e to array(empty array, empty array)
  add as many copies of e to tree as the elements of blist
  repeat for i=1 to the number of elements of blist
    set b to blist[i]
    // find all parents of b
    if b has no '1's then next repeat
    if b has an odd number of '0's then set s to 1
    otherwise set s to 2
    repeat for j=1 to 9
      if b[j]=s then
        set p to a copy of b, replacing the s with 0
        set k to the position of p in blist
        append i to tree[k][2] // children of i
        append k to tree[i][1] // parents of k
      end if
    end repeat
  end repeat
  return tree
end function

```

The `boardList()`, `makeMinimaxStrategy()`, and `minimaxIteration()` functions calculate the winning strategy. Here is the `boardList()` function:

```
function boardList (board, symbol, n, start)
    // fills a board with n copies of the symbol
    // in all possible ways (recursive)
    if n=0 then return array(board)
    set bl to an empty array
    set c to the number of 0's between board[start] and board[9]
    repeat for i=1 to c-n+1
        set b to a copy of board
        set k to the position of the next 0
        set b[k] to symbol
        set bls to boardlist(b, symbol, n-1, i+1)
        append all elements of bls to bl
    end repeat
    return bl
end function
```

Here is the `makeMinimaxStrategy()` function:

```
on makeMinimaxStrategy (tree, bl)
    // 'prunes' tree by removing all unnecessary children,
    // and returns an initial strategy and minimax tree
    set strategy and minimaxtree to arrays of the same length as bl
    set each element of strategy and minimaxtree to "unknown"
    repeat for i=1 to the number of elements of bl
        set b to bl[i]
        if b is a win for 1 then
            set strategy[i] to "WinX"
            set minimaxtree[i] to 1
            deletechildren(tree,i)
        if b is a win for 2 then
            set strategy[i] to "WinO"
            set minimaxtree[i] to -1
            deletechildren(tree,i)
        if b is full then
            set strategy[i] to "draw"
            set minimaxtree[i] to 0
        end if
    end repeat
    return [strategy, minimaxtree, tree]
end function
```

Here is the `minimaxIteration()` function:

```

on minimaxIteration(strategy, minimaxtree, tree, b1)
repeat with i=the number of elements of b1 down to 1
    // ignore nodes with no parent
    if tree[i][1] is empty and i>1 then next repeat
    // ignore nodes that have already been calculated
    if minimaxtree[i] is not "unknown" then next repeat
    set c to the number of 0's in b1[i]
    set ply to mod (c,2)
    if ply=0 then set ply to -1

    if there is any j in tree[i][2] such that minimaxtree[j]=ply then
        set minimax to ply
        set mv to j
    otherwise
        // find best non-winning move
        set minimax to "unknown"
        repeat for j in tree[i][2]
            set m to minimaxtree[j]
            if minimax="unknown" then set minimax to m
            if ply=1 then set minimax to max(m, minimax)
            otherwise set minimax to min(m, minimax)
            if minimax=m then set mv to j
        end repeat
    end if
    set strategy[i] to mv
    set minimaxtree[i] to minimax
end repeat
return strategy
end function

```

Limitations

While the Tic-Tac-Toe system provides a fairly reliable way to compute a winning strategy, it cannot be used all the time, applying it to different games. The reason for this is that it is pretty inefficient. For even a game as simple as Tic-Tac-Toe, there are over six thousand possible positions, and more than a thousand more if you eliminate reflections and rotations. For the games that can be played with boards, the numbers are several times higher. For a game like chess, there are thirty or more possible moves from each position, so the number of possible games is vast—significantly more than there are particles in the universe. To calculate the search tree for chess would take billions of years, and to use it in a game afterward would still take years.

There are a number of ways to optimize the search tree, of which the most important is the *alpha-beta search*. With the alpha-beta search, when searching through moves at a particular depth, you keep track of two values. One is *alpha*, the best score you know you can achieve; the other is *beta*, the worst score that your opponent can force on you.

As an example of how to apply the alpha-beta search to Tic-Tac-Toe, suppose Andy has played squares 3 and 8 and Beth has played square 6. It's now Beth's turn, and she has six squares to choose from. Suppose she moves into square 5, the center. It is theoretically possible that Andy will not notice the threat and will play somewhere other than square 4, but it is not likely that this will happen. Instead, Andy's going to play the correct move and block her. Because you assume that each player will always play the best move they can, it's no good thinking in terms of the best-case scenario. You must always think about the worst case. Alpha-beta searching means that you focus on the best moves the players can be sure of playing.

If you encounter in the course of a search a node whose value is less than the current alpha value, you know you can ignore it, since you've already discovered a better one. Similarly, if you find a node whose value is higher than beta, you can ignore this, too, since you know you'll never get to play it.

With the current discussion, the assumption of a perfect opponent is somewhat unrealistic when dealing with a human game. In fact, such assumptions are one of the major problems with rational models of game theory and models that involve economics. However, one of the main goals involved in creating an intelligent player is to give the player the tools to play without such brute-force methods. Chief among these is analyzing a board and choosing a move appropriately. In particular, you're looking for a way to reduce or simplify the search space by performing some kind of pre-analysis to detect promising paths through the tree.

Tactical AI

When you play a game by yourself, you don't generally think in terms that have much in common with game theory. Instead, you think in broader terms, such as gaining territory, gambits, threats, and maneuvers. This is what might be dubbed *tactical* thinking as opposed to *strategic* thinking. (Strategic thinking will be discussed later on.) Strategic thinking is concerned with planning long-range goals. Tactical thinking involves the execution or abandonment of goals when the immediate moves of a player are considered. Tactical thinking involves moment-to-moment play.

How Chess Programs Work

In 1997, one more item had to be struck off the list of things that computers can't do when the program Deep Blue beat World Champion Gary Kasparov at chess. This was a climactic moment in the history of computers that fulfilled half a century of work in designing AI programs that could play chess.

Chess is one of the most complicated and intractable games, especially given that, just like Tic-Tac-Toe, complete information about the game is available to both players, and no randomness at all occurs. The number of valid games of chess is more than astronomical. It is nearly inconceivable. Given this situation, it didn't take long for programmers to realize that creating a search tree for chess was impossible. They focused instead on trying to understand how a human being plays the game and how to formalize this into rules that a chess program can follow.

The key insight was that chess is primarily about territory. When grand masters are given a chessboard position to memorize, they can do so very quickly and can reconstruct the board much more accurately than novices. The reconstructions work only for genuine game positions. When the pieces are scattered randomly on the board, grand masters and amateurs perform much the same. Also, when grand masters make an error, the errors are highly global. The pieces are in positions far from where they should be, but the board is left tactically unchanged. The same squares are threatened and the same pieces in danger.

The insights concerning territory led naturally to the idea that, instead of analyzing a position in terms of how likely it is to lead to a win, you can analyze it in terms of its current tactical advantage to one or another player. Analyzing for current tactical advantage takes place through *estimation functions*. An estimation function is basically an equivalent of the underestimate function in the A* Algorithm. Such functions allow you to assign a number to any particular position. By combining this approach with a search tree, you can limit the depth of your search. The result is that you replace the win/lose/draw value with the result of the estimation function for a particular node. You then continue with the alpha-beta algorithm.

Use of an estimation function and a search tree can be combined with more sophisticated methods. For example, you can perform a quick estimation of all the first layer moves. Most of these can be discarded to create a smaller set to be evaluated to a second layer (or *ply*). When the moves are culled again, only the most promising paths are explored by more than one or two steps. This is also in keeping with the actions of real players. Experts tend not to analyze many possible moves, and bad moves are not only ignored but aren't even perceived. Even an amateur player won't consider the impossible option of a pawn moving backward or a castle moving diagonally.

The unavoidable downside of using an estimation approach to limit the search depth is that some winning paths might never be discovered because they look bad at the start. For example, programs that rely on estimation functions find it difficult to discover a winning plan involving a major sacrifice. The immediate loss is much more perceptible than the long-term gain. Still, if this is a problem with a computer algorithm, it is also a problem with human players.

Training a Program

Calculating the estimation function involves a lot of guesswork, plenty of trial and error, and some mathematical analysis, but in essence it boils down to one thing: you need to find a set of measurable parameters that succinctly describe the current position and that might affect your chances of success. Any number of such parameters apply to a chess program. Consider, for example, the total number of pieces (in pawn value) for each player and the number of pieces under threat. More subtle concepts include, among others, control of the center, exposure of the king, pieces in play, and strength of pawn line.

Describing measurable parameters has another advantage. It can be used just as effectively when dealing with games with a random element, such as Backgammon. Even when you can't determine the moves you'll be able to use with each successive round of play, you can still classify the strength of your current position. How spread out are your pieces? How many are exposed to capture? The list goes on.

To translate measured parameters into an expression useful for computation, after you have determined your parameters, you must then create a weighted sum of these numbers, which is to say a value $a_1p_1 + a_2p_2 + \dots + a_np_n$, where the weights a_1, a_2, \dots have been predetermined.

Coming up with such a parameterization is something you have to do as the designer of the program. Since there is no way that you can theoretically calculate the correct value of the weights, you must calculate them by trial and error. The most effective method to calculate by trial and error is to use an algorithm based on natural selection, which will be examined in detail in Chapter 26. Another method for calculation is to use a *training* system. Using a training system, the program modifies the weights it uses by analyzing the success or failure of particular moves.

To present a simple example of a training system, consider a chess game in action. Two training systems are in place, one for the player, the other for the player's opponent. Under a certain system of weights, the training program enumerates the value of the current position. Then, looking ahead by a certain number of moves, it chooses the move that

maximizes its expected value. At this point, the opponent program makes a move. After the opponent's move is concluded, the player's program starts again. If its estimation function for the current position gives a current value equal to or higher than the program expected to be the case, the move is deemed successful and the weight of the parameter that is most involved in the decision is slightly increased. If the current value is lower than expected, the move is deemed a failure and the weight of the responsible parameter is decreased.

Training and learning algorithms are related to the concept of *operant conditioning*. Operant conditioning is a form of learning that takes place using reward and punishment. It is often associated with the psychologist B. F. Skinner (1904–1990). With operant conditioning, a behavior is reinforced according to its past success. Programs that involve neural networks make use of notions associated with operant conditioning.

A Tactical AI Approach to Tic-Tac-Toe

With respect to how a training program might be applied to Tic-Tac-Toe, the first step is to suggest a few plausible candidates for measurable parameters of potential success. Here are a few possibilities:

1. The number of empty rows, columns, or diagonals
2. The number of rows containing only your symbol
3. The number of rows containing only your opponent's symbol
4. The number of potential forks (intersecting rows containing only your symbol)
5. The number of potential opponent forks
6. The number of threats (rows with two of your symbol and an empty square)
7. The number of opponent threats

The list of parameters contains some items that conflict with each other. Consider once again the game of Tic-Tac-Toe in which Beth opposes Andy. If Andy plays the top center and Beth plays the bottom center, Beth blocks one of Andy's potential rows, thus improving her score by criterion 3. On the other hand, by playing a bottom corner, Beth improves her score by criterion 2. Which of these is the better option? There is by no means an obvious answer. To find out, you can use a training process to find out what criterion should be assigned a higher weight. It's more than possible that one or more criteria might be assigned a weight of zero, meaning that it has no effect on the eventual outcome. You might also find that there are different sets of weights that produce a different but equally effective style of tactical play.

Top-Down AI

Instead of applying a tactical approach, you can also approach the AI problem using strategy. In other words, you can use a system that sets itself some kind of goal and tries to achieve it. Setting a goal and trying to achieve it is sometimes called a *top-down* approach. With a top-down approach, decisions at a high level of reasoning about the situation are used to make decisions at a lower level. The acronym GOFAI is sometimes applied to the top-down approach. This acronym stands for *good old-fashioned AI*. This name arose because for many years it was the primary route for mental modeling. Researchers worked by creating a *symbolic architecture* to represent the world and a *reasoning module* that tried to deduce facts and make decisions. In recent times, the approach has fallen out of favor among cognitive researchers. The bottom-up method has supplanted it. Still, the top-down approach remains fairly popular for dealing with limited domains and expert problems like games.

Goals and Subgoals

The goal-based approach to AI is similar to the process of programming. You start with a difficult task (“I want to make a first-person shooter”). You then break the task into subtasks (“I need a character, an environment, and some enemies”). Then, you break these up into smaller subtasks, until eventually the tasks are primitive in the sense that they can be programmed directly.

The toughest part of the process involves how to determine the subgoals. It’s not always obvious what steps to take to get to a particular destination. In a game of chess, the goal is “checkmate my opponent’s king,” but breaking this goal into smaller subgoals is a complex task. To accomplish this, the program requires a *knowledge base*. A knowledge base is as a representation of the world that the program can use to make deductions and judgments. A chess program, for example, might be equipped with advice (*heuristics*) such as “avoid getting your king trapped,” “try to castle early,” or “if you are ahead, then exchange like pieces for like.” The knowledge base also has some mechanism for deduction, and in particular for speculation. For example, “If I had a castle on that side, it would protect the queen when she checkmates the king.”

Armed with a knowledge base, the program formulates its strategy and, given a particular situation, forms a representation of the strategy and proceeds from there. Among other things, it assesses its goal (“checkmate the king with my queen on e8”). It notices potential pitfalls to the goal (“the bishop on g7 could move in to block it”). It creates a subgoal (“eliminate the bishop”). It searches for a solution to the subgoal (“capture it with my knight”). These actions continue. As it proceeds, the program homes in on a single move that advances the most immediate subgoal.

The goal-based program has more of a resemblance than the tactical approach to a human being playing chess. For this reason, AI researchers held so much hope for the top-down approach in the early days. In fact, this method has had many successes, especially in expert domains. Still, due to scale of the knowledge base required, the goal-based approach has proven hard to scale up into more lifelike areas, such as natural language.

When to Change the Goal

For the obvious reason that every plan has some potential flaw, strategy has to go hand in hand with tactics. No one can anticipate everything, and constraints on computing time mean that every strategy must necessarily contain some blanks. In fact, one of the primary motives for strategic thinking is that you don't have to worry about calculating every last move. You simply try to advance your position toward some final, nebulous goal. In a game of chess, you don't care if the queen is protected by the knight or the bishop. You know only that if the master plan is to succeed, the queen has to be somehow protected. However, narrowing actions in this way leaves the door open for an unexpected move to block any plan.

One useful way to combine strategy and tactics is to think of strategy as a particular set of weights in the estimation function of a tactical program. This is not likely to work well in a game like chess, but it will work in a simpler game, like Backgammon. In Backgammon, there are two primary strategies. The first is sometimes referred to as the *usual game*. With the usual game, you try to protect your playing pieces from being captured as you build up a strong base to trap opposing pieces. When bad luck strikes, it is sometimes appropriate to switch to a different strategy called a *back-game*. With a back-game, you try to get as many of your pieces captured as possible. While this worsens your position in some senses, it also gives you a strong chance of capturing your opponent and perhaps preventing him or her from reaching the goal. This reverses the fortunes of the game dramatically. While the back-game is a risky strategy and not one to attempt lightly, it can lead to exciting play.

You can model the use of the usual game and the back-game in terms of an estimation function with variable weights. At each stage, the function yields a best possible score under the current strategy. But if this score drops below a particular threshold, the program starts to try out alternative sets of weights. If any of them yield a higher result for some move or for several moves, then the program might decide to switch to an alternative strategy.

When playing against a strategic program, it becomes advantageous to try to induce its strategy from the moves it plays, just as you do when playing a human opponent. In light of this, another important part of strategic play involves *pattern analysis*. With pattern analysis, you try to determine an underlying principle for the past few moves of a player.

The web furnishes a number of examples of games that use pattern analysis. For example, you can find versions of Rock-Paper-Scissors that will most inevitably beat you. This is possible because humans are poor random-number generators. Such programs search for patterns in your previous moves and extrapolate to guess what your next move will be.

The simplest system of pattern analysis keeps track of each triple move you have made. In the sequence RPRSSRP, the program finds RPR, PRS, RSS, SSR, SRP. The program then predicts that if you have just played RP, you will probably play R. As you play more and more rounds, the program determines that, for example, having just played RP, you are twice as likely to play P as S. It then guesses that this pattern will continue. In general, the longer you play these games, the less well you will do. More subtle systems track whether you won or lost the previous round. In Exercise 25.1 you are asked to create such a program for yourself.

For a more complex game, pattern analysis relies on the program's strategy module. An example of this might be the question, "What would I do in that situation?" The advantage of asking such a question is that it gives the program a much more powerful way to estimate the likelihood of different moves from the opponent. Moves can even be classified as "aggressive" or "defensive," which significantly helps with culling of the search tree.

One final example of goal-changing under pressure is the system of *scripts*. With a script, you follow a predetermined strategy unless an unexpected event occurs. An example of this is a bot in a stealth game. A bot moves in a standard pattern or a random pattern with certain parameters until it hears a noise or discovers a dead body. Then the script for the bot changes, switching to a different alarm or search strategy. The new script might include such goals as "warn the others" or "find the intruder."

A Top-Down AI Approach to Tic-Tac-Toe

To repeat an earlier definition, top-down AI involves deriving decisions at a lower level from decisions made at a high level. Goals lead to subgoals. With reference to the tree analysis, the principle subgoal of Tic-Tac-Toe is the fork. The fork is effectively the only way to win a game. Below the fork, there is a subgoal of gaining control of empty pairs of rows. These goals coincide with the parameters you use to create an estimation function. This is an appropriate path since situations that are tactically useful tend also to be strategically useful. However, this is also to some extent a result of the simplicity of the

game. In a more complex domain, there are levels of description that are beyond the scope of the estimation function. For example, in a game of chess consider the question, “Can the queen reach square b5?” The usefulness of this question depends on your current goal. For most strategies, it will be a completely irrelevant question, so it would constitute a useless addition to the estimation parameters.

With Tic-Tac-Toe, your primary goal might be “create a fork,” but a subgoal is likely to be “create a fork between the left and top lines.” With such subgoals, you then arrive at a further question: “Is the top row empty?” If you are using an estimation function, such a question is not going to be useful. It is useful, however, if you have created a strategy.

To explore this scenario in greater detail, consider Figure 25.8, which illustrates a complete game of Tic-Tac-Toe with the moves numbered.

6 O	5 X	1 X
9 X	2 O	8 O
3 X	4 O	7 X

Figure 25.8

A strategic game of Tic-Tac-Toe.

One again, the two players are Andy and Beth. At the start of the game, Andy begins with a blank slate. His goal generator immediately starts looking for a possible fork, and he decides to play the corner square 3. Such a decision is based on probability. From his knowledge base, he knows that a good route to a fork using a corner square is to get the corner 7, forcing Beth to move in the center and leaving him open to move in one of the other corners, making a fork on two edge rows. This is his current strategy.

However, consider the difference between tactical thinking and strategic thinking. When thinking strategically, you are mostly thinking in terms of best-case scenarios. You are thinking about what you would like to happen if your opponent doesn’t guess what you’re up to. In the game theory approach, you think in terms of the worst case and assume your opponent will do the worst thing possible from your point of view. Of course, you need to think about your opponent’s reaction when deciding which strategy is best. There’s no point choosing a strategy that is bound to fail.

Beth, too, knows the three-cornered fork trick, so she knows that she can't let Andy force her into such a situation. Since the easiest way to foil it is to block the diagonal, she considers placing her next move in the center. She realizes that this move would also be a useful step toward creating her own fork, with three pieces in some corners.

Andy's original strategy is now partially damaged. Since he can no longer force the fork on Beth, he must now find some other way to take advantage of his original move. However, all is not lost because he still has partial control over the top and right rows, and there are only two other rows not controlled by Beth. This leaves him with two possible forks. One move, in corner 7, leaves both of them possible. While this is the move he was aiming toward before, at this point he's taking it for a different reason.

Incidentally, note that Andy's move damages Beth's strategy. One of the rows she wanted to use, on the bottom, is now unavailable. An alternative is available on the left. This makes it so that her only possible rows are the two in the center and the remaining diagonal. Further, there is no way to create a fork without forcing Andy to stop her. Her only option is to force him to a draw, so this becomes her new primary strategy. A quick prediction shows that if she plays a corner, he will fork her, so she has to play one of the edge pieces. From here on, play is predetermined. Each player is forced to play moves that block the other.

Ultimately, such analysis is overkill, but to some extent, it enriches the experience of what it is to play the game, consciously moving back and forth between strategic and tactical perspectives.

Bottom-Up AI

In the opposite camp from the top-down goal-led AI programmers are the *connectionists*. Connectionists advocate a bottom-up approach. The connectionist approach involves trying to create intelligent behavior through the interaction of large numbers of simple, stupid elements. Systems of this type are evident in the working of the human brain, through the interaction of its nerve cells. They are also evident in the behavior of ant hills, where the interactions of many ants give the ant hill the appearance of being driven by an overall purpose. In this view, purposes, goals and decisions arise as higher-level interpretations of mechanical events (*epiphenomena*). As it stands, however, the bottom-up approach is not used very often in games, but it is worth examining in this context due to the potentials it offers.

Neural Networks

Perhaps the purest example of connectionism is the *neural network* (or just *neural net*). A neural net is like a simulated massively parallel processing computer. Such a computer is made up of lots of smaller computers modeled on the nerve cells in the brain and called *neurons*. These artificial or simulated neurons are joined together in a network with input and output and trained to produce appropriate results by a learning process.

To understand how neural nets work, consider the Figure 25.9, which shows a line drawing of a neuron in your brain. A neuron is basically a simple calculation device. Each neuron has a main cell body or *soma*. Approximately a thousand filaments called *dendrites* lead into the soma. The dendrites provide input to the soma. For output, the neuron has an *axon*, which is attached to a side of the soma. The axon in turn splits into various filaments called *terminals*. As with the dendrites, there are roughly a thousand terminals, and each terminal ends in a small nodule called a *terminal button*. The terminal buttons are connected to the dendrites of other neurons via an entity called a *synapse*.

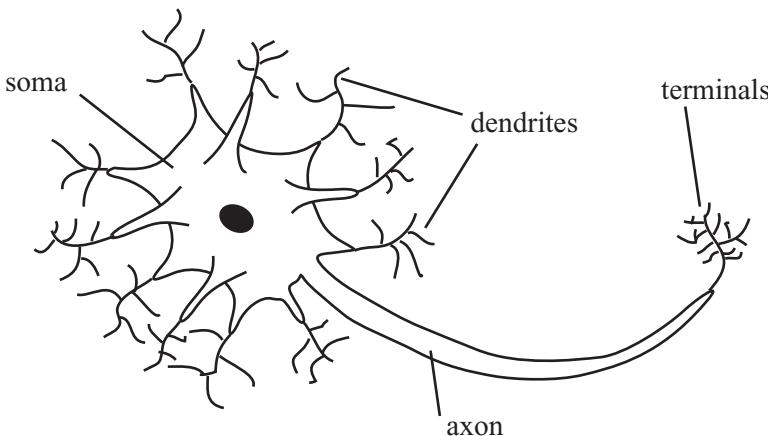


Figure 25.9
A neuron in the brain.

The neuron works by chemical-electrical signals. At any time it can fire or send a signal of a certain strength down the axon. The strength of the signal is always the same for a particular neuron, although the frequency at which the signals are sent can vary. Whether a neuron fires at a particular moment depends on the signals coming into the dendrites.

