GAME PHYSICS
ENGINE DEVELOPMENT

Ian Millington

MK

Praise for *Game Physics Engine Development*

"*Game Physics Engine Development* is the first game physics book to emphasize building an actual engine. It focuses on the practical implementation details and addresses trade-offs that must be made. Ian's experience with commercial physics engines definitely shows. His book fills a gap by demonstrating how you actually build a physics engine."

— Dave Eberly, President, Geometric Tools

"Ian Millington has achieved the remarkable task of creating a self-contained text on game physics programming. If you are charged with putting together a game physics engine, this book will carry you through from beginning (with a math and physics primer) through the process (with detailed attention to extendable software design) to the end (collision handling and constrained dynamics). If you are asked to use a game physics engine, this text will help you understand what is going on under the hood, and therefore make you a better user.

The text is practical enough to serve the industry practitioner (read this before starting your project!), and the writing is solid enough to be used in an undergraduate course on video game physics. *Game Physics Engine Development* comes bundled with working source code closely matched to the text—a valuable service to the reader: the bundled implementation serves as a foundation for experimentation, a reference for comparison, and a guide to modular software design. Millington's writing maintains a steady pace and a friendly tone; the narrative level will appeal to a broad audience of students and practitioners."

*— Professor Eitan Grinspun, Department of Computer
Science, Columbia University, New York*

"