At any moment, the neuron will fire if the total signal coming into the dendrites is above a certain threshold. The sum is weighted according to the behavior of the dendrite. Some dendrites are *excitatory*, meaning that a signal received will be added to the total sum. Others are *inhibitory*, meaning that the signal is subtracted from the total sum. There is also some additional processing that interprets several rapid weak signals as a single stronger one.

After the logic of the neuron was understood, computer scientists sought to use it in computing. To create a neural network, you must model an artificial neuron. An artificial neuron lacks most of the complexity of a biological neuron. The actions of dendrites, terminals and other elements are replaced by a single function as shown in Figure 25.10.

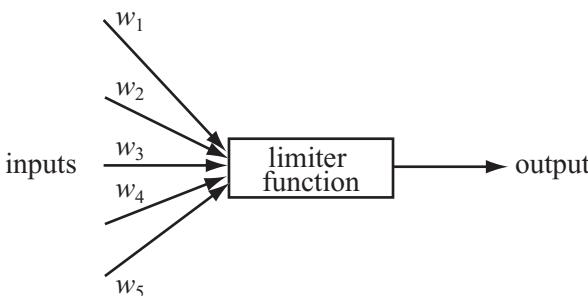


Figure 25.10

An artificial neuron.

The artificial neuron has a certain number of inputs, each of which has an associated weight. It can be given a stimulation threshold that determines whether it will fire. Likewise, it has an output that can be connected to any number of other neurons. At each time-step in the neuron program, you determine the output strength of each neuron by calculating the weighted sum of the inputs and seeing if they exceed the threshold for firing. If the neuron fires, the output can be either discrete, 1 or 0, as in the brain, or it can vary according to the size of the input.

You can avoid some worry by assuming that the total sum of inputs is passed through a *limiter function*. The limiter function constrains the output to a value between 0 and 1. For a discrete output using a threshold, this is a step function that outputs 1 if the input is greater than a certain value and 0 if it is not greater than a certain value. For variable output, you usually employ one of many versions of the sigmoid function $\frac{1}{(1+e^{-x})}$. The `neuralNetStep()` function provides an example of how to accomplish this:

```

function neuralNetStep (netArray)
  set nextArray to a copy of netArray
  repeat for each neuron in netArray
    set sum to 0
    repeat for each input of neuron
      set n to the source neuron of input
      add (weight of input)*(strength of n) to sum
    end repeat
    set strength of neuron in nextArray to
      limiterFunction(sum, threshold of neuron)
  end repeat
end function

```

Having set up a function for the neural net, you must provide the network some kind of connection to a problem. As illustrated by Figure 25.11, to do this, you create an *input layer* of neurons, which are connected to the *source*. The source might be a visual display or the values of a problem. You also connect the input layer to an *output layer*. The output layer might be a second visual display. Between the input and output layers, you place one or more *hidden layers*. This creates a multi-layer perceptron (MLP) network.

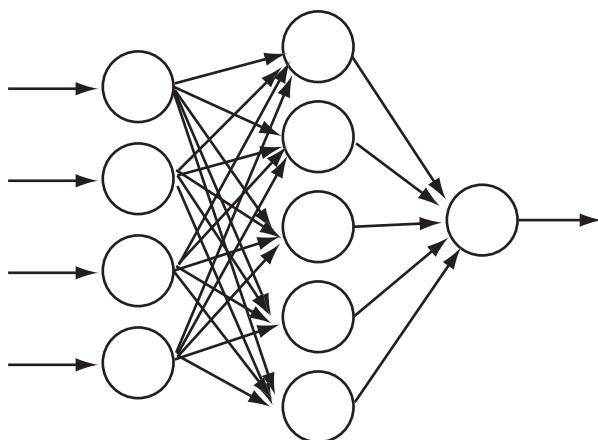


Figure 25.11

A multi-layer perceptron.

In Figure 25.11, the network might be described as having the problem of calculating the emotional state of a photographed face. It can classify the face as smiling or frowning. The input nodes are given a strength according to the grayscale color of the pixels in a bitmap, and the answer is determined by the output strength of the single output neuron, from 1 (smiling) to 0 (frowning). A more complex network could have more output nodes.

Each neuron in a layer is connected to every neuron in the neighboring layers, so information from one layer is available in some sense to every neuron in the next. In this kind of model, there is no feedback. No neuron can pass information back from one layer to an earlier one. This makes training the network easier, as is discussed in the next section, but it is unrealistic when compared to what happens with neurons in the brain.

Training a Neural Network

Given a particular problem, you need to train the network to solve it. The interesting thing about the neural net approach is that the neural net program knows nothing about the problem it is to solve. There are no explicit procedures or rules. All of the answers emerge from the interaction of all the neurons. As a result, training the network amounts to creating a computer program by an evolutionary process.

The key trick in training an MLP or other simple network is called *back-propagation*. Rather like the training method you saw earlier for the alpha-beta search, back-propagation works by a kind of behavioral conditioning. You give the network an input and reward it for an output close to the one desired. On the other hand, you punish the network for an output that is incorrect. The rewards and punishments consist of alterations to the weights of the network that improve the results the network delivers.

With the smiling-frowning detector, say that you seek to train it on a particular image. Suppose that after you feed the image to the input layer as a set of color values, the output neuron gives you a value of 0.62. You wanted the output value to be 1, however. The error value is 0.38. You now feed this value backward through the network.

The idea is to adjust the weights of each layer so that they produce a better result with each pulse of the network. You start by altering the weights of the output layer so that as it receives input, it produces a result closer to the expected value of 1. You trickle this effect back to the hidden layer(s), altering their weights to produce a value closer to the desired value. The process is similar to the training process discussed previously. In fact, a neural network is a useful way to create the estimation function for an alpha-beta search.

It's also possible to train a neural network by using an evolutionary mechanism, "mating" different networks to produce child networks and then selecting the best results from the offspring. This is a variant of the genetic algorithms explored in the next chapter.

Actors and Emergence

At a level more advanced than neural networks is a brand of connectionism that is based on the idea of small interacting subprograms often called *actors* or *agents*. This approach is popular among object-oriented programmers. An actor, like an object, is a self-contained entity in the software space that can be treated as autonomous. It has a simple behavior, but because it can interact with other actors in various ways, the overall effect can be to produce complex higher-level behavior.

An example of the use of actors is the *bulletin board model*. With the bulletin board model, actors interact by posting messages on a bulletin board. Other actors can pick up these messages. The messages are marked with an urgency and contain information or goals that must be achieved. Different actors might or might not be able to use the content of the messages. Another type of bulletin board provides a blackboard with messages written on it for all actors to see. Actors may erase or alter existing messages in light of their own knowledge.

The common feature of all these connectionist approaches is *emergence*. Emergence allows for higher degrees of organization to bubble up from interactions of simple elements. Emergence is an exciting phenomenon of interest mostly to the strong AI camp. Understanding it is worthwhile when looking at any contexts in which many individuals interact. Such contexts include traffic, crowds, and economics.

One example of emergence involves modeling of *flocking behavior*. Flocking behavior involves the motion of flocks of birds, schools of fish, or herds of animals. An early and extremely influential model of flocking was given by the Boids program, created by Craig Reynolds. The Boids program featured a number of simple "organisms" that followed just three rules:

1. Move toward the center of mass of all the other boids.
2. Try to match velocity with the average velocity of the other boids.
3. Never move closer than a certain distance to any boid or other obstacle.

A number of other rules can be added, such as avoiding predators or searching for food, but even with these three rules, you see surprisingly realistic flock-like behavior, suggesting that such rules might be behind the motion of real flocks. The boid model has been used for computer graphics and games, and most notably in films such as *Jurassic Park* (1993). More complex actor-based systems are used for battle simulations such as were seen in the *Lord of the Rings* film trilogy (2001–2003).

A final method is worth mentioning. Douglas Hofstadter has worked on a process called *high-level perception*. High-level perception involves building up a representation of a situation under various top-down pressures. This approach is more or less halfway between the bottom-up and top-down approaches. The system uses a number of code objects called *codelets* that search for patterns in data. The codelets use criteria in a shifting network of associations called the *slipnet*. The slipnet contains all the concepts the model understands, such as “sameness,” “difference,” “opposite,” “group,” and it uses them to build up a picture of the particular situation. What is interesting about this model is that it can “perceive” a situation differently according to different circumstances. For example, “abc” might be seen in one context as a “successor-group” of letters but in another as a “length-3 group.” This intertwining of perception and cognition is intuitively very lifelike.

A Bottom-Up AI Approach to Tic-Tac-Toe

Tic-Tac-Toe does not lend itself well to the bottom-up approach. A neural network could of course be used in place of the estimation function for the alpha-beta search. If this were used, however, it would be unlikely to be an improvement, and there is very little analysis of the board position for the actors to share between them. This isn’t to say that a bottom-up approach wouldn’t be interesting, but it wouldn’t be useful as a way to create an intelligent program, as compared to the other approaches.

One method that could be successful would involve creating a neural net that takes any board position and returns the next. This would involve a slightly unusual training regime. Instead of presenting it with a single stimulus and evaluating the response, it would have to play a complete game and be evaluated according to whether it won or lost.

Exercises

EXERCISE 25.1

Create a pattern-matching program that plays a game of Heads or Tails against a human opponent. You might use any of the approaches discussed in this chapter, but the easiest is the method mentioned in the section on top-down AI.

EXERCISE 25.2

Create a simulation of flocking boids. The three rules described in the chapter are fairly easy to implement and produce satisfactorily lifelike behavior, especially in 3-D. You might like to play with adding predator avoidance rules as well.

Summary

In this chapter, you've encountered a rapid overview of a vast topic. Hopefully, you have emerged with a good understanding of the general approaches to AI and how they are applied to different problems. While this chapter does not provide you with enough information to create your own AI implementation, it should at least allow you to know what you are looking for.

In the final chapter, you will follow up on this chapter and look at genetic algorithms and other methods for searching through large problem spaces.

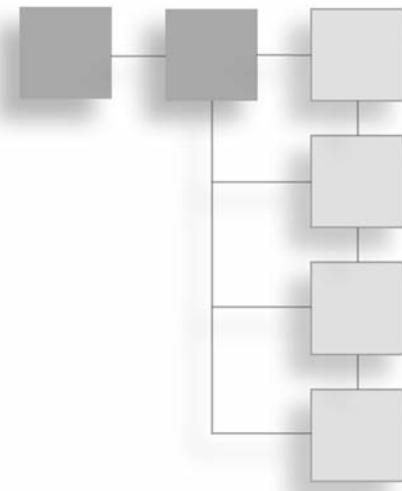
You Should Now Know

- How to analyze a simple game using *game theory*
- How to create a complete strategy for a short game
- How to simplify a strategy using *alpha-beta* searching
- What an *estimation function* is and how to train it
- The difference between *top-down* and *bottom-up* approaches, and between *strong* and *weak* AI
- How a program can break a problem up into *goals* and *subgoals*
- How a *neural network* is built and how it can be used to solve problems

This page intentionally left blank

CHAPTER 26

SEARCH TECHNIQUES



In This Chapter

- Overview
- Problem Solving
- Case Study
- Genetic Algorithms

Overview

You have arrived at long last at the final chapter, which follows naturally from the previous chapter by looking at methods for using the computer to solve problems. A leading topic of this chapter concerns how the computer can be used to search through many possibilities to find an answer to a problem or a problem with a particular answer. As was discussed in Chapter 22, finding the optimum solution to a difficult problem can be a big task, with no computationally efficient algorithm that can solve it. However, as you saw in Chapter 24, you can take advantage of a few tricks to arrive at better answers. While you're looking at such tricks, you'll explore how to organize the computer's resources during a long search (one that might take days or longer).

Many of the techniques in this chapter are exact analogs of ideas in the previous two chapters. There isn't really a strict dividing line between AI problems and complex search problems. However, the domains in which they are being applied are different, and it should be instructive to see how similar methods can be adapted to different situations.

Problem Solving

In this section, the discussion involves general ways in which a problem can be broken down into a form that the computer can be set to solve. Primary concerns of problem solving are computational capacity and search space.

Representing a Problem

There are two main areas in which you might want to use the computer to solve problems, especially in a game context. The most obvious is as a game player, when you want to solve a difficult puzzle and don't know how. An example might be an anagram generator used to find an elusive answer in a crossword. Less obvious is as a game creator who wants to use the computer to create a puzzle with a particular answer. An example might be filling a crossword grid with words or creating a chess problem. The two situations are fairly similar, but filling in a crossword is more interesting, and for this reason, this chapter will concentrate on it.

When assessing a problem involving filling in a crossword, the first step is always to try to classify your *search space*. A search space is the set of possible answers to your problem. If you're filling the crossword grid Figure 26.1 illustrates, your search space is the set of all possible combinations of one letter in each blank square or cell. The search space in this instance consists of a total of 26^{25} possible crosswords. You can represent this as a 25-element array.

1		2		3
	4		5	
6				
7				

Figure 26.1

An empty crossword grid.

However, this might not be the best way to represent the grid, for only a tiny proportion of these arrays represent valid solutions to the problem. A more useful method might be to represent the words instead of the cells. This yields an eight-element array, with each entry containing a single word. An eight-element array is going to be much more amenable to searching, but you need to take into account the links between words in addition to the words themselves. You can do this by creating a template that sits alongside the solution in progress. In the template, each word is named and represented by a smaller array, something like this:

2dn: [1ac/3; blank; 6ac/3; blank; 7ac/3]

Using the template, whenever your algorithm fills in a potential solution to a word, it can check the template for cross-references and simultaneously fill them in. Such initial considerations can be crucial to creating a representation of your problem that the computer can use to execute a search strategy.

Searching for Answers

You'll be looking at some specific examples of search strategies shortly, but prior to that, consider some general concepts first. The main feature that distinguishes problems that you would like your computer to solve from problems that might be solved manually is that you want your computer to solve problems involving a very large search space but relatively simple solutions. For example, the crossword puzzle illustrated in Figure 26.1 has over 10^{35} possible combinations of letters. Searching through all combinations is far too vast an order if you expect any quick results. Given any particular combination of letters, however, it becomes relatively easy to check for a solution. To do so, you simply check whether each word appears in your dictionary.

The preceding paragraph informally describes the class of problems known as *NP-hard*. (NP stands for *nondeterministic polynomial-time*.) Formally stated, an NP-hard problem is one for which there is no known algorithm that can find the solution in polynomial time, but for which, once you have found the solution, it is possible to prove in polynomial time that it is a solution. NP-hard problems are rather common, and there is an interesting subspecies of them known as *NP-complete* problems. The two types of problem are equivalent in that if you find a polynomial-time algorithm that can solve any one of them, then the same algorithm can be adapted to solve all others in polynomial time. Whether the class of NP-complete problems is actually solvable in polynomial time is itself not yet settled, although most mathematicians seem to think they are not.

Given that you know these problems are impossible to solve quickly, why do you bother with them? The answer is that although there is no general algorithm guaranteed to find an answer to any given problem, you can find algorithms that will improve your chances. They do this by speeding up the journey through the search space. Recall the discussion of the A* Algorithm from Chapter 24. Although in a worst-case scenario the A* Algorithm is no faster than an exhaustive search, in the vast majority of cases, it will be much faster than an exhaustive search.

The principal tool in a search is to look for bottlenecks. Starting in this way is basically the equivalent of conducting the alpha-beta search looked at in the previous chapter. A useful starting point is to try to solve a simple example by hand and see where you naturally focus your attention. In the crossword grid in Figure 26.1, you might start by filling in 1ac at random. Say that you start with the word SPACE. It's natural to focus next on the three long down words (or *lights*, as they are known by crossword aficionados). In particular, the most obvious word to look at is 2dn, which begins with the letter A, since there are fewer words starting with A than there are with S or E. You then might choose the word ANVIL, which would lead you naturally to focus on the V in the center. Alternatively, you might decide against ANVIL and choose ALTER, since this gives a greater number of options for the crossing words.

This process can be semi-formalized as follows. Choose a word that maximizes the minimum number of options for the crossing words. (This should seem rather familiar.) Then search on the light that has the smallest number of options remaining. If you find that a particular light is blocked—there are no possible words that fit—backtrack one step and choose another word.

Note

Since it is a less difficult task, in this section, you won't look further at the question of how to search through a dictionary for entries that fit a particular pattern. A number of programs will perform it for you, and many languages include Regular Expression and other functionality to make such searches easy.

The strategy that begins with searching for bottlenecks and proceeds as described in the previous paragraphs is an example of what is called a *depth-first* search. This was discussed in Chapter 24, and it is applicable in many of circumstances. Generally, the problems it is used most frequently to address involve any case in which several co-dependent options have to be chosen in parallel.

Another example is searching for a solution to the *polyominoes problem*. As shown in Figure 26.2, the polyominoes problem involves finding how to fit a certain set of shapes into a particular space. In the partially complete example Figure 26.2 illustrates, you might try focusing on the square marked with an X. This square can be covered only by one of the remaining pieces. You might also focus on the piece shaped like a cross, which can fit in only three spots on the board. Either of these approaches is equivalent to the method described previously in this section.

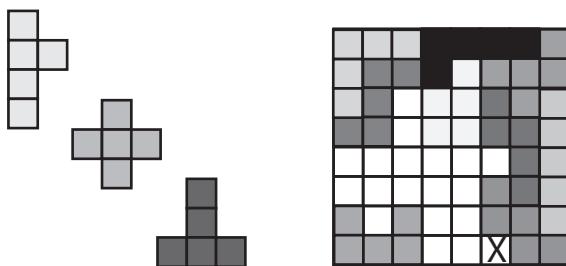


Figure 26.2
A polyominoes problem.

The alternative to a *depth-first* search is a *breadth-first* search, which is a search in which all the possible alternatives at each stage are considered in parallel. Generally breadth-first searches are less efficient than a well-informed depth-first search because of the large number of possibilities to be considered. A circumstance in which you might consider a breadth-first search is one that provides a good chance that the search depth is shallow. This is the kind of problem for which you know that a simple solution exists, for which you do not know the solution, and for which there is no particular reason to try one solution rather than another.

As you can imagine, such situations are quite rare, as it's almost always possible to think of some heuristic that can be used to prefer some solution paths over others. Having said this, it is often possible as with the A* Algorithm to combine depth-first and breadth-first strategies into a more complex method that uses the best features of both. This constitutes a shallow analysis of all possible paths that informs the choice of paths for a more thorough exploration. Such an approach provides a more complex heuristic for choosing the paths for the depth-first search. It is reminiscent of the look-ahead strategy discussed in Chapter 24.

Interaction

Important points to consider are how you can get an indication of progress with solving the problem and also how you can influence the process of solution. If a problem is going to involve several hours or days of computer time, you don't want to set it going and have no idea whether it is working correctly. More importantly, if the computer crashes during a run, you don't want to have to start the whole thing over again.

One result of auditing progress is that, despite the obvious benefits in terms of problem solving, as a general rule you can't use direct recursion in your search algorithm. This is a pity, because many of these problems lend themselves to a recursive structure. To see how this is so, consider the crossword problem once again. The crossword problem can be summarized using the following steps:

1. Try to fill in the next light.
2. If you can't, then backtrack and try another possible word for the previous light.
(If there are no other words for the previous light and it was the first word, then there is no solution.)
3. If you can fill in the light, then if this was the last word, you're done; otherwise, find the next word to try and repeat the steps of this algorithm.

This recursive structure seems natural in a depth-first search, and it's relatively simple to construct an algorithm that implements it. Unfortunately, it quickly leads to a call stack that is very large and memory-hungry. What's more, if the computer crashes, the whole stack is lost. To respond to this situation, you must unroll the recursion. While you must preserve the recursive structure, you must also fake it by creating your own call stack. This allows the search to be broken into discrete steps, instead of running as a single function call.

Having done this, you can take advantage of the discrete steps of the function to intersperse them with visual displays of the current state of the search. You can generate a partially completed crossword, for example. You can also take periodic snapshots in the form of text files that represent the best result so far. The result is the best so far in the sense that it combines a large number of steps through the search with a large number of alternative options at each step.

Case Study

It's interesting to examine how the concepts discussed so far can be used to deal with a complex problem. In light of this, it is worthwhile to spend time examining the process of creating a program that finds chess problems. Stated differently, given a particular checkmate position, how do you search for a starting position that leads to a checkmate

of the form “white to move and mate in n ”? Although this problem is very specific, it serves to highlight several important issues.

In the following discussion, it is assumed that you know a little about how chess works, but you don’t really need to know more than the fact that players take turns to move a single piece and that a single piece may or may not capture (remove from the board) an opponent’s pieces. Likewise, if one player maneuvers the other into a position in which his or her king piece will be captured (checkmate), then this player wins the game.

Likewise, this section does not provide any implemented functions. Instead, it provides only function names, and the implication is that the descriptions of the functions will help you if you attempt to construct them for yourself. At the same time, you can also refer to the source code for the book for some examples.

Preparing the Ground

As before, you need to start by defining the search space and how it can be represented. This is fairly straightforward in this instance, for a chessboard is an 8×8 array whose elements can either be empty or contain one of 12 different pieces: a pawn (P), knight (N), bishop (B), castle (R for rook), and king (K) or queen (Q) of either color (W or B). For convenience, you label the squares of the board with letters for each column and numbers for each row. Using this scheme, as Figure 26.3 illustrates, A1 (or a1) is in the bottom-left corner. You also need one more piece of information: which player is due to play next.

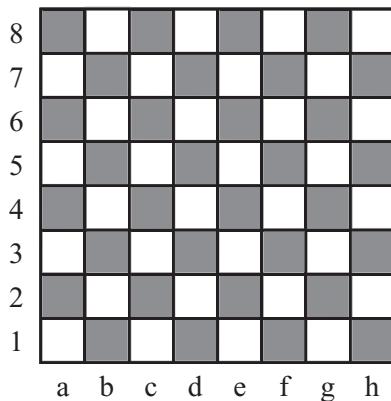


Figure 26.3

A chessboard.

For a completely free chess problem, while the only other constraint on the search space is that each color must have exactly one king on the board, you can define a few other constraints that apply to any position that occurs as the result of an actual game. Most of these constraints can be violated in extreme circumstances due to special rules, such as promotion of pawns, but in this case this option will be ignored. Here is an essential list of constraints:

- There can be at most eight pawns, two knights, bishops and castles, and one queen of each color on the board at any one time.
- If there are two bishops of any one color, they must be on different colored squares of the chessboard.
- No pawn can be on rows 1 or 8. A pawn can move to row 8 but is immediately promoted to some other piece.
- The player who last moved cannot currently be in check. In other words, a player's king cannot be under threat from any opponent's piece.

Of these, all but the last are what might be called *syntactic* constraints. In other words, you don't need to know anything about the rules of chess to determine if they have been violated. You need only to be able to count pieces, look at the board, and so on. The last constraint is a *semantic* constraint. To determine if it is true, you have to know how each piece can move.

Such considerations quickly lead you to create initial functions for working with the search. The most fundamental is one you might call `possibleMoves(board)`. This function takes as input a board position (including the current player) and returns a list of all possible moves in some useful format, such as "a1b1". This function in turn calls a function `possibleMovesForPiece(board, square)`, which returns a list of all possible moves for a particular piece. You can use these functions, in turn, to create a function called `underThreat(board, square)` that looks through the list of all possible moves and determines whether any of them end on a particular square. If a move meets this criterion, then the target square is considered to be under threat.

To create the `possibleMovesForPiece()` function requires implementing the rules of chess, and it's not entirely simple, especially if you want to take into account special moves such as castling or pawn capture *en passant*. In general, however, such special moves are not used in chess problems. The only issue that really needs to be considered is pawn promotion.

With these functions in place, you can create a function called `validBoard(board)`. This function returns `TRUE` if the board is a possible position and `FALSE` otherwise. This function takes into account all the considerations mentioned above and uses the `underThreat()` function to determine if the opponent's piece is in check. You can also use the `validBoard()` function to improve the `possibleMoves()` function by making a `validMoves()` function that not only checks whether a move is possible but also determines whether the resulting position is valid. In this respect, it determines if, after the move has been played, the player's king is under threat. With this function, you can quickly determine if a player is in checkmate. Checkmate is the equivalent to saying that the king is under threat and no valid moves remain. If there are no valid moves and the king is not under threat, this is called a stalemate.

You now need just one more function. This function might be called `validMovesInto(board)`. It calculates all valid moves that can lead to a particular position. Fortunately, because most chess moves are reversible, this function is fairly similar to the `validMoves()` function, but it involves an added complication. For most moves, the moving piece can perform a capture. This means that when undoing the move, you might reveal a new piece that wasn't on the board before. As a result, there are more possible moves that lead to a particular position than there are moves that lead from that position.

Given that you have constructed all these functions, you are in a much better position to define exactly what it is your search function is to do. You want it to take a board position, usually a checkmate by white, and to find a position preceding it by n moves such that the following conditions are fulfilled:

- No sequence of moves from this position leads to any other checkmate in n moves or fewer, as long as black plays perfectly.
- No sequence of moves from this position leads to black avoiding checkmate after n moves, as long as white plays perfectly.

Creating the Search Function

To create a search function, one place to begin is with the most readily identified movement or set of movements. The simplest case is a “mate in one.” If you give the computer a checkmate position, drawing on the set of functions discussed in the previous section, the search uses the `validMovesInto()` function to find all possible board positions that lead to this position. For each of these positions, it then finds the list using `validMoves()`. If any of these leads to a checkmate that is not the final game position you're looking for, then the given chess problem is not valid. Chess problems should have exactly one correct answer. If you've found one that has no other checkmating move, your search is complete.

Notice the back-and-forth nature of this search. You first take a step backward to find a possible pre-position, and then you play a game of chess from that position. In this instance, only one move is involved. Your action allows you to see whether this position is suitable for your purposes. Each step involves a search process. You search through all the possible pre-positions to discover the most likely to be suitable. You also search all the possible post-positions looking for one that invalidates the step. To accomplish these two tasks, you require a suitable heuristic to find the best order to search.

To find a suitable heuristic and successfully conduct the searches, the key is to minimize your alternatives. When searching through pre-positions, you want to preferentially choose those that have the fewest valid moves from the designated position. In the forward search, you have to check every move. (There are fewer forward than backward moves.) Again, however, you want to look first for those most likely to invalidate the puzzle, which means using normal chess heuristics to find the best moves from the designated position.

The search becomes more complicated when finding a mate in more than one move, because you now must take the opponent's moves into account, as well. Having found a mate in one, you then must consider an opponent's move that could lead to this position. This is going to involve a backward search followed by a forward one, but this time, the opponent's move has to be forced, in the sense that the particular move you're interested in must be the best possible move that can be made. In this case, when the next move by white is to be the checkmate, black's move has to be the only valid move available. Any other move, even if it leads to checkmate, will not lead to the checkmate you want. Further down the line, black may have other moves available, but they must all lead to checkmate sooner than the target path.

Having established black's move, you now have another position to aim for with white, which means another back-and-forward check. No matter how good your heuristics are, there is a lot of ground to search. But this process is still going to find a suitable position faster than a simple examination of all possibilities. What's more, this approach seems intellectually plausible. If you try to create a chess problem for yourself, you'll find yourself working through exactly the same process.

Genetic Algorithms

One further example of a search method is worth examining. This approach is to connectionist AI what standard search methods are to top-down AI. The genetic algorithm simulates the force of natural selection acting on a population of computer programs.

Natural Selection

To understand the workings of the *genetic algorithm*, it helps to understand a little about real-life genetics. Toward this end, it is necessary to take a break from mathematics and physics to look at biology. An organism's physical structure is principally determined by a molecule called DNA. DNA is a long chain-like structure made up of copies of four different molecular units called *bases*, denoted by the symbols A, C, G, and T. A typical strand of DNA might consist of the bases AGCCATAGTTACGT. While each cell in a particular organism contains a copy of the DNA molecule with the same sequence of bases, the exact sequence varies from organism to organism. Unless you happen to be an identical twin or a clone, your DNA is different from the DNA of every other human being.

In this context, the most useful way to think of a DNA strand is as a program that contains instructions on how to build an organism. A DNA strand is made up of individual subsequences called *genes*. The word *gene* is used rather loosely to refer to various structures, from sequences that code for particular proteins to abstract components of the DNA that happen to determine a particular trait of the organism. From a programming point of view, you can consider *genes* to be like functions or objects. A complete set of genes is referred to as the *genotype*, and the organism built by the genes is called the *phenotype*.

The simplest way for an organism to reproduce is to create a new organism with an exact copy of its DNA. This is called *asexual reproduction*, and most bacteria, fungi, and plants can reproduce in this way. Asexual reproduction is very rapid, but it poses one problem. When all organisms come from copies of the same genotype, they tend to be highly susceptible to diseases, predators, and parasites. If a particular disease comes along that can harm one organism, that disease can most likely harm the whole population.

The alternative to asexual reproduction is *sexual reproduction*. Here, two different individuals, with different genotypes, join together, mingling their DNA to create a unique individual. Such creation requires that their DNA structure must be compatible. The sexual partners must have a similar set of genes (a *genome*) distributed in similar places along the DNA strand. If Romeo and Juliet were to create a child, the gene determining Romeo's eye color must be in the same position on his DNA as the eye-color gene on Juliet's DNA. Different versions of the same gene are called *alleles*. (Continuing the programming analogy, they are like different instances of the same class, with different properties.)

For the vast majority of genes, you inherit not one but two alleles, one from each parent. Romeo and Juliet's child will have two genes for eye color. Of these alleles, one is *dominant* and determines the color of the child's eyes. However, either of them might be passed on to the child's children. Two organisms whose genomes are organized in the same way are said to belong to the same *species*.

The process of creating a child by sexual reproduction is quite simple. First of all, each parent produces a number of sex cells or *gametes*. For animals, this involves sperm and eggs. Sperm and eggs are like normal cells, except that they contain only one copy of each gene. The key step is called *mitosis*, which is where the two versions of the genes line up in the cell and split in half to create two gametes, with half of the alleles going into one gamete, and the rest into the other.

The distribution of alleles is random. It makes no difference whether a particular allele comes from the father or the mother. They each have the same chance of ending up in either gamete. These gametes are then sent out to see if they can find a gamete of the other sex. If they can, they combine their half-set of genes (or rather, complete set of unpaired genes) with those of the other gamete to create a new cell that has a complete and entirely new combination of genes. In the process, there will also be occasional copying errors called *mutations*, which means that the alleles received by the new organism are not exactly the same as those of its parents.

What is the advantage of all this effort? The answer is easiest to see if you move back to the level of the organism. In a population of similar organisms, only some can survive to reproduce. Many will die of diseases or be eaten by predators before they can mate. Others will fail to reproduce because they do not attract potential mates. The result is that only those alleles carried by successful organisms will survive to the next generation, carried forward in proportion to the success of those organisms carrying them.

The process of carrying forward alleles is *natural selection*. Natural selection was first described by Charles Darwin (1809–1882). Use of the word *selection* suggested a similarity to the process of artificial selection, or selective breeding, which had been used for centuries by farmers to improve the yield of their crops and livestock. Natural selection will occur in any situation in which you find reproducing organisms passing on their characteristics to their offspring, variation between different organisms in the population, and competition for resources. Sexual reproduction is an excellent way for genes to improve their chances of spreading through a population.

Under natural selection, successive generations of organisms evolve to adapt to changing environments. Natural selection, unlike artificial selection, is a blind process. Evolution is not directed toward a particular goal but simply proceeds by incremental steps depending on the reproductive success of one generation after another. Random events like meteor strikes can mean a whole species adapted to existing environments dies off before it can evolve new mechanisms to cope. However, it's possible after the fact to see the existence of particular evolutionary pressures that happened to direct evolution in one direction rather than another for a particular species. Humans evolved under evolutionary pressures toward an upright stance, larger brains, and tool use, among other things.

Evolutionary pressures are emergent phenomena in the sense that they are the result of any number of smaller micro-evolutionary events. To draw from the previous discussion of search algorithms, one way to conceive of evolutionary pressures is to imagine organisms as exploring a search space of possible strategies for existence. There are vast numbers of ways to create an organism. A particular species consists of small variations on a theme, all exploring the same region of the search space, otherwise known as an *evolutionary niche*. If some of these strategies happen upon a previously unexplored corner of the niche, then they may find themselves doing a little better than the rest of their species. Their success means that this new area of the search space can be opened up for further exploration. If it proves better than the area inhabited previously, the whole species might start to migrate toward it.

Even if it has accounted for all currently known life, evolution by natural selection might seem a hopelessly inefficient way to develop new strategies. After all, artificial selection is hugely faster and more reliable. Still, it remains that it's a remarkably good way to find hidden solutions to a problem. With sufficient environmental pressures, organisms can evolve dramatically in just a few generations, especially if they reproduce sexually.

The Genetic Algorithm

Genetic algorithms apply the evolutionary approach to search through complex spaces. They follow the biological model fairly closely, although of course they are an idealization, just as neural networks are an idealization of biological neurons.

To explore a computational problem, consider once again the realm of chess and imagine that you're searching for a solution to the puzzle of finding a way to place eight queens on the board so that none of them threatens any of the others. To create a genetic algorithm, you have to characterize your search in terms of a genome. In the context of a program, a genome is a list of values that represent the search space. In this case, you can use a 64-element list in which each element is either 1 or 0. Each set of eight bits represents the position of one of the eight queens. If there is a duplicate, then the list is invalid, but you can deal with this by ignoring invalid genotypes. Such genotypes might be considered as phenotypes that are not viable organisms.

The other factor important in the creation of the algorithm is a utility function. The utility function tells you how close you are to a correct solution. To develop the utility function, you can calculate the number of ways that each queen is under threat. As illustrated in Figure 26.4, for example, the function evaluates to eight. Your aim is to minimize the value returned by the utility function. The value returned by the function for each possible organism gives what is called a *fitness landscape*. A fitness landscape is a representation

of success of different answers. Viewed graphically, valleys in the landscape represent areas that are successful, and you are searching for the lowest such valley (or in a case like this, for any valley with a minimum of zero).

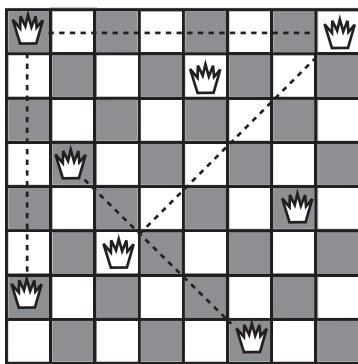


Figure 26.4

A set of eight queens that does not solve the problem, together with its genotype.

The utility function is the weakest link in the genetic algorithm approach. The reason for this is that if your problem is an all-or-nothing deal, then there is no advantage to being close to the correct solution. Your algorithm has nothing to get hold of. The valleys in the fitness landscape have very steep sides, so the algorithm can't find them. This problem reflects the classic objection to evolution suggested by creationists, the problem of intermediate forms. Half an eye might be said to be an intermediate form. What is the advantage of having half an eye? The objection is an important one in the context of a search algorithm.

Advocates of natural selection are generally glad to admit that if anyone could find a biological structure that had no immediate advantage at each stage of its evolution for the organisms carrying it, then it would be a serious problem for evolutionary theory. Fortunately no such structures have been found to date. (Half an eye is actually quite a big improvement on no eye at all.) In computation, it's almost always possible to find some utility function that allows you to evaluate the success of a particular proposed solution. What's more, the function can often be coarse-grained, with lots of different phenotypes having equal fitness. In fact, this can be advantageous to the algorithm.

The basic principle of the genetic algorithm involves taking a small population of computationally defined organisms and evaluating them with the utility function. Those that do best (those with the lowest score) are allowed to “mate,” producing a new generation of organisms. The mating process is based on mitosis. You take the DNA strands of each organism and splice them together. The `mateOrganisms()` function encapsulates this activity:

```
function mateOrganisms(strand1, strand2)
    set child to an empty array
    set currentStrand to strand1
    set length to the number of elements of strand1
    repeat for i=1 to length
        // randomly switch strands
        if random(length)<5 then switch currentStrand
        set element to currentStrand[i]
        // randomly mutate the occasional element
        if random(length)=1 then set element to not(element)
        add element to child
    end repeat
    return child
end function
```

While the figures given in the `mateOrganisms()` function are set fairly arbitrarily, you are likely to find that approximately one mutation and five switches per strand work well. Although it might seem like magic, the algorithm tends to approach an optimal solution. What’s more, it deals well with local minima in the fitness landscape. In other words, it generates answers that are close to optimal but are dead ends. The lowest point in the particular valley is not an actual minimum. Local minima are the biggest problem for all search strategies, but the random element of genetic algorithms gives them a means of exploring far-flung areas of the search space at the same time that they are focusing on the current primary route.

Tweaking

Despite the simplicity of the system, there are many ways to vary the genetic algorithm methodology to improve the rate of searching. Which of these is more effective depends on the situation, particularly the number of local minima. One of the principal problems with genetic algorithms is that they have a tendency to get stuck in a rut, with the gene pool losing variation and one basic pattern taking over. Conversely, too much mutation can mean that the algorithm never gets a chance to approach a solution before it is distracted by some alternative. The following list provides a few themes for variation.

1. In the standard method, no distinction exists between the winners of the evolutionary race. At each stage, the field is simply divided in two. Suppose there are 15 organisms. You can choose the six best and mate each one with each of the others. This produces a new set of 15 organisms. An alternative is to weight the results by awarding more children to the most successful organisms. In this case, in one scenario, the overall winner might get to mate twice with each of the others while the worst does not get to mate with anyone except the winner. This speeds up the process of finding a solution in a landscape with few or no local minima.
2. You can use a radiation variable. This affects the rate of mutation and possibly also the rate of crossover. By raising and lowering the radiation periodically, in order to focus on the current path, you can give the algorithm periods of stability interspersed with periods of instability in which more far-flung ideas may get tried out. An example of this can be seen in the `geneticAlgorithm()` function in the source code for this book. This helps to break free of ruts, but it might be worth preserving some of the more successful genotypes in case they get lost.
3. You can alter the population size, allowing for a more varied gene pool to be maintained. Using this approach tends to make less of a difference than one might think. A population of around 50 to 100 organisms seems to be fairly optimal.
4. Generally, the new generation completely supplants the previous one. Instead, you can allow parents to compete with their children, so that the previous best solution is not completely lost.
5. You can introduce mass extinctions by occasionally killing off a larger number of the population—possibly even the more successful ones. This is equivalent to an epidemic that attacks the most prevalent genotype in a population, allowing the underdogs to come through.
6. You can allow a kind of speciation by splitting off separate populations and allowing them to evolve separately. You then put them together again to compete and interbreed.
7. You can introduce a system of paired genes, as in biology, where each organism inherits two alleles instead of just one. However, this involves coming up with some reasonable method for determining which allele is dominant.

Most of the ideas in the list of variation themes are designed with a view to simulating conditions in biology, particularly the process of artificial selection. You give natural selection a helping hand to speed it up. However, the danger in all search strategies is that you might push the algorithm too hard and lose track of all its advantages.

The alternative method is to tweak the utility function. You can do this by changing its granularity. As you saw above, coarser-grained utility functions can help to smooth out local minima. Those with finer grains can make it easier to approach a solution once it's in sight.

Another interesting option is to create a second utility function that uses an alternative means to evaluate the success of an organism. For example, when searching for an algebraic solution to a differential equation, you might have one measure for "simplicity" and another for "accuracy." An ideal organism minimizes both, but in general there is a trade-off between the two. When comparing two organisms, one is considered better than the other if it does better on both measurements. If it does not do better on both measures, then it is considered equally good.

It's also worth remembering that the utility function is generally the most computationally costly part of the algorithm. This is so because it needs to be evaluated at every stage for each member of the population. Speeding up the execution of the function allows more generations to be completed in a particular time.

Exercise

EXERCISE 26.1

Following the suggestions in this chapter, create a genetic algorithm process to solve the eight queens problem: place eight queens on a chessboard so that none of them is threatening any other.

Summary

This chapter brings the book to an end. In this chapter, you explored a few methods for searching through large numbers of possibilities to solve difficult problems. While focusing on extensions of the AI approaches from the previous chapter, you have also explored the genetic algorithm method.

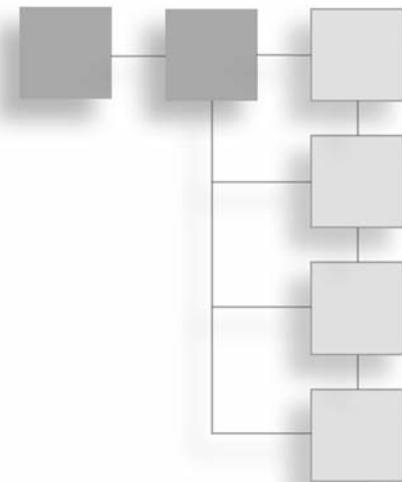
It is hoped that reaching this point has left you with memories of an enjoyable journey that began with the simple principles of numbers and elementary algebra you saw in Part I. From the foundational elements of mathematics and physics, you've seen how you can build complex and subtle techniques and apply them to programming, particularly in games. Although some of the more difficult topics have been only sketched out, you've had a chance to think about the issues involved and get to know some of the key terminology. In many cases, you have had a chance to work with the principles in action, so that you are in a better position to make use of technical coverage found in specialized resources on the subject. Above all, it is hoped that along the way you've found some tidbits that have inspired you to try new things. Creating an AI bot or the ultimate game of 3-D bowling are certainly worthwhile explorations, among many, many others.

You Should Now Know

- How to classify a problem in terms of *a search space*
- How to explore a search space using *depth-first* or *breadth-first* strategies
- The relationship between *genetic algorithms* drawn from evolutionary biology
- How to create a genome to represent a search space
- The meaning of the terms *utility function* and *fitness landscape* and how a genetic algorithm uses them
- Some strategies for optimizing the speed of a genetic algorithm

APPENDIX A

GLOSSARY OF MATHEMATICAL TERMS



This glossary can't cover all the mathematical terms introduced in this book, but it outlines the most fundamental terms. Some complicated, strict mathematical terms are left undefined, since their vernacular meaning is clear enough (for example, words like *point*, *shape*, and so on). A few other terms that don't appear in the book are introduced here to help make other explanations briefer and more rigorous (such as *locus*), although many of the definitions still fall short of a strict mathematical formulation.

absolute value—For a real number n , if $n < 0$ then its absolute value is $-n$, otherwise it is n . In other words, the absolute value of a number other than zero is always positive.

acceleration—The rate of change of velocity or speed.

acute angle—An angle smaller than a right angle.

affine transformation—A transformation that preserves parallel lines.

air resistance—The force experienced by an object moving through the air, resisting the motion; also called fluid resistance or drag.

algebraic solution—The result of finding the values of the unknowns in an equation in general terms of the other variables in the equation; compare this with *numeric solution*.

algorithm—A computational process consisting of a number of predefined calculations that take on particular arguments (essentially a program with inputs).

aliasing—The process by which a line vector is converted to hard-edged pixels; compare this with *antialiasing*.

ambient light—The general illumination of a region as a result of light reflected in all directions (the amount of illumination in areas of shadow); see also *directional light*, *attenuated light*.

amplitude—The maximum distance from equilibrium achieved by an oscillator during any oscillation.

angle—A measure of rotation, defined as a fraction of a circle and usually measured in degrees or radians.

angle of incidence—The angle at which an object or wave strikes a surface.

angle of reflection—The angle at which an object or wave bounces off a surface after colliding with it.

angular frequency—The number of rotations completed by a rotating object in a particular time period (see *frequency*).

angular velocity—The angle turned through by a rotating object in a particular time period.

antialiasing—The process by which a line vector is converted to smooth-colored pixels by gradually interpolating its color with the background.

argument—A variable provided as an input to a mathematical or programming function.

armature—A polygon or polyhedron used to create a simplified description of an object for the purposes of collision detection; compare this with *collision map*.

associative—Of some operator $\#$, says that for each a, b, c in the domain, $a \# (b \# c) = (a \# b) \# c$ (for example: $a + (b + c) = (a + b) + c$); compare this with *commutative*, *distributive*.

asymptote—A line or plane such that the graph of some function approaches it infinitely closely.

attenuated light—Light emitted from a particular point, which diminishes with distance as a result of spreading out; see also *ambient light*, *directional light*.

average—A generic word for various kinds of “middle” value of a set of values; most commonly the mean.

axis—Of a Cartesian space, a line through the origin, parallel to one vector of the basis.

axis of rotation—The line around which an object is rotating.

baked texture—A texture map in which lighting calculations have been made in advance.

ballistics—The study of objects in motion when acted on only by the force of gravity.

barycentric coordinates—A homogeneous coordinate system where the coordinates of a point P are given by the (signed) areas of the three triangles made between P and three predefined points A, B, C .

basis—A set of vectors, together with a defined origin, used to define a Cartesian space (or any other vector space).

Bezier curve—A spline defined by a cubic function between successive control points, with a defined tangent at each control point; compare this with *Catmull-Rom spline*.

Boolean algebra—A system for working with “Boolean numbers”: numbers that can take only two possible values, TRUE and FALSE, using operators such as AND, OR, and NOT.

bounding volume—For some shape in 3-D space, a volume of space entirely enclosing the shape; similarly “bounding area.”

bounding volume hierarchy—A partitioning tree in 3-D space (similarly, “bounding area hierarchy” in 2-D) in which objects are divided into successively smaller regions that enclose smaller sets.

bracketing method—A method for finding the approximate numeric solution to an equation by narrowing down an interval within which the solution is to be found.

breadth-first search—A method for searching through a search tree by examining each possible branch at a particular level; compare this with *depth-first search*.

bump map—A texture map describing the height of a surface at particular points.

calculus—The study of infinitesimally small values, particularly integration and differentiation.

Cartesian coordinates—The set of numbers that define the position of a point in Cartesian space under some basis.

Cartesian plane—A two-dimensional Cartesian space.

Cartesian space—(also called Euclidean space): a set of points defined by a vector of n real numbers such that every triangle of points has three angles summing to 180° (very briefly!).

Catmull-Rom spline—A spline defined by a cubic function between successive control points, such that each curve segment is determined by four control points, creating a smooth transition; compare this with *Bezier curve*.

center of mass—For any object, the point such that for any line or plane through the point, half the object’s mass is on each side of the line.

centrifugal force—The force exerted by a rotating object moving in a circle; compare this with *centripetal force*.

centripetal force—The force required to keep an object moving in a circle; compare this with *centrifugal force*.

centrum—The center of mass of a triangle.

child node—In a tree, a node x is a child of node y if and only if y is the parent node of x .

circle—A shape defined as the locus of a point at a constant distance r from another point in a given plane.

circumference—The perimeter of a circle.

coefficient—The constant part of a term in an expression: so the coefficient of the term $4x$ is 4.

coefficient of elasticity—The constant of proportionality relating the extension of a stretched spring to the tension.

coefficient of friction—The constant of proportionality μ relating the force perpendicular to the interface between two surfaces to the frictional force that resists their motion.

collinear—Of three or more points, means that they lie in a straight line.

collision map—An image map describing an object’s shape in simplified form for the purposes of collision detection; compare this with *armature*.

common factor—For two values a and b , a number (or expression) that is a factor of both a and b .

commutative—Of some operator $\#$, says that for each a and b in the domain, $a \# b = b \# a$ (for example: $a + b = b + a$); compare this with *associative, distributive*.

complex numbers—The set of numbers that can be represented as the sum of a real number and an imaginary number, denoted by the symbol \mathbb{C} .

component—The length of a vector when projected in a particular direction. If the direction is defined by some unit vector \mathbf{u} , then the component of the vector \mathbf{v} in that direction is found by the scalar product $\mathbf{v} \cdot \mathbf{u}$; sometimes as “the components of a vector” means its components in the directions of the underlying basis.

composite number—A natural number that is not prime.

computational complexity—A measure of the time taken for an algorithm to run, as a function of the size of its arguments.

concave—Of a shape, means that it is not convex.

cone—The surface of rotation of a line rotated about a non-parallel line in space; compare this with *cylinder*.

congruent—Two integers a and b are congruent modulo a particular base m if and only if $a = b + nm$ for some integer n .

conjugate—Of a complex number or quaternion, the result of taking the number and changing the sign of the imaginary part.

conservation of energy—The law stating that for any set of objects with no net force acting on them (other than those included in potential energy calculations), the total energy of the system is constant.

conservation of momentum—The law stating that for a system of particles moving with no external force, the total momentum of the system is constant.

constant—An element in a function that is considered to be fixed when evaluating the function.

constant of proportionality—See *proportional*.

control point—Of a spline, a point defining the shape of the curve in some way, usually a point on the curve.

convex—Of a shape, means that any line drawn between two points on the perimeter does not pass outside the shape; compare this with *concave*.

coordinates—See *Cartesian coordinates*.

coprime—Two natural numbers a and b are said to be coprime if and only if their greatest common divisor is 1.

countable set—A set whose elements can be listed in some order, even though the list may continue forever (they can be put in a one-to-one correspondence with the natural numbers).

counting numbers—See *natural numbers*.

coupled oscillators—Two or more oscillators that are connected, so that each exerts a force on the other.

critical—Of some value, the point at which the behavior of a system changes qualitatively, for example, the critical angle at which light cannot escape from a particular medium due to refraction.

cross product—See *vector product*.

cubic—A polynomial of degree 3.

curve—A set of connected points in space definable by a single continuously varying parameter.

cylinder—The surface of rotation of a line rotated about a parallel line in space; compare this with *cone*.

damped harmonic motion—The motion of an oscillator that would normally undergo simple harmonic motion but is also experiencing a damping factor.

damping—A factor opposing the motion of an oscillation, proportional to the speed.

degree—Of a polynomial function $f(x)$, the largest exponent of x , so the function $x^3 + 2x$ has degree 3.

degree—A unit of angle, defined as 1/360 of a circle.

denominator—In a fraction, the number on the bottom (also called the “divisor”).

depth-first search—A method for searching through a search tree by examining a particular branch until it succeeds or fails, then backtracking; compare this with *breadth-first search*.

derivative—The function g that gives the gradient of some other function f for each value of its arguments; compare this with *integral*, and see *rate of change*.

determinant—The “magnitude” of a matrix: the amount by which the matrix scales a cube when used as a transformation.

diagonal—A line connecting two vertices of some shape; more specifically, the longest possible such line.

difference—The result of subtracting two values.

differential equation—An equation relating a function to its various derivatives.

differentiation—The process of finding the gradient of some function (the derivative).

diffraction—A phenomenon where a wave changes shape as a result of passing through a small gap, acting as if the gap were a new emitter for the wave.

diffuse reflection—(also Lambertian reflection) Light emitted from a surface as a result of light incident on the surface, with no preferred direction; compare this with *specular reflection*.

dimension—Of a vector or Cartesian space, the number of components or basis vectors required to define it.

directional light—Light emitted in a particular direction, with no presumed diminution over distance (for example, sunlight); see also *ambient light*, *attenuated light*.

discriminant—A number derived from the parameters of a function whose value can be used to classify the behavior of the function in some way, for example, to find the number of roots.

displacement—The vector from one point to another.

distance—The length of a line from one point to another, defined as the magnitude of the vector between them.

distributive—Of some operators $\#$ and $@$, says that for each a, b, c in the domain, $a \# (b @ c) = (a \# b) @ (a \# c)$, for example, $a \times (b + c) = (a \times b) + (b \times c)$; compare this with *commutative, associative*.

divisor—See *factor*.

domain—The set of values for which a function is defined.

Doppler shift—A phenomenon where a wave has a different frequency in a different reference frame.

dot product—See *scalar product*.

drag—See *air resistance*.

e—The number $2.718\ldots$ such that the function $x \rightarrow e^x$ has a derivative equal to itself.

eccentricity—The amount by which an ellipse deviates from a circle, defined by

$\sqrt{1 - \frac{b^2}{a^2}}$ where a and b are the halfmajor and halfminor axes, respectively.

efficiency—The proportion of energy used in some system which is converted to useful work.

elastic limit—The maximum extension of a stretched spring for which the coefficient of elasticity remains valid.

elastic potential energy—The potential energy of an object under the influence of a stretched spring or elastic.

ellipse—The locus of a point in a plane whose total distance from two defined points is constant.

ellipsoid—The result of an affine transformation applied to a sphere.

emergent phenomenon—The collective behavior of a group of simple interactive processes.

energy—A measurement of the amount something is moving, or could move given an opportunity; see also *kinetic energy, potential energy, gravitational potential energy, elastic potential energy*, and *conservation of energy*.

equation—A statement that two values are equal to one another, usually with one or more unknowns.

equilateral triangle—A 2-D shape with three sides of equal length.

equilibrium—A state of a system of objects where there is no net force.

estimation function—A function evaluating how likely a particular solution to a problem is to be near to the correct answer.

Euclidean space—See *Cartesian space*.

exponent—The power to which a number has been raised, so in the term x^5 , x has an exponent of 5.

exponential—Of a function, says that it behaves similarly to the power function mapping $x \rightarrow e^x$; compare this with *logarithm*.

expression—A function consisting of a combination of terms, such as $3y^2 + 10xz - 5(x^3 - 2)$.

extension—The amount a spring or other elastic material is stretched from its natural length (see *elastic limit*).

factor—For two values a and b , a is a factor of b if and only if b is an integer multiple of a .

factorization—The process of separating a function or number into a multiple of two or more factors.

field of view—The area visible at a particular distance from a camera; alternatively the angle subtended by the viewport at the camera.

finite set—A set with a limited number of elements.

focus—Of an ellipse, one of the two points which define it (pl. foci).

formula—An equation containing more than one variable, which relates these quantities to one another (usually defining one variable as a function of the others). Formulas do not have unknowns; they are assumed to be true for all possible (true) values of their variables.

fraction—The result of dividing one number by another. (When both numbers are integers, this is a rational number.)

frequency—The number of complete oscillations completed by an oscillator in a particular time; compare this with *period*.

friction—The force experienced between two surfaces parallel to their direction of motion, resisting the motion, proportional to the force between them perpendicular to their surfaces; see *coefficient of friction*.

frustum—A 3-D shape formed by taking any kind of pyramid or cone and cutting off the top.

fulcrum—The point or line around which a lever or other object rotates; its *axis of rotation*.

function—A mapping from one set of values (the domain) to another (the range). A function can be represented algebraically ($f: x \rightarrow x^3$ or more succinctly $f(x) = x^3$) or simply by a description ($f(n) = 1$ if n is odd, -1 otherwise).

general solution—A function defining all possible solutions to a differential equation in terms of parameters remaining to be defined by the initial conditions; compare this with *particular solution*.

genetic algorithm—A method for searching through a search space by “evolving” solutions using an analogue of natural selection.

global maximum—The highest value of some function for any value of its arguments (similarly “global minimum”); compare this with *local maximum*.

gradient—The slope of a line in some Cartesian space, defined by the vertical distance moved divided by the horizontal distance for some infinitesimal movement.

graph—Either a picture of a function created by plotting its values against its arguments, the Cartesian plane on which this happens, or a set of nodes connected together by edges.

gravitational potential energy—The potential energy of an object under the influence of gravity.

greatest common divisor—For two natural numbers a and b , the largest number which is a common factor of both.

greedy algorithm—An algorithm that searches preferentially for a local solution to a problem.

halfmajor axis—Of an ellipse, half of the distance between the two most extreme points on the perimeter (along the line through the two foci). The halfminor axis is then the longest distance from this line to the perimeter.

homogeneous coordinates—A method of representing a point in a space of n dimensions by a projection of a line in $n + 1$ dimensions.

hypotenuse—In a right-angled triangle, the side opposite the right angle.

identity—A value which leaves all other values unchanged under some operator. For example, the identity function $f(x) = x$, the identity matrix which has 1s in the leading diagonal and 0s elsewhere, the number 0 under addition or 1 under multiplication.

if and only if—To say one statement is true “if and only if” the other is true means that either both are true or both are false (sometimes written “iff”).

image map—An image (usually in 2-D) that is projected to a surface by some mapping, or vice versa, to represent some property of the surface; see also *collision map*, *texture map*, *bump map*.

imaginary number—Any multiple of the value i , defined as the square root of -1 . The set of imaginary numbers is in a sense perpendicular to the set of real numbers.

independent equations—Two or more simultaneous equations, none of which can be derived from the others.

inequality—A statement that two values are related to one another in some way (for example, that one is greater than the other); compare this with *equation*.

inertia—See *mass*.

initial conditions—Facts about the initial state of a system that allow you to deduce its later behavior, particularly when using a differential equation.

integer multiple—For two values (numbers or expressions) a and b , b is an integer multiple of a if and only if there is some integer n such that $b = an$; compare this with *factor*.

integers—The set of natural numbers combined with the set of negative natural numbers ($\dots, -2, -1, 0, 1, 2, \dots$) and denoted by the symbol \mathbb{Z} .

integral—The function f that gives the (signed) area of an infinitesimally thin slice between some other function g and the axis (indefinite integral), or by substituting two values for its arguments, the area under the curve between those two points (definite integral). The inverse of the derivative.

integration—The process of finding the area between a graph of some function and the horizontal axis (the integral).

intercept—For a function $f(x)$, the value $f(0)$.

interpolation—The process of finding an intermediate value between two other values by parameterizing the line between them.

interval—A set of real numbers, which may be “open” (not containing its end points, as in the set $-1 < x < 0$) or “closed” (containing the end points, as in the set $-1 \leq x \leq 0$), or a mixture of the two.

inverse function—The inverse of a one-to-one function f , denoted f^{-1} , is defined over the range of f such that if $f(x) = y$, then $f^{-1}(y) = x$. Equivalently, $f^{-1}(f(x)) = x$ for all x in the domain of f . A many-to-one function can also be loosely defined to have an inverse by restricting the domain or accepting a multivalued function.

inverse kinematics (IK)—The process of calculating the motion required by a series of connected rigid bodies to achieve a particular goal.

inverse matrix—Of a square matrix \mathbf{M} , with non-zero determinant, the matrix \mathbf{M}^{-1} such that $\mathbf{M}^{-1}\mathbf{M} = \mathbf{M}\mathbf{M}^{-1} = \mathbf{I}$, where \mathbf{I} is the appropriate-sized identity matrix.

inversely proportional—Two values x and y are inversely proportional if there is some constant k such that $x = \frac{k}{y}$ for all valid values of x and y ; compare this with *proportional*.

inverse-square law—A formula relating two quantities x and y , saying that x is inversely proportional to the square of y .

irrational numbers—The set of real numbers, such as π and the square root of 2, which are not rational.

isosceles triangle—A triangle with two of its sides having equal length.

iterative function—An algorithm that works by successively using the results of each step in the next; compare this with *recursive function*.

kinetic energy—The energy of a moving object, defined as half the square of its speed multiplied by its mass, with similar calculations for angular velocity.

kinetic friction—The friction between two surfaces moving relatively to each other; compare this with *static friction*.

Lambertian reflection—See *diffuse reflection*.

lamina—An idealized object in three-dimensional space with a thickness of 0.

law—Also called a “physical law,” a formula that states some kind of supposedly fundamental truth about the physical world, usually false.

leading diagonal—Of a matrix (m_{ij}) , the values along the diagonal from top left to bottom right, with values m_{ii} .

leaf node—Of a tree, a node with no children.

line segment—A curve of points with position vectors $\mathbf{a} + t\mathbf{v}$ for some vectors \mathbf{a} and \mathbf{v} and some value $0 \leq t \leq 1$.

linear function—A polynomial of degree 1.

local maximum—A value of some function f for some values of its arguments such that all values of f for nearby arguments are less (similarly “local minimum”); compare this with *global maximum*.

locus—The set of points with a particular property (for example, a straight line is the locus of a point in a plane at an equal distance from two other points).

logarithm—The inverse of the power function: for two numbers a and b , a number x such that $b^x = a$. b is called the “base.”

lowest common multiple—For two natural numbers a and b , the smallest number that is an integer multiple of both a and b .

lowest terms—Of a fraction, means that the numerator and denominator are coprime.

magnitude—The length of a vector defined by the Pythagorean formula as the square root of the sum of its components.

many-to-one function—Of a function, states that several elements in the domain can map to the same element in the range; compare this with *one-to-one function* and *multivalued function*.

map—See *function*.

mass—The property of matter that defines how much force is required to make it accelerate; also called “inertia.”

matrix—A two-dimensional array of real numbers.

maximum—A point on the graph of some function where it is at its highest. For a smooth curve, the gradient is zero; see also *local maximum*, *global maximum*.

mean—For n values, the sum of the values divided by n ; compare this with *average*.

mechanics—The study of objects in motion or equilibrium.

mesh—A shape in a computer-generated 3-D world, consisting of a number of vertices connected by polygons.

minimax—In a game, the move for any player that minimizes their maximum possible loss (equivalently, maximizes their minimum possible gain), hence an algorithm making use of this concept to find the best strategy for playing a game.

minimum—A point on the graph of some function where it is at its lowest. For a smooth curve, the gradient is zero; see also *local maximum*, *global maximum*.

mip-map—A texture map in several variants of different sizes used to create smooth shading over distance.

model—A shape created in a computer-generated 3-D world, made from a mesh.

modulo—A function mapping the set of integers to the finite set $\{0, 1, \dots, m - 1\}$ for some natural number m , by mapping each number to its remainder when divided by m ; also called a “congruence relation.”

moment of inertia—The angular equivalent of mass, which for a particle is equal to the product of its mass and the square of its distance from the axis of rotation.

momentum—The product of an object’s mass and velocity; see *conservation of momentum*.

multivalued function—Of a function, states that some elements in the domain can be mapped to more than one element in the range (though this is not a function in a strict definition of the word); compare this with *one-to-one function* and *many-to-one function*.

natural numbers—The set of “counting numbers” including zero (0, 1, 2, . . .) denoted by the symbol \mathbb{N} .

network—A graph in which the edges have been assigned a value, usually representing a “cost” of some kind.

node—A virtual object, such as a vertex of a graph or element of 3-D space.

normal—Of two vectors, means that they are perpendicular. Of a surface, means a vector perpendicular to the surface.

normalized—Of a vector, the result of dividing it by its magnitude to get a unit vector.

numerator—In a fraction, the number on the top (also called the “dividend”).

numeric solution—The result of finding the values of the unknowns in an equation by searching for the specific numbers which make the equation true (as opposed to an algebraic solution).

obtuse angle—An angle greater than a right angle but less than a straight line.

octree—See *quadtree*.

one-to-one function—Of a function, states that each element in the domain is mapped to exactly one element in the range, and vice versa; compare this with *many-to-one function* and *multivalued function*.

operator—A function that maps one or more arguments from a particular set to a single result in the same set. You usually think of the binary operators such as $+$, $-$, \times , \div .

origin—A point defined as the starting point of some basis.

orthogonal—Of two or more vectors, means that they are mutually at right angles.

orthonormal basis—A basis where the basis vectors are orthogonal and have a magnitude of 1.

oscillation—A motion that repeats over time, such as a bouncing spring or an object bobbing on water; see *frequency*, *period*, *amplitude*.

out of phase—Of two oscillations, means that they have the same frequency but their phase is different, or specifically is half the period.

pairwise multiplication—Of two vectors, the operator that creates a vector from the products of each component of the arguments, for example, the pairwise product of the vectors $(2 \ 3)$ and $(1 \ -2)$ is the vector $(2 \ -6)$.

parabola—A curve in two dimensions; the graph of a quadratic function.

parallel—Of two vectors, means that one is a scalar multiple of the other. Of two lines, means that any vector on one is parallel to any vector on the other.

parallelogram—A quadrilateral with two pairs of parallel sides.

parameter—An element in a function that can be varied to create a family of different functions with similar properties.

parameterization—Defining some curve or surface in terms of a parametric equation.

parametric equation—A set of formulas relating one or more variables to one or more real-valued parameters.

parent node—In a tree, the node next in line from any particular node toward the root node; compare this with *child node*.

partial derivative—The derivative of a curve drawn on a surface.

particle—An idealized object in space, with a size of zero but some mass, as well as perhaps other properties.

particular solution—A function for which a differential equation holds true, given particular initial conditions; compare this with *general solution*.

partitioning tree—A tree representing a group of objects in space according to their relative positions.

perimeter—The curve or surface defining the outer border of some shape.

period—The time taken for some oscillator to complete a single oscillation; compare this with *frequency*.

perpendicular—Of two vectors, means that they are at right angles; compare this with *orthogonal*.

perturbation—Altering a value by some small amount.

phase—Of two waves with the same waveform, the distance by which their wavefronts are separated at a particular time.

plane—The locus of a point in space at the same distance from two defined points. Equivalently, it has position vector $\mathbf{a} + t\mathbf{v} + s\mathbf{w}$, where \mathbf{a} , \mathbf{v} , and \mathbf{w} are defined vectors and t and s are parameters.

ply—In a game, a move by one player.

point of inflection—A point on the graph of some function where its second derivative is zero.

polygon—A closed shape made up of a number of straight line segments connecting the same number of vertices.

polyhedron—A closed 3-D shape whose perimeter is a number of polygons joined by straight edges.

polynomial—A function in one variable, consisting entirely of terms of the form ax^n , such as $2x^3 + 3x^2 - 4$.

position vector—A vector from the origin to a particular point.

potential energy—The energy held by an object under the influence of an external force, particularly gravity, a stretched spring, magnetic fields, etc. Generally, the amount of work a system is capable of doing.

power—A number obtained by multiplying a value by itself a certain number of times, so 8 is the third power of 2, denoted 2^3 . More generally, it can be defined as the power function mapping pairs of numbers to either a real or a complex number.

power—The amount of work done (energy released or used) by a system or machine over time.

prime factors—For two natural numbers a and b , a prime number that is a factor of both a and b .

prime number—A natural number greater than 1 with no factors other than itself or 1.

principle of relativity—The principle that all physical calculations should be independent of the reference frame within which they are calculated.

product—The result of multiplying two values.

projectile—An object moving under gravity.

projection—A map that reduces the number of dimensions of a space, such as mapping a sphere to a circle.

projection plane—The virtual plane in space representing the position of the image you are looking at through a particular camera.

proportional—Two values x and y are proportional, or “in proportion,” if there is some constant of proportionality k such that $x = ky$ for all valid values of x and y ; compare this with *inversely proportional*.

quadrant—An area of the Cartesian plane such that all the points are of the same sign in each of their coordinates.

quadratic function—A polynomial of degree 2.

quadrilateral—A four-sided polygon.

quadtree—A partitioning tree in 2-D space, in which objects are divided according to their position within successively smaller squares (in 3-D, an octree).

quaternion—A special kind of 4-D vector, the vector equivalent of a complex number.

quotient—Either the result of dividing one number by another (see *fraction*) or the integer part of this division (see *modulo*).

radians—A unit of angle, defined as $\frac{180^\circ}{\pi}$ of a circle.

radius—The distance of a point on the circumference of a circle from its center.

radix point—In column number notation, the dot that signifies the end of the integer part of the number.

range—The set of values returned by a function.

rate of change—The amount by which a value changes over time, equivalent to the derivative with respect to time.

rational numbers—The set of numbers that can be represented as a fraction of two integers, denoted by the symbol \mathbb{Q} .

ray—A line in 3-D space from a given point to infinity.

real numbers—The set of numbers that can be represented on an infinitely precise number line, denoted by the symbol \mathbb{R} .

reciprocal—For any number n , the fraction $\frac{1}{n}$.

rectilinear—Of a system of lines, means that any two are either parallel or perpendicular.

recursive function—An algorithm that works by calculating each step in terms of the same algorithm applied to a simpler case, until it finds a case simple enough to solve directly; compare this with *iterative function*.

reductio ad absurdum—A process of deduction where you prove something to be true by assuming it is false and deducing a contradiction: “if x were true then y would be true, but you know y is false therefore x must be false.”

reference frame—In a physical situation, the basis of the physical space along with a reference velocity used to define all other calculations; see *principle of relativity*.

reflection—A transformation that moves each point to another point opposite it in some mirror line or plane.

reflex angle—An angle greater than two right angles.

refraction—A phenomenon where a wave changes its direction of travel as a result of changing its speed.

remainder—When dividing one natural number n by another, m , the value r such that $0 \leq r \leq m$ and $n = am + r$ for some natural number a .

resonance—A phenomenon where an oscillation gradually increases in amplitude, as a result of an oscillating force of the same frequency.

rhombus—A parallelogram whose sides are all of equal length.

right angle—An angle of 90 degrees or $\frac{\pi}{2}$ radians.

right-angled triangle—A triangle with one of its angles being a right angle.

rigid-body transformation—A transformation which preserves angles between lines.

root—Of a function f , a value x such that $f(x) = 0$.

root node—Of a tree, the node with no parent.

rotation—A transformation that moves each point by some angle around a particular center.

saddle point—Of a surface, a point which is a local maximum in one direction and a local minimum in another.

scalar—A value that is not a vector, that is, it has no direction; alternatively, a vector of dimension 1.

scalar product—The sum of the pairwise products of the components of two vectors, written with a dot as $\mathbf{u} \cdot \mathbf{v}$: for example, the scalar product of the vectors $(2 \ 3)$ and $(1 \ -2)$ is the sum $2 \times 1 + 3 \times (-2)$. (Also called the “dot product.”)

scale—A transformation that moves each point to some multiple of its vector relative to some reference point.

search space—The set of all potential solutions to a problem.

search tree—A tree representing all the possible moves in a game, or equivalent for similar problems, which some algorithm is sought to solve.

set—A collection of objects (“elements”), which may be finite (the set $\{1,3,5,7\}$), countable (the set of all odd numbers), or uncountable (the set of all numbers between 0 and 1).

shear—A transformation that moves each point by a multiple of a vector, parallel to some reference line or plane, proportionally to its distance from this line or plane.

signed—Of a value, indicates that it may be positive or negative (for example, the signed distance of a point from a line is positive in one direction, negative in the other).

similar—Of two shapes, says that they are identical except for a scale transformation.

simple harmonic motion—A sinusoidal oscillation, generally associated with the motion of a stretched spring; compare this with *damped harmonic motion*.

simplification—The process of manipulating a function or statement algebraically to put it into a more tractable form.

simultaneous equations—Two or more equations in the same unknowns, which are all considered to hold true for the same set of solutions.

sinusoidal—Having the form of the function $A \sin(\omega x + c)$, where A is the amplitude, ω is the frequency, and c is the phase.

specular reflection—Light reflected from a surface directly, such that the angle of incidence equals the angle of reflection (mirror reflection); compare this with *diffuse reflection*.

speed—The distance traveled by a moving object over time.

sphere—The locus of a point in (usually 3-D) space at a constant distance from a particular center.

spline—A curve defined by some parametric function.

spring—An object that can extend, experiencing a tension as a result, and may also be compressed; see *extension, coefficient of elasticity*.

square—Either a planar shape with four sides of equal length and four right angles, or the result of multiplying a number by itself.

square matrix—A matrix that has the same number of rows as columns.

square root—The inverse of the square function x^2 : the square root of x is a number that when multiplied by itself gives the answer x .

stable strategy—In a game, a set of strategies for the players such that if any single player changes strategy, they do worse than before.

standard deviation—A measure of the spread of a set of values, defined as the square root of the mean squared distance of the values from their own mean.

statement—A numerical “sentence” relating some values to one another.

static friction—The friction between two surfaces exerting a force on one another but not moving relatively to each other; compare this with *kinetic friction*.

substitution—The process of evaluating a function for a particular set of values of its arguments.

subtend—If you draw a triangle from some point P to two points on the perimeter of an object, the angle at P for the largest possible such triangle is said to be the angle subtended by the object at P . (P is often an observer of the object.)

sum—The result of adding two values.

surface—A set of connected points in space definable by a pair of continuous parameters.

surface of rotation—For some function $f(x)$, the locus of a point in 3-D space whose perpendicular distance from the x -axis is equal to $f(x)$ at any x -coordinate.

tangent—A straight line (or plane) that touches some curve or surface at a particular point without passing through it. (In other words, it has the same gradient as the curve at that point.)

tends to a limit—Of a sequence of values a_1, a_2, \dots , this says that there is some value x such that for any small number d , you can find a number N such that for every $n > N$, $|a_n - x| < d$.

tension—The force in an object being pulled from each end, such as a string or spring.

tensor—A generalization of the concepts of vector, matrix, and scalar; an array of values in one or more dimensions.

term—An element of an expression consisting only of a constant value multiplied by some combination of variables, such as $5xy^2$.

terminal velocity—The maximum downward speed of a falling object as a result of the air resistance, or generally of any object experiencing drag or friction.

texel—A point of a texture map.

texture map—An image map describing the reaction of a surface to light at particular points.

topology—The study of the shapes of objects without reference to distance, using only notions of connectedness, holes, twists, etc., hence, these properties of an object.

torque—The angular equivalent of force, defined as force \times perpendicular distance from the axis of rotation.

torus—A 3-D shape, the locus of a point whose perpendicular distance from a particular circle is constant.

transform—A special kind of 4×4 matrix representing an affine transformation of homogeneous coordinates.

transformation—A map from one set of points to another in some space.

translation—A transformation that adds a constant vector to each point in the space.

transpose—Of a matrix \mathbf{M} , describes the matrix whose rows are the columns of \mathbf{M} and vice versa.

trapezium—A quadrilateral with a pair of parallel sides.

tree—A graph in which each node has exactly one parent (except the root node) and zero or more children, and with no loops.

trigonometric functions—The functions relating the angles in a right-angled triangle to the lengths of its sides.

trigonometric identities—Formulas relating the various trigonometric functions to one another.

trigonometry—The study of the properties of triangles.

turning point—A point on the graph of some function where its derivative (and all partial derivatives for a surface) is zero. May be a maximum, minimum, point of inflection, or saddle point.

uncountable set—A set with an infinite number of elements that cannot be listed even in infinite time.

unit vector—A vector with a magnitude of 1.

unknown—An element in an equation or other statement whose value is to be found by presuming the statement to be true.

utility function—A method for evaluating a particular solution to a search problem when using for example, a genetic algorithm.

variable—An element in a function that can be replaced throughout by some value from the domain in order to return some other value (also called an “argument”); also a generic term used to cover constants, parameters, and unknowns.

vector—An ordered set of two or more real numbers (or other values more generally) that can be added together by adding their components, and can be multiplied by a scalar.

vector product—The 3-D vector formed by combining two other such vectors such that its magnitude is equal to the product of their magnitudes with the sine of the angle between them, and its direction is perpendicular to both. (Also called the “cross product.”)

velocity—The vector traveled by a moving object over time; the vector equivalent of speed.

vertex—A corner of a shape. In a graph, equivalent to a node (pl. vertices).

view frustum—The volume of space visible from a particular camera.

viewport—The area of the projection plane visible on the screen.

visible light—Light whose wavelength lies within the range detectable by the human eye (approximately 10^{-7}).

wave—A phenomenon where a group of coupled oscillators transfer energy from one place to another.

waveform—The shape of the graph of a particular wave measuring distance from equilibrium through space.

wavefront—The surface in space representing equivalent points in the oscillation of a wave as it travels.

wavelength—The distance between successive wavefronts of a wave.

weight—The force acting on an object as a result of gravity, proportional to its mass.

weighted sum—Of a set of values x_1, x_2, \dots , the result of adding together the values multiplied by some predefined values w_1, w_2, \dots , in other words, the sum $\sum_i w_i x_i$.

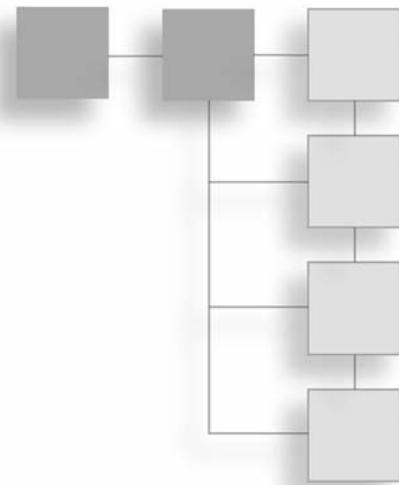
work—The energy given up by an object to induce movement in some other object.

zero-sum game—A game in which players compete for “money,” where at each stage, the total amount of “money” is constant.

This page intentionally left blank

APPENDIX B

CODE REFERENCES



The code samples used in this book are based on the Lingo scripting language. Along with JavaScript, Lingo is a scripting language supported by Adobe Director. Pseudocode in general usually designates stripped-down algorithms filled out with natural language sentences, such as *set p to the nearest point to q*, or *find the nearest point to q*. The few points made in this appendix are intended largely to orient you to the data types and language features characteristic of Lingo and so appear in the pseudocode.

Data Types

Lingo is a loosely typed language, meaning that you don't need to specify what kind of data is held in a particular variable or returned by a particular function. It's quite easy to switch between most of the following data types:

- **Boolean:** (TRUE or FALSE); essentially identical to the numbers 1 and 0.
- **Integer:** A standard 32-bit integer.
- **Float:** A single-precision floating-point number.
- **String:** A string of ASCII characters; double-byte characters can be used if specified.
- **Symbol:** A data type peculiar to Lingo—in a sense, the symbol is to a string what a float is to an integer. Symbols are represented by the hash sign and are used for “labels” like #positive. They’re just like strings but can have no spaces or non-alphanumeric characters and are not case-sensitive.

- **List:** Lists are the equivalent of “linked lists” in other languages—an array with no defined length. Lists are actually objects and have a number of methods for setting, deleting, or searching for values.
- **Point/Vector:** Specialized lists of two and three floating-point values. The vector object has methods like `getNormalized()` and `dot()`. Similarly, there are other specialized list objects, such as `rect`, `transform`, `rgb`, `time`, and so on.
- **Object:** This covers everything else. As previously mentioned, most things in Lingo are objects.

The loosely typed nature of Lingo allows you to do things like this:

```
set number to 7
if number then
  return "yes"
else
  return "no"
end if
```

This returns “yes”, so Lingo is quite happy to treat a number as a Boolean for the sake of `if` statements, and operators, and so on.

Variables

There are three kinds of variables in Lingo:

- **Local variables:** These exist only during the running of a particular function (method; handler). These include function arguments, as well as any variable specified during the course of the function. Because they are local, there is no need to worry about naming conflicts. In this book, I’ve tried to use only local variables unless it makes things harder to read.
- **Property variables:** These are held by a particular object. Each instance of an object can have its own values for these variables, for example, the standard sprite object has the properties `loc`, `width`, `height`, and so on. Some of these properties can be set directly, but others can’t; one of the drawbacks of Lingo is that it is not possible to create either private or read-only properties in a user-defined object. In a script object, you create a property variable by declaring it, usually at the beginning of the script, by saying something like `property pSize`. An object property is accessed using dot syntax (`object.propertyName`) or verbose syntax (the `propertyName` of `object`). In the example code, the prefix letter `p` is used to name property variables, as in `pName`, `pMember`.

- **Global variables:** These have the scope of the whole code, and can be accessed at any time. Although there's nothing wrong with using global variables, many see it as a sign of poor program organization.

Operators

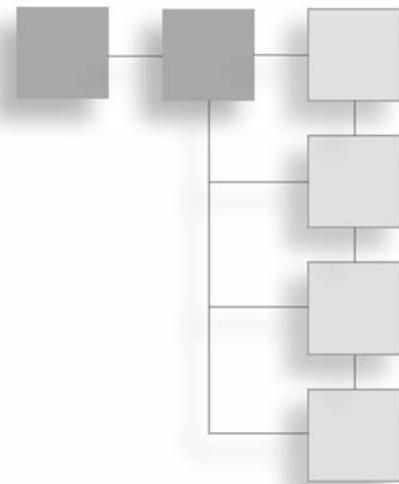
The code in this book uses only the basic operators `+`, `-`, `*`, `/`, and `=`. It does not make use of the increment operator `++`, for example, or the distinction between “equals” (`==`) and “set to” (`=`). So the equivalent of `a += b` could be any of the following (from most verbose to most succinct):

```
set a to a+b  
set a = a+b  
a = a+b
```

This page intentionally left blank

APPENDIX C

THE GREEK ALPHABET



Greek letters are often used in mathematics, and it's useful to know how to read them out loud. Many of them tend to appear in standard contexts, some of which are listed here. However, some are rarely used because they look too much like Roman letters.

TABLE C.1 Greek Letters and Common Mathematical Uses

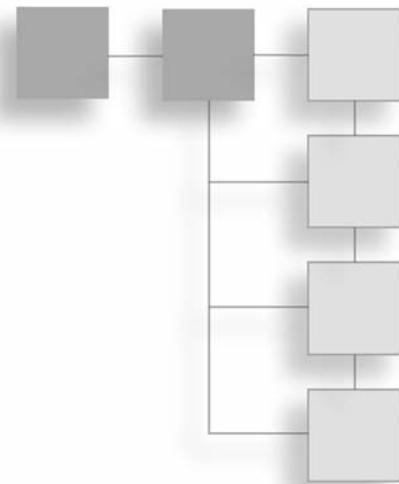
Greek Letter	English	Usage
α	Alpha	Often symbolizes an angle.
β	Beta	Often symbolizes an angle.
γ	Gamma	Often symbolizes an angle.
Δ	Delta	An infinitesimally small number, especially in calculus.
ϵ	Epsilon	An infinitesimally small number, especially in functional analysis.
ζ	Zeta	Occasionally symbolizes an angle.
η	Eta	Occasionally symbolizes an angle.
θ	Theta	Usually symbolizes an angle.
ι	Iota	Rarely used.
κ	Kappa	Rarely used.
λ	Lambda	The symbol for wavelength, also used to symbolize a ratio or proportion, especially eigenvalues.
μ	Mu	The symbol for “micro,” or a one-millionth-part; also the symbol for various material constants such as the coefficient of friction.

TABLE C.1 (continued)

Greek Letter	English	Usage
ν	Nu	Rarely used.
ξ	Xi	Sometimes symbolizes an angle.
ο	Omicron	Rarely used.
π	Pi	The constant value 3.14159. . ., the ratio of the circumference of a circle to its diameter.
ρ	Rho	The symbol for pressure, and used for some material constants.
σ	Sigma	Used for some material constants.
τ	Tau	Rarely used.
υ	Upsilon	Rarely used.
φ	Phi	Often symbolizes an angle or frequency; also the constant 1.618. . ., the golden ratio.
χ	Chi	Sometimes symbolizes an angle.
ψ	Psi	Sometimes symbolizes an angle or frequency.
ω	Omega	Often used for the frequency of an oscillation or rotation, and a number of other physical properties.

APPENDIX D

LEARNING RESOURCES



This appendix covers sources that might be useful as you extend your work with mathematics and programming. The sources named are but a few among many, and the areas covered concentrate on general math, collision detection, mazes, and game physics.

General Mathematics

There is a wealth of mathematical resources on the Internet, far too many to list and mostly free. One of the most recent additions to online learning is www.khanacademy.org/, which has been funded in part by the Gates Foundation. This site is certainly one of the most promising self-teaching sites on mathematics ever created and features excellent videos on everything from basic algebra to linear algebra and differential equations, and it has the advantage of featuring material that is of consistently high quality.

Another site is MathWorld (<http://mathworld.wolfram.com>), and the related ScienceWorld (<http://scienceworld.wolfram.com>), both hosted by Wolfram, the creator of the excellent mathematical program Mathematica. These sites include user submissions that vary in quality. They're also very technical, without much in the way of detailed explanation. For general reference, Wikipedia (<http://en.wikipedia.org>) remains fairly reliable, as does HowStuffWorks (www.howstuffworks.com). A good place to go if you want to ask math-related questions is Math Forum (<http://mathforum.org>), the home of "Ask Dr. Math."

Many sites are devoted to games computing, particularly GameDev.net (www.gamedev.net) and GamaSutra (www.gamasutra.com), both of which tend to focus on 3-D techniques, mostly for the more advanced programmers. A moderated and very focused mailing list for hardcore game enthusiasts is GDAlgorithms-List (<http://lists.sourceforge.net/lists/listinfo/gdalgorithms-list>). Also for more advanced math, MIT has rather wonderfully released many lectures and course notes in their MIT Open Courseware project: a list of math topics can be found at <http://ocw.mit.edu/index.htm>.

A good start for ActionScript/JavaScript programming using basic math is Macromedia Flash Professional 8 Game Development (Charles River Media, 2006). This book is basic enough that recent updates in the software do not affect its value. More elementary math is covered in a variety of sites, mostly aimed at children but as a result usually easy to follow! One large resource is the BBC Education Web site, including an area devoted to the 16+ age group, which is well written and thorough: www.bbc.co.uk/schools/16-maths.shtml.

Among books, suggestions might begin with *Beginning Math and Physics for Game Programmers* (Wendy Stahler et al., New Riders, 2004). It doesn't go quite as far as this volume, but it covers much of the same early ground. A more advanced book is *3D Math Primer for Graphics and Game Development* by Fletcher Dunn and Ian Parberry (Jones & Bartlett, 2002). To relate mathematics to physics, see *Physics for Game Developers* (David Bourg, O'Reilly, 2001), which provides a good treatment of how to model physical simulations in terms of forces. At a more advanced level, see *Essential Mathematics for Games and Interactive Applications, 2nd Edition* (Morgan Kaufmann, 2008) by James M. Van Verth and Lars M. Bishop.

Specialized Resources

In this section, the topics concern specific areas dealt with in the text. These are not the only areas of specialization in game development, but they are the areas the book most readily prepares you to investigate.

Collision Detection

There are a number of reliable books on collision detection. For example, Christer Ericson's *Real-Time Collision Detection* (Morgan Kaufmann Series in Interactive 3-D Technology, 2005) provides an excellent, if technical, view of the topic. A book that can be described along roughly the same lines is Gino Van Den Bergen's *Collision Detection in Interactive 3D Environments* (Morgan Kauffman, 2003). David Eberly's *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics, 2nd Edition*

(Morgan Kauffman, 2006) is more or less a classic in the area and includes a large section on the topic. Chris Hecker has also written a very readable and accessible series of tutorials on rigid body dynamics, including source code, which is available at Definition Six www.d6.com/users/hecker/dynamics.htm#articles.

3-D Engines and Geometry

In contrast to the previous section, the number of resources on 3-D topics is enormous, which is one of the reasons that the subject is dealt with relatively briefly here. Most of them have a large focus on graphics issues, but an excellent mathematical treatment is to be found in *Mathematics for 3D Game Programming and Computer Graphics, Third Edition* (Eric Lengyel, Course Technology PTR, 2011), although its scope is quite narrow. Several other titles already mentioned focus on 3-D topics too.

Game Physics

One book that covers the specifics of pool games is *The Illustrated Principles of Pool and Billiards* (David Alciatore, Sterling, 2004). This book is supported by www.engr.colostate.edu/~dga/pool/technical_proofs/index.html. For a broader view of the topic of game physics, see Dave Eberly's *Game Physics, 2nd Edition* (Morgan Kaufmann, 2010).

Mazes, Searches, and AI

Several helpful books are available on AI and associated topics in video games. Among these are Guy Lechy-Thomson, *AI and Artificial Life in Video Games* (Charles River Media, 2008), Neil Kirby, *Introduction to Game AI* (Course Technology PTR, 2010), and if you feel like challenging yourself, Dave Mark's *Behavioral Mathematics for Game AI* (Course Technology PTR, 2009). An older but more accessible book, by Mat Buckland, is *Programming Game AI by Example* (Jones & Bartlett, 2004). Another book, revised since its first release and often considered a classic is Ian Millington's *Artificial Intelligence for Games, 2nd Edition* (Morgan Kaufmann, 2009). Generally, the web provides a plethora of sites that address various types of searches used in video games. It is probably best, in this respect, to query "search algorithm" qualified by the following terms: binary, string, harmony, greedy, pattern, skip, binary, tree, and neural. These are just a few possibilities. For a comprehensive introduction to algorithms that is often seen in college courses, see Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to Algorithms, Third Edition* (MIT Press, 2009). For a site that allows you to see the current lay of the land respecting mazes, see Think Labyrinth! at www.astrolog.org/labyrnth.htm.

This page intentionally left blank

APPENDIX E

ANSWERS TO EXERCISES



There isn't space in this book to provide code listings for all exercises. The answers provided here are more along the lines of hints and suggestions, with some brief code snippets.

Code samples for the chapters in the book are available on the book's companion website at www.coursePTR.com/downloads. Examples are provided, implemented in Lingo.

Exercise 1.1 To construct the `convertBase(NumberString, Base1, Base2)` function, the easiest way to do this (although not the most efficient) is to create one function that converts the string from base 1 to a number, and then another that converts it from a number into base 2. You can use the functions in the text for this, but look at `base()` and `fromBase()` in `floats`.

Exercise 1.2 Floating-point numbers. This is fairly easy until you try to deal with division. The trick is to implement a binary version of the long division algorithm: at each stage you have a current “remainder,” to which you append digits of the dividend (the number being divided) until it is greater than or equal to the divisor, at which point you subtract off the divisor, calculate the new remainder, and continue until you reach the end of the dividend. In `floats`, you'll find the `mantissaExponent()` function, which converts a number into an IEEE-style float.

Exercise 2.1 Text scrollbar. The wording of the exercise was left deliberately vague about whether your scrollbar should be calculated according to the size of the text image, or according to the number of characters in it—that is, whether the scrollbar ratio should be “pixels/character” or “pixels/pixel.” The second is more standard (and much easier): try working on the first and see how it looks. You need to calculate how many characters are visible on the screen at any one time, in order to work out the furthest extent of your scrollbar.

Exercise 2.2 To construct the `compoundinterest(amount, percentage, years)` function, the key is to multiply the amount by $(100 + \text{percentage})/100$ for each year.

Exercise 2.3 Slide rule function. This was a rather fancy way of saying “write a function that multiplies two numbers by summing their logarithms.” So you’re just going to use

```
return exp(ln(a)+ln(b))
```

Of course, the graphical element is just padding, but is still more complicated than this part.

Exercise 3.1, 3.2, 3.3 Solving equations. These functions are part of the same overall concept. The resource files for the book contain a rather extensive implementation of the first, `substitute()`, which is used by a number of other functions, although it’s worth noting that a simpler method in the function `calculateValue()`, which uses the Lingo keyword `value`, works just fine for expressions which are written in a computer-readable form. The trick to this, and all of these examples, is to use recursion extensively. For example, when simplifying an expression, you can simplify each part of it separately, and to do that you can simplify its subsidiary parts, and so on, until it’s as simple as you can make it.

Exercise 4.1 `solvetriangle(triangle)`. Mostly this is a simple exercise in bookkeeping: there are rather a lot of different combinations of data that need to be dealt with separately. The simplest way is to try applying each of the rules (sine, cosine) in succession until you can’t apply any more.

Exercise 4.2 `rotatetofollow(triangle,point)`. The simplest way to do this is to use some of the functions you create later in the book, such as `rotateVector()`. If you’re trying to do it without having read Chapter 5 and beyond and you’re struggling, then read on and it should become clearer. In a nutshell, though, you need to calculate the angles made by the lines from the centrum to the triangle vertices, as well as the angle from the centrum to point, calculate the difference between the two, and then recalculate the positions of the triangle positions using `sin()` and `cos()`.

Exercise 5.1 Vector drawing. There isn't really an “answer” to this as such. It's just a suggestion for something to try, but you should take a look at the `createA()` function in Chapter 5 (vectors) and use that as your inspiration. There's very little limit to what you can do—if you get confident, try doing the same with 3-D images.

Exercise 5.2 calculateTrajectory(`oldPosition`, `newPosition`, `speed`). Hopefully this should be self-explanatory. It's something that you should always be doing in any working example.

Exercise 6.1 drawDifferentialEquation(`function`). If you find this difficult, then don't be afraid: creating the one in the figure was hard work, as you can see from the solution code in Chapter 6. Having said that, most of the complexity of this matches the similar work done on other graphs, namely calculating an appropriate scale, working out how best to label the axes, and so on. The main meat of the function is just these lines:

```
d=calculate2DValue(functionToDraw,pt[1], pt[2])
if d#=infinite then
    p1=0
    p2=yspacing
else if d<>#undefined then
    p=point(1.0,d)
    len=sqrt(1.0+d*d)
    p=p/len
    p1=p[1]*xspacing
    p2=p[2]*yspacing
else
    next repeat
end if
```

Here, you are at a particular point, you draw a line from that point, then start again from the end of that line. The hardest part is creating a reasonable spread of lines, which is solved by the rather cheeky expedient of simply choosing a number of random start points and seeing where they take you.

Exercise 6.2 secantMethod(). This is a fairly simple extension of the examples in the chapter. This function is designed to be as fast as possible, which is done by caching as many reference values as you can make use of.

Exercise 7.1 javelin(`throwAngle`, `throwSpeed`, `time`). The key trick is to calculate the velocity vector at each stage, and then rotate the javelin to follow it. Notice that the code also includes some simple collision-detection mechanisms to determine when the javelin reaches the edge of the screen (although it's allowed to go off the top).

Exercise 7.2 aimCannon(`cannonLength`, `muzzleSpeed`, `aimPoint`). This exercise was included as something of a trick, because it's not at all an easy calculation. Here is how you might work through it:

Suppose the cannon has length l and speed v , and you're trying to hit the point with position vector \mathbf{p} . The ball leaves the cannon from the point with speed v , so at time t its position is

$$\left((l + tv) \cos \theta, (l + tv) \sin \theta - \frac{gt^2}{2} \right)$$

So you are solving

$$(l + tv) \cos \theta = p_1$$

$$(l + tv) \sin \theta - \frac{gt^2}{2} = p_2$$

Now, there are a number of possible approaches to this. It is solvable directly (substituting for $\sin \theta$ in the second equation yields a quartic equation in t , which can be solved algebraically), but it's really easier to use an approximation method to approach the solution iteratively.

You can start by creating a function that tests whether the current aim is high or low (by working out the ball's y position when its x position is p_1). This function should return 1 for high, -1 for low, and 0 for a hit or close to it. Notice first of all that if you aim the cannon directly at the target point, it will always fall short. If you use this angle as your baseline, then work up in increments until you find a firing angle that is aiming high. You can then use a simple binary approximation method to find the solution.

Exercise 7.3 fireCannon(`massOfBall`, `massOfCannon`, `energy`). This should be a straightforward application of the formulas in the chapter.

Exercise 8.1 pointParallelogramCollision(`pt`, `parr`, `tm`). The chapter suggests using a skew transformation to turn the parallelogram into a normal rectangle. If you would prefer to solve the problem directly, you can represent it as a collision with four separate walls: you can work out which two walls are potential collision material by finding the dot product of their normals with the particle velocity, and then use the usual methods to determine which of them collide.

Exercise 8.2 Inner collisions. Generally, inner collisions are simpler to deal with than outer ones. For example, a rectangle inside a circle can only collide at its vertices, while a circle inside a rectangle can never collide with a vertex, and a rectangle inside a rectangle can only collide at a vertex if they are axis-aligned.

Exercise 9.1 `checkCollision()`. This is a tricky thing to do in abstract. You might find that without creating a generic collision detection environment, you can't really test how well it's working. But the version in the chapter includes quite a lot of detail, so keep trying.

Exercise 9.2 Newton's Cradle. Depending on how simplified you want this to be, it can be either very easy or extremely difficult. The goal in this context is to keep it simple.

Exercise 10.1 `splitPolygon(poly)`. As suggested in the chapter, this is best done recursively: choose three adjacent vertices, check if the third line of their triangle intersects with any others in the shape, and if so, split it off.

Exercise 10.2 `smoothNormals(collisionMap)`. This is found in the chapter's discussion of object modeling and collisions. At each step, the function checks through all the pixels of the collision map, and for any edge pixel it finds any adjacent pixels and sets the normal to the average of adjacent pixels. After about four iterations, you get a very close fit to the surface.

Exercise 11.1 Checking for a pocketed ball. If the ball passes between the pocket jaws, it will intersect the circle defining the pocket, so this should be foolproof, although it never seems very elegant. Another method would be to check against a line drawn along the pocket entrance.

Exercise 11.2 Ball preview. This is easier to do than you might think. The trick is to perform a regular collision detection, but with an extremely high "velocity." When you've found your first collision, you can calculate its resolution and draw the result into your image.

Exercise 12.1 Worlds in collision. As noted in the chapter, the simulation is only of limited success because of problems of scale. Nevertheless, it works as far as it goes.

Exercise 13.1 `resolveCushionCollision(obj1, obj2, normal, moment, slow)`. Review the notes in the chapter.

Exercise 13.2 A moving, rotating square and a wall. This is a tricky problem but not insurmountable. Since you don't need to deal with the ends of the wall, the only possible collision is between the vertices of the square and the wall. You can calculate this to a precise degree of accuracy by an approximation method, but really this is a good example of a case where it's easiest just to calculate whether any vertex of the square starts on one side of the wall and ends up on the other. The result will be close enough for any common purpose.

Exercise 14.1 applyFriction(velocity, topSpin, radius, mass, muK, muS, time). Although fairly straightforward, this is something of a drop in the ocean where implementing spin fully in the pool game is concerned. The real difficulty is dealing with how to calculate the topspin in the first place. As soon as the ball changes direction, it is spinning at a strange angle compared to the direction of travel, and this complicates things quite a bit. Your pool simulation cheats: you might like to think about how to do things more realistically.

Exercise 15.1 Rockets. This one should be pure book work.

Exercise 16.1 Springs. You'll find that the versions quite quickly start to behave differently as a result of calculation errors. The hardest thing to get right is the elastic limit, and this code doesn't quite achieve it.

Exercise 17.1 Drawing a 3-D cube and culling the back faces. The way you do this is quite simple. You don't draw any face whose outward normal is pointing away from you. Such a face has a positive dot product with the vector from the camera to the face. In a wireframe drawing, you can work on a vertex-by-vertex basis: a vertex is culled if all faces sharing that vertex are facing backward. Then you only draw lines both of whose vertices are visible. The next step in this code would be to create z -ordering—that is, to draw objects nearer to the camera in front of those farther away.

Exercise 18.1 Applying relative transforms. These can be found in Chapter 18. They should be straightforward enough.

Exercise 19.1 3-D collisions. The exercise didn't specify which collision to implement, as by now you should be used to the formula. At the very least, you should find spheres and planes reasonably simple.

Exercise 20.1 Texture maps. There's a lot involved in creating the mesh, but you only really need to look at the lines involving the `textureCoordinates` property.

Exercise 21.1 2-D NURBS. The implementation is fairly simple if you just follow the equations in the chapter.

Exercise 21.2 IKapproach(chain, target). The trick with this is to work backward: start by adjusting the most central limb to rotate it toward the target, then adjust the next one out, and so on until you get to the last bone.

Exercise 22.1 Complete drawBresenham(). This can be found in Chapter 22. It should be a simple extension of the function in the chapter.

Exercise 22.2 Car control. One of the great goals of pseudo-physics is, “How simple can this be and still feel complicated?” The aim here is to get the perfect balance between simplicity and complexity.

Exercise 23.1 box3DTileTopCollision(). This is a difficult problem. It is more difficult if you try it with a non-aligned box.

Exercise 24.1 Maze navigation. Consider what is involved in recursive backtracking for maze creation.

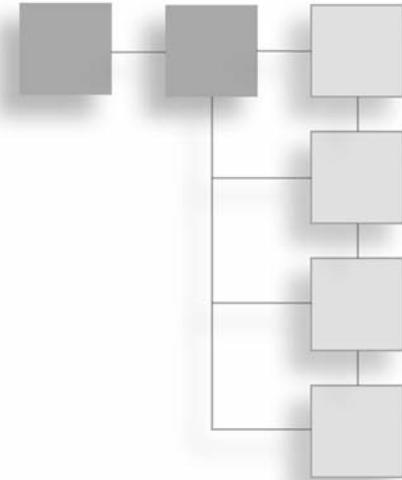
Exercise 25.1 Heads or tails. Few approaches improve over the one that is most basic.

Exercise 25.2 Boids. Boid rules are well-documented, so if you have trouble with this, you might want to do some research online. Try Craig Reynolds’ own site (www.red3d.com/cwr/boids) to start with; it’s well worth a visit.

Exercise 26.1 Eight queens genetic algorithm. The demo `geneticAlgorithm()` function shows one approach to this. This includes a system of varying radiation to speed things up a little. (It also demonstrates some examples of how you can keep tabs on your program’s performance when running this kind of extended search.)

This page intentionally left blank

INDEX



Numbers

0 (zero), representing, 15
1, rejecting as prime, 29
-1 e35 computation, 14
1's and 0's, using in binary system, 9–11
2-D world, creating, 517. *See also* TBG (tile-based games)
3-D, splines in, 475–476
3-D collision detection
 bounding spheres, 448
 boxes, 437–442, 448
 cans, 442–448
 cones, 446
 cylinders, 443–447
 ellipsoids, 435–437, 448
 footballs, 435–437
 meshes, 448–449
 moving sphere and wall, 433–434
 point of contact, 434
 resolving, 449
 spheres, 432–433
 spheres and moving point, 434
 See also collision detection
3-D modeling
 NURBS, 476–480
 sine and cosine surfaces, 480–481
 splines, 475–476
 surfaces of rotation, 474–475
 tessellation, 481–483
3-D platform games, 524–528
 `box3DTileTopCollision()` function, 525–527
 collision detection, 525
3-D RGB vectors, explained, 453

3-D space

 defining lines in, 394
 defining planes in, 394–395
3-D vectors, 393–394
 cross product, 395–399
 defining planes, 395
 homogeneous coordinates, 399–402
 left-hand axis, 394
 orthographic projections, 408–409
 perspective, 407–408
 raycasting, 409–414
 rendering, 402–406
 See also vectors
4 × 4 matrix, using, 418–421
10.5 value computation, 14
32- vs. 64-bit machines, 10, 13. *See also* bits

A

`A*` Algorithm, 554–556, 592
AABB (axis-aligned bounding box), 270, 512
`abs()` function, 16
acceleration
 and deceleration, 184
 due to gravity, 186–188
 equations of motion under, 184–186
 vs. velocity, 185
`acos()` function, 103–104
actors and emergence
 boid model, 586
 bulletin board model, 584
 codelets, 586
 flocking behavior, 584–585
 high-level perception, 586
 slipnets, 586

- addFractions() function, 26
- addVectors() function, 127
- agents, 584
- AI (artificial intelligence)
 - actors and emergence, 585–586
 - back-game, 577
 - changing goals, 577–578
 - chess programs, 573–574
 - connectionists, 580
 - epiphenomena, 580
 - estimation functions, 574
 - goals and subgoals, 576–577
 - GOFAI, 576
 - heuristics, 576
 - knowledge base, 576
 - neural networks, 581–584
 - overview of, 559–560
 - parameterization, 574
 - pattern analysis, 578
 - scripts system, 578
 - strategy and tactics, 577
 - tactical, 572
 - Tic-Tac-Toe (bottom-up), 586
 - Tic-Tac-Toe (tactical approach), 575
 - Tic-Tac-Toe (top-down), 578–580
 - training neural networks, 584–585
 - training programs, 574–575
 - usual game, 577
- aimCannon() function, 197
- air resistance
 - calculating, 342
 - dependence on speed, 342
 - terminal velocity, 342
- algebra
 - constants, 54–55
 - cubic equations, 68–70
 - equations, 58–59
 - expressions, 55–56
 - factoring, 64–67
 - formulas, 59
 - functions, 56, 84–85
 - graphs, 77–83
 - inequalities, 59
 - parameters, 54
 - parametric curves and functions, 86–87
 - polynomials, 58
 - representations, 57
 - solving quadratic equations, 64–67
 - sqrt() function, 57
 - symbols, 57
 - terms, 55–56
 - variables, 54–55
- algorithms
 - A* Algorithm, 554–556, 592
 - Bresenham's Algorithm, 501–504, 510
 - computational complexity, 498–499
 - constant-time calculations, 499
 - Eller's Algorithm, 540–541
 - exponential time calculations, 498
 - greedy, 540
 - Kruskal's Algorithm, 540–541
 - logarithmic time calculations, 499
 - polynomial time calculations, 498
 - Prim's Algorithm, 539
 - See also* genetic algorithm; TBG (tile-based games)
- aliasing, dealing with, 465
- alpha-beta search, using, 572
- AND operator, 11
- angleCollideStatLine() function, 328–329
- angles
 - acute, 92
 - defined, 89
 - and degrees, 90–92
 - measuring, 90–91
 - obtuse, 92
 - reflex, 92
 - right, 91
 - rotating objects by, 112–113
 - rotations by, 115
- angular collisions, resolving, 331–334
- angular momentum, conserving, 317
- angular motion
 - angular collisions, 331–334
 - circle and moving line, 323–325
 - collisions between lines, 325–329
 - defined, 309
 - and friction, 343–346
 - rotating lines, 330–331
 - rotational kinetic energy, 318
 - spin, 316–318
 - spin applied to pool game, 334–335
 - spinning collisions, 319–322
 - tensors, 316
 - See also* levers
- angularCollisionLineCircle() function, 321
- animated surfaces
 - cloth and hair, 483–486
 - waters, 486–487
- anisotropic filtering, 468
- applyFriction() function, 347
- approximation methods
 - bracketing, 173–176
 - checks and balances, 179
 - gradient, 176–179

- Archimedes, 310
 $\arcsin()$ function, 103–104
 $\arctan()$ function, using, 104–105
 area
 calculating for rectangle, 92
 of circle, 93
 Aristotle, 298
 arithmetic
 clock, 33
 compound interest, 41
 cycling through data, 33–34
 debts and interest, 42–43
 e and $\exp()$, function, 45–46
 exponentials, 43–47
 Fabonacci Sequence, 38
 factors and factorization, 27–31
 fractions, 24–26
 GCD and Euclid's Algorithm, 30
 Golden Ratio, 37
 LCM (lowest common multiple), 30–31
 logarithms, 47–51
 modulo, 31–32
 percentages, 40–43
 powers, 44–45
 prime numbers, 28–29
 ranges of values, 35–38
 slider, 38–40
 arithmetic mean, explained, 114
 armature, finding, 268–273. *See also* shapes
 arrays, convention for, 479
 arrow model (figure), 422
 artificial intelligence (AI)
 actors and emergence, 585–586
 back-game, 577
 changing goals, 577–578
 chess programs, 573–574
 connectionists, 580
 epiphenomena, 580
 estimation functions, 574
 goals and subgoals, 576–577
 GOFAI, 576
 heuristics, 576
 knowledge base, 576
 neural networks, 581–584
 overview of, 559–560
 parameterization, 574
 pattern analysis, 578
 scripts system, 578
 strategy and tactics, 577
 tactical, 572
 Tic-Tac-Toe (bottom-up), 586
 Tic-Tac-Toe (tactical approach), 575
 Tic-Tac-Toe (top-down), 578–580
 training neural networks, 584–585
 training programs, 574–575
 usual game, 577
 As, drawing, 132
 $\sin()$ function, 103
 $\text{aStar}()$ function, 554–555
 $\tan()$ function, 103
 axis of reflection, explained, 115
 axis of rotation, explained, 397
 axis-aligned bounding box (AABB), 270, 512

B

- backspin, explained, 344
 ballistics, 183–190
 acceleration, 184
 acceleration due to gravity, 186–188
 deceleration, 184
 equations of motion, 184–186
 motion of cannonball, 188–190
 projectiles, 183, 189
 solving problems related to, 193–196
 balls
 colliding, 234–235
 hitting movable ball, 232–234
 hitting wall, 230–231
 incident vs. object, 232
 velocity of, 232
 barycentric coordinates (figure), 468
 $\text{barycentric}()$ function, 469
 base 2 system
 1's and 0's, 9–11
 bits, 9–10
 multiplication in, 11
 off symbol, 9
 on symbol, 9
 using, 9–11
 base notation
 designating slots, 8
 writing numbers in, 8
 base systems
 binary, 9
 creating, 6–7
 examples of, 8
 hexadecimal, 9
 $\text{baseStringToValue}()$ function, 7–8
 bespoke look-up table, using, 500
 Bezier curves and splines, 251–255
 BigInteger classes using, 19–20
 bilinear filtering (figure), 466
 binary computations, performing, 10–11
 binary digits, combining, 11
 binary space partitioning (BSP tree), 511

- binary system
 1's and 0's, 9–11
 bits, 9–10
 multiplication in, 11
 off symbol, 9
 on symbol, 9
 using, 9–11
 bisection method, using, 173
`bisectionMethod()` function, 174–175
 bits
 32 vs. 64, 10
 defined, 9
See also 32- vs. 64-bit machines
 blob collision map (figure), 248
`boardList()` function, 570
 bone animations
 inverse kinematics, 490–493
 joint constraints, 488–489
 kinematics, 489
 ragdolls, 488–489
 series of motions, 488
 systems, 487–488
 Boolean digits, combining, 11
 Boolean functions, 56
 bounding boxes
 axis-aligned, 270
 object-aligned, 270
 bounding shapes, finding, 268–273. *See also* shapes
 bounding spheres, collisions, 448
`boundingCircle()` function, 269
`box3DTileTopCollision()` function, 525–527, 529
 boxes
 colliding, 437–442, 448
 and moving points, 438–439
 multiple, 439–441
 and spheres, 441–442
 bracketing methods, 173–176
 Bresenham's Algorithm, 268, 501–504, 510
 BSP (binary space partitioning) tree, 511
 B-splines, 475–480. *See also* rational B-splines; splines
 building height, measuring, 109
 bump maps, 460–461
 bumps, assessing heights of, 410
 BVH (bounding volume hierarchy), 512–513
- C**
- `calculateDHParameters()` function, 380–381
`calculateNormals()` function, 264–266, 274
`calculatePosition()` function, 186
`calculateTrajectory()` function, 134
 calculations
 cheap vs. expensive, 497
 expensive, 499–500
 integer, 500–504
- calculus
 approximation methods, 173–179
 defined, 157
 differential equations, 170–172
 gradient of functions, 158–160
 integration, 168–169
 parametric equations, 167
 partial derivatives, 167
 solving linear ODEs, 172–173
See also differentiating
 cannon firing, energy transfer during, 193
 cannonball, motion of, 188–190, 307
 cans, colliding, 442–447
 Cartesian plane
 axes, 77
 coordinates, 77
 (figure), 77
 plotting points, 77–78
 reading points, 79
 x- and y-coordinates, 77
 Catmull-Rom curves, 252–253
`ceil()` function, 16–17
 center of gravity, explained, 114
 central projection, explained, 408
 centrifugal force, explained, 305
 centripetal force, explained, 305
 centrum, finding for triangle, 114
 chain rule, using in differentiation, 162–163
`checkCollision()` function, 238–240, 287
 chemical potential energy, 193
 chess case study
 constraints, 596
 functions, 596–597
 search function, 597–598
 search space, 595
 chess programs
 culling moves, 573
 estimation functions, 573–574
 heuristics, 576
 territory, 573
 training systems, 574–575
 chessboard (figure), 595
 circle
 area of, 93
 circumference, 90
 diameter, 90
 dividing into radians, 94
 drawing using `sin()` and `cos()`, 118
 perimeter, 90
 radius, 90, 93
`circleCircleCollision()` function, 208–209
`circleCircleInnerCollision()` function, 210
`circleCircleStraightCollision()` function, 207–208
`circleRectangleInnerCollision()` function, 228

- circles
 - collisions with rectangles, 226–227
 - described for collision detection, 202
 - inside circles, 209–210
 - moving at angle, 208–209
 - and moving lines, 323–325
 - moving on straight line, 206–208
 - moving with wall, 202–204
 - point of contact in collision, 210
 - and rotating line collision, 319–323
 - stationary and moving point, 205–206
 - `circleWallCollision()` function, 204, 206, 227
 - circular mazes (figures), 545–546
 - circular motion and SHM (simple harmonic motion), 367
 - Clarke, Arthur C., 306
 - `cleverClockAdd()` function, 34
 - clock arithmetic, 31–33
 - `clockwise()` function, 326
 - `clockwiseNormal()` function, 326–327
 - cloth
 - dashpot springs, 486
 - modeling for animation, 484–486
 - code optimization
 - binary-space partitioning, 511
 - BVH (bounding volume hierarchy), 512–513
 - cheap calculations, 497
 - computational complexity, 498–499
 - culling, 507–513
 - expensive calculations, 497
 - integer calculations, 500–504
 - look-up tables, 499–500
 - octrees, 508–510
 - quadtrees, 508–510
 - segregating space, 507–508
 - simplifying collisions, 504–506
 - simplifying motion, 506
 - code samples
 - data types, 629–630
 - operators, 631
 - variables, 630–631
 - codelets, defined, 586
 - coefficient
 - defined, 55
 - of elasticity, 362
 - representing, 66
 - of restitution, 333
 - collision detection
 - Bezier curves and splines, 251–255
 - bitmaps and vector shapes, 246–247
 - bounding shapes, 268–273
 - built-in solutions, 273
 - Catmull-Rom curves, 252–253
 - circle inside circle, 209–210
 - circles, 202
 - circles and rectangles, 226–227
 - `collisionMap()` function, 248–249
 - complex shapes, 246
 - convex and concave shapes, 255–256
 - defining complex shapes, 247–250
 - ellipses, 220–222, 226
 - ellipses and coordinates, 222–223
 - ellipses translation, 223–224
 - leading edge of complex shape, 259–263
 - between lines, 325–329
 - movable splines, 254–255
 - moving circle and wall, 202–204
 - moving circles, 206–209
 - parametric functions, 249–250
 - point of contact for circles, 210–211
 - point of contact for ellipses, 226
 - point of contact for rectangles, 219–220
 - points inside shapes, 256–259
 - rectangles, 215–219
 - rectangles and squares, 211–213
 - stationary circle and moving point, 205–206
 - stationary ellipse and moving point, 224–225
 - stationary rectangle and moving point, 213–215
 - with sticky surface, 335
- See also* 3-D collision detection; spinning collisions
- collision halo, using, 268
- collision maps
 - associating tiles with, 522
 - for blob, 248
 - finding tangent to, 264
 - height maps, 263
 - using, 247, 263–268
- collision resolution
 - ball hitting movable ball, 232–234
 - ball hitting wall, 230–231
 - inelastic collisions, 235–238
 - moving balls, 234–235
- `collisionMap()` function, 248–249
- collisions
 - elastic, 230, 234
 - as recursive functions, 238–240
 - simplifying, 504–506
 - simultaneous, 241–242
- color elements
 - applying to surfaces, 456–459
 - diffuse color, 456
 - emissive color, 456
 - specular highlight, 457
- color values, processing, 266
- `colorAtPoint()` function, 469
- colors
 - 3-D RGB vectors, 453
 - emergence of, 452–453
 - complex numbers, 5
 - `component()` function, 140
 - `componentVector()` function, 140, 231

- composite number, explained, 28
 - compound interest, 41
 - computational complexity, 498–499
 - concave and convex shapes, 255–256
 - cones
 - (figure), 443
 - infinite, 444
 - and particles, 446
 - and spheres, 446
 - surface of rotation, 443
 - congruence, expressing, 31–32
 - conic sections, explained, 444
 - constants
 - described, 54
 - using, 55
 - constant-time calculations, 499
 - continuous momentum
 - conveyor belts, 355–357
 - rocket fuel, 357–358
 - convex and concave shapes, 255–256
 - convex polygon (figure), 247
 - convex polyhedron, explained, 437
 - conveyor belts, 355–357
 - Copernicus, Nicolaus, 303
 - `cos()` function
 - on circle, 117–118
 - inverse of, 104
 - using, 100–102
 - counting numbers, 4, 12
 - coupling
 - defined, 375
 - natural modes of motion, 375
 - out of phase springs, 375
 - `createA()` function, 132
 - `createBalls()` function, 282–283
 - critical angle, defined, 339
 - cross product
 - axis of rotation, 397
 - calculating, 395–396
 - explained, 395
 - properties of, 396
 - using, 397–399
 - crossword grid (figure), 590
 - crossword problem
 - recursive structure, 594
 - summarizing, 594
 - cubic equations, solving, 68–70, 72
 - cuboids, described, 437–438
 - `cueRotation()` function, 285–286
 - culling, 291–292
 - binary-space partitioning, 511
 - BVH (bounding volume hierarchy), 512–513
 - octrees, 508–510
 - partitioning trees, 507
 - quadtrees, 508–510
 - segregating space, 507–508
 - `currentPosition()` function, 135, 358
 - `curvedPath()` function, 136–137
 - curves
 - Catmull-Rom, 252–253
 - finding area under, 169
 - finding gradient of, 158
 - cylinders
 - as cones, 443
 - defined, 443
 - halfminor axis, 447
 - multiple, 447
 - and points, 444–446
 - projecting to planes, 447
 - and spheres, 444–446
 - surface of rotation, 443
 - cylindrical mapping (figure), 464
 - cylindrical maze (figure), 545
- D**
- Daedalus* software, 535
 - damped harmonic motion (DHM)
 - differential equations, 370
 - explained, 369
 - vs. SHM (simple harmonic motion), 370
 - damping
 - applying, 372
 - critical, 372–373
 - defined, 369
 - overdamping, 373
 - underdamping, 371–372
 - Darwin, Charles, 600
 - debts and interest, calculating, 42–43
 - decimal notation, 8
 - decimal point, defined, 12
 - `defineTable()` figures, 280–281
 - degrees and radians, converting between, 94
 - delta (δ), using to represent limits, 158
 - denominator, defined, 5
 - derivatives
 - functions with, 163
 - interpreting, 163–164
 - point of inflection, 164
 - turning point at x , 163
 - using in differentiation, 160–164
 - `detectCollisionWithWorld()` function, 518–521
 - determinant() function, 149
 - DHM (damped harmonic motion)
 - differential equations, 370
 - explained, 369
 - vs. SHM (simple harmonic motion), 370
 - diagonal functions, 84

difference of two squares, 68. *See also* square root
 differential equations, 170–173
 integration parameters, 171
 numeric plot (figure), 172
 numeric solutions, 170–173
 vs. vectors, 171
 differentiating
 exponential functions, 164–165
 functions, 160–164
 functions in variables, 167
 logarithms, 164–165
 trigonometric functions, 165–166
See also calculus
 differentiation
 chain rule, 162–163
 vs. integration, 168
 partial, 167
 diffuse reflection, 453, 457
 discriminant expression, using, 67, 69
 division, scalar, 124–126
 divisor, defined, 27
 Doppler effect, explained, 388
 Doppler shift, explained, 388
 dot products, using in 3-D space, 395
 double precision numbers, 15
 drag
 calculating, 342
 dependence on speed, 342
 terminal velocity, 342
 dragging, using raycasting in, 411–414
 drawBresenham() function, 503–504, 514
 drawEllipseByAxes() function, 223–224
 drawEllipseByFoci() function, 221–222
 drawEllipseFromMatrix() function, 223–224
 drawGraph() function, 80–83
 drawWorld() function, 517
 Dresher, Melvin, 562
 dropping, using raycasting in, 411–414

E

Earth
 circumference of, 50–51
 geo-stationary orbit, 306
 eigenvectors, right- vs. left-, 154
 Einstein, Albert, 201, 297
 elastic potential energy, 192
 elasticity, coefficient of, 362
 electrical energy, 193
 electromagnetism, explained, 452
 eliminating x, process of, 71
 elimination, solving equations by, 73–74
 eller() function, 541–542
 Eller's Algorithm, 540–541

ellipses
 collision, 226
 described for collision detection, 220
 describing using coordinates, 222–223
 drawing, 221
 point of contact, 226
 stationary and moving point, 224–225
 ellipsoidPlaneCollision() function, 436
 ellipsoids
 collisions, 448
 described, 435
 and moving planes, 435–436
 and moving points, 435–436
 multiple, 436–437
 spheroids, 435
 emission maps, 460
 energy
 chemical potential, 193
 conservation of, 193–196
 elastic potential, 192
 electrical, 193
 and friction, 340–341
 gravitational potential, 192
 heat, 192
 kinetic, 192–193
 potential, 193
 equations, 58–59
 balancing, 59–61
 cross-multiplying, 61
 eliminating x process, 71
 vs. functions, 58
 grouping like terms, 61
 multiplying out parentheses, 62
 negatives and substitution, 62–63
 parametric, 167
 redux list, 75
 removing fractions, 61
 simplification, 61–62
 solving, 63–64
 solving by elimination, 73–74
 solving by substitution, 70–72
 solving set of, 70–71
 solving simultaneous, 70–76
 solving systems of, 74–76
See also quadratic equations
 Eratosthenes' Sieve, 28
 escalator (figure), 356
 estimation functions, calculating, 574
 Euclid's Algorithm, 30
 Exercises
 3-D cube, 414
 aimCannon() function, 197
 applyFriction(), 347

Exercises (continued)

box3DTileTopCollision() function, 529
 calculateNormals() function, 274
 checkCollision() function, 242
 circleRectangleInnerCollision() function, 228
 compound interest, 51
 differential equation, 179
 drawBresenham(), 514
 fireCannon(), 197
 getRocketSpeed() function, 359
 IKapproach() function, 494
 javelin() function, 196
 moveBalls() function, 293
 planets moving under gravity, 308
 pointParallelogramCollision() function, 227–228
 pointRectangleCollision() function, 227–228
 rectangleCircleInnerCollision() function, 228
 rectangleRectangleInnerCollision() function, 228
 resolveCushionCollision() function, 335
 rotatetofollow() function, 118
 scrollbar functions, 51
 secant approximation method, 179
 secant method, 179
 shapes, 155
 simplify() function, 87
 slide rule function, 51
 smoothNormals(collisionMap) function, 274
 solve() function, 87
 solvetriangle() function, 118
 splitPolygon(poly) function, 274
 springs, 390
 substitute() function, 87
 velocity vector, 155
 writing functions, 20
 exp() function, 45–46, 164
 exponent, defined, 6
 exponential time calculations, 498
 exponentials
 calculating with, 44–45
 decay, 46–47
 defined, 6
 differentiating, 164–165
 exp() function, 45–46
 vs. logarithms, 47
 using, 46–47
 using logarithms with, 50
 expressions, 55–56
 discriminant, 67, 69
 factoring polynomials, 72
 vs. functions, 58
 quadratic, 65
 eyes, light detection in, 452

F

factoring, 64–65
 factors
 common, 64
 defined, 27
 and quadratics, 65–67
 Fermat's Last Theorem, 99
 Fibonacci Sequence, 38
 findEdges() function, 265
 fireCannon() function, 197
 fixed-point representation, 13
 floating-point representation, 13, 15, 18–19
 floats to integers, 16
 flocking behavior, 584–585
 Flood, Merrill, 562
 floor (n / m), finding, 32
 floor() function, 16–17, 56
 fluid resistance
 calculating, 342
 dependence on speed, 342
 terminal velocity, 342
 flywheel, behavior of, 316–317
 footballs, colliding, 435–437
 force
 centrifugal, 305
 centripetal, 305
 explained, 297–298
 of gravity, 302
 impulse, 301
 forceDueToSpring() function, 376–377
 formulas, 59
 Fourier analysis, applying to waves, 386
 fractions
 addFractions() function, 26
 addition of, 25–26
 cross-multiplying, 26
 defined, 5
 dividing, 24–25
 incompatible, 12
 as instructions, 24
 multiplying, 24
 as reciprocals, 5
 representing, 12, 24–25
 subtraction of, 25–26
 vulgar, 5
 See also percentages; ratios
 frame of reference, explained, 200
 friction
 air resistance, 341–342
 and angular motion, 343–346
 cause of, 338
 coefficient of, 338–340

critical angle, 339
 and energy, 340–341
 falling over (figure), 346
 on inclined plane (figure), 338
 in running (figure), 347
 and slipping, 345–347
 terminal velocity, 341–342
 wheels and grip, 343–345
frustum
 explained, 403
 planes in, 404
functions
`abs()`, 16
`addFractions()`, 26
`addVectors()`, 127
`aimCannon()`, 197
`angleCollLineStatLine()`, 328–329
`angularCollisionLineCircle()`, 321
`aStar()`, 554–555
 asymptotic, 85
`barycentric()`, 469
`baseStringToValue()`, 7–8
`bisectionMethod()`, 174–175
`boardList()`, 570
`Boolean`, 56
`boundingCircle()`, 269
`box3DTileTopCollision()`, 525–527, 529
`calculateDHMPatterns()`, 380–381
`calculateNormals()`, 264–266, 274
`calculatePosition()`, 186
`calculateTrajectory()`, 134
`ceil()`, 16–17
`checkCollision()`, 238, 287
 for chess case study, 596–597
`circleCircleCollision()`, 208–209
`circleCircleInnerCollision()`, 210
`circleCircleStraightCollision()`, 207–208
`circleRectangleInnerCollision()`, 228
`circleWallCollision()`, 204, 206, 227
`cleverClockAdd()`, 34
`clockwise()`, 326
`clockwiseNormal()`, 326–327
`collisionMap()`, 248–249
`colorAtPoint()`, 469
`component()`, 140
`componentVector()`, 140, 231
`cos()`, 104
`createA()`, 132
`createBalls()`, 282–283
`crossProduct()`, 396
 cubic, 68–70
`cueRotation()`, 285–286
`currentPosition()`, 135, 358
`curvedPath()`, 136–137
`defineTable()`, 280–281
 with derivatives, 163
`detectCollisionWithWorld()`, 518–521
`determinant()`, 149
 diagonal, 84
 differentiating, 160–164
 domain of, 56
`drawBresenham()`, 503–504, 514
`drawEllipseByAxes()`, 223–224
`drawEllipseByFoci()` function, 221–222
`drawEllipseFromMatrix()`, 223–224
`drawGraph()`, 80
`drawWorld()`, 517
`eller()`, 541–542
`ellipsoidPlaneCollision()`, 436
 vs. equations, 58
`exp()`, 45–46, 164
 vs. expressions, 58
`findEdges()`, 265
`fireCannon()`, 197
 floats to integers, 16
`floor()`, 16–17, 56
`forceDueToSpring()`, 376–377
`getOscillatorSpeed()`, 381–382
`getRocketSpeed()`, 358–359
 gradient of, 158–160
 graphics, 80
 horizontal, 84
`IKapproach()`, 494
`illumination()`, 459
`intersectionPoint()`, 143–144
`intersectionTime()`, 144–145, 204
 inverse, 57
`kruskal()`, 540
`leadingEdgeOfPolygon()`, 261–262
`leadingPointOfPolygon()`, 260–261
`lineOfBestFit()`, 272
`linePlaneIntersection()`, 399
`log()`, 164
`madPath()`, 138
`magnitude()`, 127
`makeMinimaxStrategy()`, 570
`makeTTTlist()`, 568–569
`makeTTTtree()`, 568–569
`mateOrganisms()`, 603
`matrixMultiply()`, 150–152
`mean()`, 272
`minimaxIteration()`, 571
`mod()`, 32
`modulate()`, 459
`moment()`, 317
`moveBalls()`, 287–290, 293
`moveCannonBall()`, 307
`multiplyConnected()`, 544

- functions (*continued*)
- multivalued, 57
 - `neuralNetStep()`, 582–583
 - `newtonRaphson()`, 178
 - `norm()`, 128
 - `numberToBaseString()`, 7
 - `NURBSbasisFunction()`, 478
 - one-to-one, 57
 - parabolic, 84–85
 - parametric, 249–250
 - parametric curves, 86–87
 - `particleEllipseCollision()`, 225
 - `particleHmapCollision()`, 267
 - `particleOnSpring()`, 378–380
 - `pi()`, 93
 - `planePlaneIntersection()`, 399
 - plotting on graphs, 79–83
 - `pointCircleCollision()`, 206, 227
 - `pointCmapIntersection()`, 267–268
 - `pointInsidePolygon()`, 258–259
 - `pointInsidePolygonIncomplete()`, 257–258
 - `pointInsideRectangle()`, 213
 - `pointOnRectangle()`, 213
 - `pointParallelogramCollision()`, 227–228
 - `pointPolygonCollision()`, 255–256
 - `pointRectangleCollision()`, 227
 - `pointRectangleIntersection()`, 214–215
 - `pointsToCheck()`, 217
 - polynomials, 58
 - `pos3DToScreenPos()`, 405–406
 - `readColor()`, 266
 - `rectangleCircleInnerCollision()`, 228
 - `rectangleRectangleAngledCollision()`, 218–219
 - `rectangleRectangleCollisionStraight()`, 216
 - `rectangleRectangleInnerCollision()`, 228
 - `rectangleRectangleInnerCollision()`, 228
 - `recursiveBacktrack()`, 539
 - `resolveAngularCollision()`, 333–334
 - `resolveCollisionEqualMass()`, 235, 283
 - `resolveCollisionFree1()`, 234–235
 - `resolveCushionCollision()`, 335
 - `resolveFixedCollision()`, 231
 - `resolveInelasticCollisionFixed()`, 237–238
 - `resolveInelasticCollisionFree()`, 237
 - `resultantForceOnObject()`, 340
 - `rotateToFollow()`, 118
 - `rotateVector()`, 223
 - `round()`, 16–18
 - `rrVertexCollisionAngled()`, 219
 - `rrVertexCollisionStraight()`, 216–217
 - `scaleVector()`, 127
 - `screenPosTo3DPost()`, 411
 - `sillyAdd()`, 498
 - `sin()`, 103
 - `smoothNormals(collisionMap)`, 274
 - `solveSimultaneous()`, 75
 - `solveTriangle()`, 118
 - `splitPolygon(poly)`, 274
 - `sqrt()`, 57
 - `StrictModulo()`, 32
 - `surfaceColor()`, 458–459
 - `switchBasis()`, 140
 - `tan()`, 104
 - `unitVector()`, 327
 - using, 56
 - `variance()`, 272
 - `visibleSquares()`, 551–552
 - `writeColor()`, 266
 - writing, 20
- See also* trigonometric functions
- Fundamental Theorem of Arithmetic, 28–29
- ## G
- Galileo, 186
- game algorithms
- `A* Algorithm`, 554–556, 592
 - Bresenham’s Algorithm, 501–504, 510
 - computational complexity, 498–499
 - constant-time calculations, 499
 - Eller’s Algorithm, 540–541
 - exponential time calculations, 498
 - greedy, 540
 - Kruskal’s Algorithm, 540–541
 - logarithmic time calculations, 499
 - polynomial time calculations, 498
 - Prim’s Algorithm, 539
- See also* genetic algorithm; TBG (tile-based games)
- game space, defined, 516
- game theory
- limitations, 571–572
 - Tic-Tac-Toe, 566–571
 - zero-sum games, 560–562
- games
- optimal strategy, 565
 - parallax scrolling, 518
 - Rock-Paper-Scissors, 564
 - Rube, 562–563
 - scrolling directions, 517
 - simultaneous, 566
 - solving, 562–566
 - zero-sum, 560–562
- GCD (greatest common divisor), 27, 30
- `gcd(n, m)`, 27, 30
- genetic algorithm
- DNA strands, 603
 - fitness landscape, 601–602
 - genome factor, 601
 - `mateOrganisms()` function, 603

natural selection, 599–602
 tweaking, 603–605
 using, 601–603
 utility function, 601
See also algorithms
 geometry, defined, 89
`getOscillatorSpeed()` function, 381–382
`getRocketSpeed()` function, 358–359
 gloss maps, 460
 GOFAI (good old-fashioned AI), 576
 Golden Ratio, 37. *See also* ratios
 Gouraud shading, 468–469
 gradient methods, 176–179
 gradient of functions, 158–160
 graphics functions, using, 80
 graphs
 asymptotic functions, 85
 Cartesian, 77
 of $\cos(x)$, 102
 diagonal functions, 84
 drawing on computers, 80
 features of, 77
 horizontal functions, 84
 parabolic functions, 84–85
 plotting functions on, 79–83
 quadrants, 91
 reading plots of, 83
 of $\sin(x)$, 102
 of $\tan(x)$, 102
 gravitation, law of, 302
 gravitational constant, 302
 gravitational potential energy, 192
 gravity
 discovery of, 302
 inverse-square relationship, 302
 motion of planets under, 303–304
 greatest common divisor (GCD), 27, 30
 Greek alphabet, 633–634
 grip, explained, 343

H

hair, modeling for animation, 484
 halfminor axis, explained, 447
 heat energy, 192
 height maps, using, 263
 hexadecimal system, 9
 Hofstadter, Douglas, 133, 561, 586
 homogeneous coordinates, 399–402
 Hooke's law, applying to springs, 362, 364
 horizontal functions, 84
 human eye
 cones and rods, 452
 light detection in, 452

I

IEEE (Institute of Electrical and Electronics Engineers), 13
 IK (inverse kinematics), problem of, 490–493
`IKapproach()` function, 494
`illumination()` function, 459
 image maps
 bespoke, 462
 bump, 460–461
 cubical, 462
 cylindrical, 462
 emission, 460
 explained, 460
 fitting to shapes, 462–466
 gloss, 460
 light, 460–461, 465
 mapping texels, 462
 matching points and texels, 462
 normal, 460–462
 planar, 462–463
 reflection, 460, 465
 spherical, 462
 texture, 460
 images, displaying, 406
 impulse, explained, 301
 inequalities, representing, 59
 inertia
 defined, 300
 and mass, 190–191
 moment of, 313–314
 inspection, using with quadratics, 66–67
 Institute of Electrical and Electronics Engineers (IEEE), 13
 integer calculations
 benefit of, 500
 Bresenham's Algorithm, 268, 501–504, 510
 interactive method of solution, 503
 problem with, 501
 rounding errors, 501
 using, 19
 integers
 `BigInteger` classes, 19–20
 divisors of, 27
 factors of, 27
 floats to, 16
 representing, 4, 9–11
 integration
 definite integral, 168
 in differential equations, 171
 vs. differentiation, 168
 indefinite, 168
 process of, 168–169
 interest
 calculating for debts, 42–43
 compound, 41

interpolation, explained, 425
 intersection, determining with vertex, 259
`intersectionPoint()` function, 143–144
`intersectionTime()` function, 144–145, 204
 inverse functions, 57
 inverse kinematics (IK), problem of, 490–493
 irrational numbers
 vs. real numbers, 5
 representing, 12–13
 isometric view, explained, 523–524, 409

J

`javelin()` function, 196
 just mean, explained, 114

K

Kepler, Johannes, 302–304
 kinematics, ragdoll problem, 489–490
 kinetic energy, 192, 318
 kite, defined, 211
 Kruskal’s Algorithm, 540–541

L

labyrinths vs. mazes, 534
 Lambertian reflection, 453, 457
 laminar objects, 315–316
 lathe technique, using with surfaces, 474–475
 LCM (lowest common multiple), 30–31
 leading point, calculating, 260. *See also* points
`leadingEdgeOfPolygon()` function, 261–262
`leadingPointOfPolygon()` function, 260–261
 left-hand axis, explained, 394
 letter As, drawing, 132
 levers
 inertia, 313–315
 inertial and laminar objects, 315–316
 light rod, 310
 physics of, 309–310
 rod and fulcrum, 310
 torque, 310–313
 See also angular motion
 light maps, 460–461, 465
 lighting, baking, 460
 lights
 ambient, 454–455
 attenuated, 454–455
 directional, 454–455
 electromagnetism, 452
 emergence of colors, 452
 fake, 454–455
 point, 454–455
 real, 452–453
 specular reflection, 453
 spot, 454–455
 visible, 452

“limit of the series,” explained, 12
 limits, representing, 158
 line segment, defined, 130
 linear inequalities, solving, 438
 linear particle, angular momentum of, 317
`lineOfBestFit()` function, 272
`linePlaneIntersection()` function, 399
 lines
 calculating collision of, 441
 defining in 3-D space, 394
 detecting collisions between, 325–329
 finding intersection of, 142
 finding intersections with planes, 398–399
 intersecting, 145–146
 parallel, 130
 representing slopes of, 100
 rotating, 330–331
 Lingo scripting language
 Boolean data type, 629
 data types, 629
 float data type, 629
 global variables, 631
 integer data type, 629
 list data type, 630
 local variables, 630
 object data type, 630
 operators, 631
 point/vector data type, 630
 property variables, 630
 string data type, 629
 symbol data type, 629
 variables, 630–631
`log()` function, 164
 logarithmic time calculations, 499
 logarithms
 calculating with, 47–48
 differentiating, 164–165
 vs. exponentials, 47
 for large numbers, 50–51
 simplifying calculations, 48–49
 using with powers, 50
 look-up tables
 bespoke, 500
 using, 497
 lowest common multiple (LCM), 30–31

M

m vs. m , 196
`madPath()` function, 138
`magnitude()` function, 127
`makeMinimaxStrategy()` function, 570
`makeTTTList()` function, 568–569
`makeTTTtree()` function, 568–569
 mantissa, defined, 13
 markers, using in counting, 12

- mass
 - and inertia, 190–191
 - representing, 196
- `mateOrganisms()` function, 603
- materials
 - color elements of surfaces, 456–459
 - defined, 455
 - fitting maps to shapes, 462–465
 - image maps, 460–462
 - mip-maps, 465–468
 - textures, 460–462
- matrices
 - arithmetic, 149–152
 - defined, 147
 - `determinant()` function, 149
 - as instructions, 152
 - leading diagonal, 148
 - multiplying, 152–153
 - representing values of, 147
 - rows and columns in, 147–148
 - as tensors, 149
 - as transformations, 152–154

See also vectors
- `matrixMultiply()` function, 150–152
- maze data, handling, 538
- maze navigation
 - camera control, 547–549
 - collision detection, 547–549
 - line of sight, 549–552
 - pathfinding and A* Algorithm, 554–556
- mazes
 - basing on concentric circles, 545
 - circular, 544–546
 - classifying, 531–532
 - converting, 546
 - convolution, 536
 - degrees of river, 536–537
 - degrees of run, 537
 - dimensionality, 535
 - distance from mean, 536
 - edges and nodes, 532
 - generating automatically, 538–543
 - geometry, 535
 - graph classifications, 533
 - graphs and connectivity, 532–534
 - labeling edges in graphs, 533
 - vs. labyrinths, 534
 - multiply-connected, 533, 543–544
 - navigating in, 547
 - Prim's Algorithm, 539
 - pseudo-graphs, 533
 - psychological factors, 535
 - recursive backtracker, 538–539
 - simply connected, 533
 - standard deviation, 536–537
- texture, 535
- topology, 532
- true graphs, 533
- twists and turns, 534–537
- underlying grid, 535
- wall-sliding, 547
- maze-threading
 - marking system, 553–554
 - recursive search, 553
 - turning in directions, 552–554
- `mean()` function, 272
- measurement, isolation of, 200
- meshes
 - collisions with, 448–449
 - defined, 415
 - LOD (level of detail), 482
- meters, units of, 196
- `minimaxIteration()` function, 571
- minutes, evaluating, 34
- mip-maps
 - aliasing, 465
 - anisotropic filtering, 468
 - bilinear filtering, 465–466, 468
 - oversampling, 466
 - terminology, 467
 - trilinear filtering, 467
- `mod()` function, 32
- modeling techniques
 - animated surfaces, 483–487
 - bone animations, 487–493
 - NURBS (non-uniform rational B-spline), 476–480
 - sine and cosine surfaces, 480–481
 - splines in 3-D, 475–476
 - surfaces of rotation, 474–475
 - tessellation, 481–483
- models, defined, 415. *See also* objects
- `modulate()` function, 459
- modulo arithmetic, 31–33
- `moment()` function, 317
- moment of inertia, calculating (figure), 314
- momentum
 - calculating, 191
 - conserving, 191
 - continuous, 355–358
- monitor screens
 - high-resolution display, 453
 - RGB, 453
- Morgenstern, Oskar, 560
- motion
 - in 3-D space (figure), 428
 - of cannonball, 188–190
 - creating with transforms, 421–424
 - interpolating, 425
 - programming, 134–135
 - simplifying, 506

`moveBalls()` function, 287–290, 293

`moveCannonBall()` function, 307

multiplication

 in binary system, 11

 concept of, 4

 pairwise, 124

 scalar, 124–126

`multiplyConnected()` function, 544

multivalued functions, 57

N

(n, m) , explained, 27

Nash Equilibrium strategy, applying to Rube, 563

natural numbers, 4–5

natural selection, 599–601

negative numbers

 acceptance of, 5

 explained, 4

neural networks, 581–584

 back-propagation, 584

 limiter function, 582

 multi-layer perception (figure), 583

 training, 584–585

 training MLP, 584

`neuralNetStep()` function, 582–583

Newton, Isaac, 302

 inverse-square relationship, 302–303

 lifetime of, 297

`newtonRaphson()` function, 178

Newton–Raphson method, 176–179

Newton’s Cradle, 242, 345

Newton’s Laws

 first: velocity of objects, 298–299, 355

 second: force on body, 299, 316, 341, 351, 364

 third: exertion of force, 300, 339–340, 343, 350

n-gon, describing, 248

nodes

 defined, 415

 describing, 416–418

 parents and children, 427–429

non-uniform rational B-spline (NURBS). *See* NURBS

 (non-uniform rational B-spline)

`norm()` function, 128

normal maps, 460–462

NOT operator, 11

NP-hard problems, solving, 591

numbers

 adding as strings in binary, 10–11

 complex, 5

 composite, 28

 computed, 13–15

 counting, 4

 double precision, 15

 integers, 4, 9–11

 irrational, 5, 12–13

natural, 4

negative, 4

perfect, 27

prime, 27–28

rational, 5, 12–13

real, 5

size vs. precision, 14

square roots of, 6

as strings of digits, 6–8

as switches, 9

 unique symbols for, 6–7

`numberToString()` function, 7

numerator, defined, 5

NURBS (non-uniform rational B-spline), 475–480

 advantages of, 480

 arrays, 479

 basis functions, 477

 behaviors, 478

 blending functions, 477

 knots, 476

 NU part of, 476

 RBS part of, 479

See also surfaces

`NURBSbasisFunction()`, 478

O

OABB (object-aligned bounding box), 270–271, 273, 512

objects

 angular momentum of, 316

 displacement of, 200

 inertial and laminar, 315–316

 positioning, 416

 properties of, 201

 rotating, 417

 rotating around center, 114

 rotating by angles, 112–113

 scaling, 35–36, 416–417, 423

See also models

octrees, 508–510

ODE (ordinary differential equation)

 defined, 170

 solving, 172–173

one-to-one functions, 57

operators, Lingo, 631

optimal strategy, 565

optimizing code

 binary-space partitioning, 511

 BVH (bounding volume hierarchy), 512–513

 cheap calculations, 497

 computational complexity, 498–499

 culling, 507–513

 expensive calculations, 497

 integer calculations, 500–504

 look-up tables, 499–500

 octrees, 508–510

- quadtrees, 508–510
 - segregating space, 507–508
 - simplifying collisions, 504–506
 - simplifying motion, 506
 - OR operator, 11
 - orbits
 - angular frequency, 304
 - geo-stationary, 306
 - mean motion n , 304
 - period, 304
 - semi-major axis, 304
 - stable, 304–305
 - ordinary differential equation (ODE)
 - defined, 170
 - solving, 172–173
 - orthogonal matrices, using, 420
 - orthographic projections, 408–409
 - oscillations
 - coupling, 375–376
 - damping, 371–373
 - defined, 361
 - DHM (damped harmonic motion), 369–371
 - linked motion, 375–376
 - pushing swings, 374–375
 - resonance, 374–375
 - SHM (simple harmonic motion), 364–369
 - sinusoidal, 375
 - spring motion, 376–382
 - springs, 362–364, 374–376
 - using to animate water, 486
 - waves, 382–389
 - overflow, representing, 15
- P**
- p (pi), 93–94
 - paper sizes (figure), 36
 - parabolic functions, 84–85
 - parallax scrolling, explained, 518
 - parallel lines, representing, 130
 - parallelogram, defined, 211
 - parameterization, using, 86–87
 - parametric equations, 167
 - parametric functions, 249–250
 - partial derivatives, 167
 - particleEllipseCollision() function, 225
 - particleHmapCollision() function, 267
 - particleOnSpring() function, 378–380
 - particles
 - defined, 135
 - mass of, 190
 - partitioning trees
 - leaves, 507
 - nodes, 507
 - parents and children, 507
 - quadtrees, 508–510
 - root node, 507–508
 - pathfinding problem
 - A* method, 554–556
 - breadth-first search, 554
 - depth-first search, 554
 - vs. maze threading, 554
 - pendulum, motion of, 368
 - percentages
 - calculating, 40
 - compound interest, 41
 - debts and interest, 42–43
 - See also* fractions
 - perfect number, explained, 27
 - perpendicular lines, explained, 91
 - perspective
 - creating horizon, 407
 - developing, 407
 - drawing scenes in, 407
 - explained, 407
 - Phong shading, 468–470
 - physics
 - ballistics, 183–190, 195–196
 - energy, 192–196
 - mass, 190–191
 - momentum, 191
 - See also* pseudo-physics
 - pi() function, 93
 - picking, using raycasting in, 411–414
 - planar mapping (figure), 463
 - planePlaneIntersection() function, 399
 - planes
 - defining in 3-D space, 394
 - finding intersections with lines, 398–399
 - in frustum, 404
 - planet, speed of (figure), 303
 - platform games
 - in 3-D, 524–528
 - genre of, 515
 - pointCircleCollision() function, 206, 227
 - pointCmapIntersection() function, 267–268
 - pointInsidePolygon() function, 258–259
 - pointInsidePolygonIncomplete() function, 257–258
 - pointInsideRectangle() function, 213
 - pointOnRectangle() function, 213
 - pointParallelogramCollision() function, 227–228
 - pointPolygonCollision() figure, 255–256
 - pointRectangleCollision() function, 227
 - pointRectangleIntersection() function, 214–215
 - points
 - determining inside shapes, 256–259
 - projecting to projection plane, 404
 - testing inside polygons, 439
 - translating from space to screen, 405–406
 - See also* leading point

pointsToCheck() function, 217
 polygons
 defined, 211
 describing using coordinates, 248
 polynomial time calculations, 498
 polynomials
 defined, 58
 factoring, 72
 polyominoes problem (figure), 593
 pool game
 applying spin to, 334–335
 balls for, 281–283
 collisions, 287
 cue, 285–286
 culling, 291–292
 friction, 287
 logic, 293
 main loop, 287–290
 physical parameters, 283–285
 pocket mouth (figure), 279
 pocketed balls, 293
 repeating, 287
 table for, 278–281
 table partitions, 291–292
 taking shots, 285
 time elapsed, 287
 pos3DToScreenPos() function, 405–406
 powers
 calculating with, 44–45
 decay, 46–47
 defined, 6
 differentiating, 164–165
 exp() function, 45–46
 vs. logarithms, 47
 using, 46–47
 using logarithms with, 50
 prime factors, finding, 29
 prime numbers
 examples of, 28
 explained, 27
 generating, 28–29
 Prim's Algorithm, 539
 principle of relativity, 200–201
 Prisoner's Dilemma zero-sum game, 561–562
 probability theory, 564
 problem solving
 depth-first searches, 592
 interaction, 594
 looking for bottlenecks, 592
 polyominoes, 593
 searching for answers, 591–593
 problems
 auditing progress with, 594
 breadth-first searches, 593

NP-hard, 591
 representing, 590–591
 projectile, firing, 189
 projection plane
 defined, 403
 projecting point to, 404
 projections
 central, 408
 orthographic, 408–409
 proportion, defined, 35
 pseudo-physics
 simplifying collisions, 504–506
 simplifying motion, 506
 See also physics
 Pullen, Walter, 535–536
 pulleys, 353–355
 Pythagorean Theorem, 97–100, 445
 extending, 100
 right-angled triangles, 100
 triples, 99

Q

Q, representing rational numbers with, 5
 quad word, defined, 10
 quadratic equations, 64–65
 cubic equations, 68–70
 factoring, 65
 factors, 65–67
 simplifying, 66–67
 using inspection, 66–67
 See also equations
 quadratic expression, defined, 65
 quadratic formula, using, 68
 quadrilaterals, examples of, 211
 quadtrees, 508–510
 quaternions, applying to rotations, 425–427
 quotient, defined, 5

R

R, representing real numbers with, 5
 radians, dividing circles into, 94
 radius, defined, 90, 93
 radix point, defined, 12
 ragdoll problem (figure), 489
 rational B-splines, 479. *See also* B-splines
 rational numbers, 5, 12–13
 ratios
 defined, 35
 using, 35–36
 See also fractions; Golden Ratio
 raycasting
 defined, 404
 dragging, 411–414

- dropping, 411–414
- in interaction, 411–414
- picking, 411–414
- terrain following, 410
- using, 256, 409–414
- `readColor()` function, 266
- real vs. irrational numbers, 5
- reciprocals, fractions as, 5
- rectangles, calculating area of, 92. *See also* squares
- `rectangleCircleInnerCollision()` function, 228
- `rectangleRectangleAngledCollision()` function, 218–219
- `rectangleRectangleCollisionStraight()` function, 216
- `rectangleRectangleInnerCollision()` function, 228
- rectangles
 - collisions with circles, 226–227
 - described for collision detection, 211–212
 - at different angles, 218–219
 - point of contact, 219–220
 - quadrilateral, 211
 - at same angle, 215–218
 - stationary and moving point, 213–215
- See also* squares
- recursive algorithm, example of, 6–7
- `recursiveBacktrack()` function, 539
- red, green, blue (RGB), 453
- reductio ad absurdum* argument, 186
- redux list, using with equations, 75
- reflection maps, 460, 465
- reflections, 115–116
- refraction, occurrence of, 386
- Regula Falsa* bracketing method, 175
- relativity, principle of, 200–201
- remainder, defined, 7
- rendering
 - frustum, 403
 - process of, 402
 - projection plane, 403–406
- `resolveAngularCollision()` function, 333–334
- `resolveCollisionEqualMass()` function, 235, 283
- `resolveCollisionFree()` function, 234–235
- `resolveCushionCollision()` function, 335
- `resolveFixedCollision()` function, 231
- `resolveInelasticCollisionFixed()` function, 237–238
- `resolveInelasticCollisionFree()` function, 237
- resonance
 - defined, 374–375
 - driving frequency, 374
 - pushing swing, 374–375
- `resultantForceOnObject()` function, 340
- RGB (red, green, blue), 453
- rhombus, defined, 211
- rocket fuel, 357–358
- rockets and satellites, 305–307
- Rock-Paper-Scissors, 560, 564
- rolling, defined, 337
- root
 - bracketing, 174
 - moving closer to, 176
- `rotateToFollow()` function, 118
- `rotateVector()` function, 223
- rotating line and circle collision, 319–323
- rotating lines, 330–331
- rotation
 - determining angle of, 424
 - physics of, 313
- rotational kinetic energy, 318
- rotations by angles, 115
- `round()` function, 16–18
- rounding errors, 18–19
- in integer calculations, 501
- `rrVertexCollisionAngled()` function, 219
- `rrVertexCollisionStraight()` function, 216–217
- Rube zero-sum game, 562–563
 - Nash Equilibrium strategy, 563
 - saddle point of matrix, 563
- Russian Vine example, 46–47, 50

S

- satellites and rockets, 305–307
- scalars
 - as tensors, 149
 - and vectors, 129–130
- scales of objects, using ratios with, 35–36
- `scaleVector()` function, 127
- scientific representation, 13
- `screenPosTo3DPos()` function, 411
- scrollbars, considering as sliders, 40
- search function, creating for chess case study, 597–598
- search space, classifying, 590, 595
- search techniques, 590–591
- searches
 - breadth-first, 593
 - depth-first, 592
- secant method (figure), 178–179
- set of equations, solving, 70–71
- shading
 - Gouraud, 468–470
 - normal at vertex, 470
 - Phong, 468–470
- shapes
 - area of, 92
 - convex and concave, 255–256
 - defining complex, 247–250
 - describing with vectors, 130–132
 - determining points in, 256–259
 - finding leading edge for complex, 259–263
 - fitting image maps to, 462–466
- See also* bounding shapes

- SHM (simple harmonic motion)
 amplitude, 366
 calculating parameters, 368–369
 and circular motion, 367
 vs. damped harmonic motion (DHM), 370–371
 vs. DHM (damped harmonic motion), 370–371
 equation of, 364–367
 examples of, 367–368
 explained, 364
 initial condition, 366
 maximum displacement from equilibrium, 366–367
 period of motion, 366
 phase of motion, 366
 using to animate water, 486
 sigma notation, using with variance, 271
 significand, defined, 13
`sillyAdd()` function, 498
 simple harmonic motion (SHM). *See* SHM (simple harmonic motion)
 simplex algorithm, using, 438
 simulation, realistic, 278
`sin()` function
 on circle, 117–118
 inverse of, 103
 using, 100–102
 sinusoidal oscillation, explained, 375
 slide rule (figure)s, 48–49
 sliders
 initializing, 38
 intrinsic proportion, 38
 scrollbars, 40
 values, 38–39
 slippnet, defined, 586
`smoothNormals(collisionMap)` function, 274
`solveSimultaneous()` function, 75
`solveTriangle()` function, 118
 space, segregating, 507–508
 specular highlight, explained, 457
 specular reflection
 calculating, 458
 diffuse reflection, 453
 explained, 453
 Lambertian reflection, 453
 lights, 453
 spheres
 and boxes, 441–442
 collision normal, 434
 collisions, 448
 defined, 432
 designating, 432
 hyperspheres, 432
 and moving points, 434
 multiple, 434
 points on, 432
 vectors, 432
 and walls, 433–434
 spherical mapping (figure), 464
 spheroids, defined, 435
 spin
 applying to pool game, 334–335
 flywheel, 316
 rotational kinetic energy, 318
 of three-dimensional objects, 318
 velocity of, 317
 spinning collisions, 319–323. *See also* collision detection
 spline-based tiles, 528–529
 splines
 in 3-D, 475–476
 and Bezier curves, 251–255
 Catmull-Rom, 252–253
 movable, 254–255
See also B-splines
`splitPolygon(poly)` function, 274
 springs
 calculating motion of, 376
 coefficient of elasticity, 362
 compressive, 362
 dashpot, 486
 described, 362
 DHM oscillation, 380–382
 extension of, 362
 force due to, 376–378
 force in, 362–363
 Hooke’s law, 362
 undamped and uncoupled, 378–380
 using to measure weight, 363–364
 sprite, defined, 135
`sqrt()` function, representing, 57
 square root, explained, 6. *See also* difference of two squares
 squares
 constructing with vectors, 130–131
 described for collision detection, 211–212
See also rectangles
 starfish shape (figure), 250
`StrictModulo()` function, 32
 strings
 and circular motion, 351–353
 inextensible, 350
 passing through tables, 352
 on tables, 350–351
 substitution, solving equations by, 70–72
 surface of rotation
 creating, 443
 lathe tool, 443
`surfaceColor()` function, 458–459

surfaces
 color element of, 456–459
 creating, 474–475
 finding normal to (figure), 483
 of rotation, 474–475
 sine and cosine functions, 480–481
 tessellating, 481–483
See also NURBS (non-uniform rational B-spline)
 swing, pushing, 374–375
`switchBasis()` function, 140
 symbols, Greek alphabet, 633–634

T

tables, strings on, 350–351
`tan()` function
 inverse of, 104
 using, 100–102
 TBG (tile-based games)
 camera control, 516–518
 collisions, 518–521
 complex tiles, 521–522
 creating, 516
 isometric view, 523–524
 level editor, 516
 movement, 516–518
 spline-based tiles, 528–529
See also game algorithms
 tension
 calculating, 351
 force of, 349–350
 tensors
 defined, 149
 using, 316
 terms, 55–56
 terrain following, 410
 tessellation
 process of, 481
 scaling, 482
 storing surfaces as meshes, 482
 texels
 explained, 460
 mapping to surfaces, 462
 matching to points, 462
 using transforms with, 462–463
 texture maps, 460
 textures, using mip-maps with, 465–466
 third dimension, adding, 394–395
 Tic-Tac-Toe
 alpha-beta search, 572
 applying search to, 568–571
 bottom-up approach, 586
 calculating winning strategy, 570–571
 fork subgoal, 578–579
 game theory approach, 566–568
 limitations, 571–572
 optimizing search tree, 572
 parameters, 575
 strategic game of, 579
 subgoal of, 578–579
 tactical AI approach, 575
 training process, 575
 tile-based games (TBG)
 camera control, 516–518
 collisions, 518–521
 complex tiles, 521–522
 creating, 516
 isometric view, 523–524
 level editor, 516
 movement, 516–518
 spline-based tiles, 528–529
See also algorithms
 time calculations
 constant, 498
 exponential, 498
 logarithmic, 498
 polynomial, 498
 topspin, explained, 344
 torque
 bending moments, 313
 calculating, 311, 316–317
 defined, 310
 exertion on lever (figure), 310
 forces applied to, 312
 moments, 311
 shearing force, 313
 stress, 313
 torus, defined, 476
 traction, explained, 343
 trajectory, calculating, 134–135
 transformation matrix, 418–421
 transformations
 affine, 416
 applying scale to objects, 423
 arrow model (figure), 422
 creating motion with, 421–424
 decomposing, 420
 (figure), 416
 interpolation, 425
 orthogonal matrices, 420
 parents and children, 427–429
 quaternions, 425–427
 rotations, 416, 420, 423–424
 scale, 416
 translation, 416
 transforms
 position, 416–418
 rotation, 416–418
 scale, 416–418
 using with texels, 462–463
 trapezium, defined, 211

triangles
 acute, 96
 angles, 95–96
 areas of, 109–110
 congruent, 109
 defined, 94
 equilateral, 96, 131
 finding centrum of, 114
 hypotenuse, 97
 isosceles, 96
 non-collinear points, 95
 obtuse, 96
 reflection, 111, 116
 right, 97, 100
 rotation, 111, 116
 scalene, 96
 scaling, 111
 shearing, 111
 similar, 108–109
 sine and cosine rules, 105–108
 solving for, 109–110
 transformations, 110–111
 translation, 111, 116
 vertices, 95
 trigonometric functions, 100–102
`acos()`, 103
`arcsin()`, 103
`arctan()`, 104–105
`asin()`, 103
`atan()`, 103
 circular motion, 117–118
`cos()`, 100–102, 117
 differentiating, 164–166
 domains and ranges, 104
 inverse, 103–105
 rotating objects by angles, 112–113
`sin()`, 100–102, 117
`tan()`, 100–102
 transformations, 110–111
 use of brackets, 104
See also functions
 trigonometric identities, 103
 trigonometry, defined, 89, 94
 trilinear filtering (figure), 467
 troubleshooting. *See* problem solving
 Turing, Alan, 559–560

U

Undercut zero-sum game, 561, 565
 underflow, representing, 15
`unitVector()` function, 327

V

variables
 described, 54
 using, 55
 variance, formula for, 271
`variance()` function, 272
 vector paths, 135–136
`curvedPath()` function, 136–137
`madPath()` function, 138
 varying velocity, 137
 vectors
 adding together, 125
`addVectors()` function, 127
 angle between, 128
 arithmetic, 124–128
`component()` function, 140
`componentVector()` function, 140
 converting to new basis, 139
 describing shapes with, 130–132
 designating negative changes, 125
 differences of, 126–127
 vs. differential equations, 171
 direction of, 123
 direction vs. position, 397
 displacement, 129, 133
 distance, 129
 dot product, 141
 drawing, 123
 equations, 142–146
 equilateral triangles, 131
 as instructions, 122
 intersecting lines, 145–146
`intersectionPoint()` function, 143–144
`intersectionTime()` function, 144–145
`magnitude()` function, 127
 magnitude of, 123
 mass, 129
 mean of, 127
 motion, 130–138
 moving from P to Q, 133–135
 non-zero, 125
`norm()` function, 128
 normal, 128–129
 normalized, 138
 orthogonal, 138
 orthonormal, 138
 pairwise multiplication, 124
 parallel lines, 130
 perpendicular, 129
 position, 123–124
 programs, 127–128

properties of, 123
 reversing arrowhead, 125
 scalar division, 124–126
 scalar multiplication, 124–126
 scalar product, 141
 and scalars, 129–130
`scaleVector()` function, 127
 separating into components, 138–140
 speed, 129, 133–134
 squares, 130–131
 starting point of, 123
 subscript, 123
 sum of, 125–126
`switchBasis()` function, 140
 as tensors, 149
 unit, 123
 value det as zero, 146
 velocity, 129–130, 133–134
 weight, 129
See also 3-D vectors; matrices
 velocity
 vs. acceleration, 185
 calculating, 317
 vertices, using to determine
 intersections, 259
 viewport, specifying, 405
`visibleSquares()` function, 551–552

von Neumann, John, 560
 vulgar fractions, 5

W

wall, moving with circle, 202–204
 wall-sliding, using with mazes, 547
 water, modeling for animation, 486–487
 waveform, defined, 383
 wavefront
 defined, 383
 (figure), 384
 waves
 addition and subtraction, 385–386
 angle of incidence, 386
 angle of reflection, 386
 behavior of, 386–389
 compression, 383
 coupled oscillations (figure), 382
 diffraction, 389
 Doppler effect, 388
 Doppler shift, 388
 Fourier analysis, 386
 index of refraction, 388
 interface, 387
 longitudinal, 383–384
 modeling for animation, 487
 motion, 382–383
 rarefaction, 383

reflection, 386–387
 refraction, 386
 spectra of, 386
 transverse, 383–384
 weight
 describing, 190
 feeling, 299, 301
 measuring with springs, 363–364
 wheels and grip, 343–345
`writeColor()` function, 266

Z

Z, representing integers with, 4
 zero (0), representing, 15
 zero-sum games, 560–562
 Prisoner’s Dilemma, 561–562
 probability theory, 564
 Rock-Paper-Scissors, 560
 Undercut, 561, 565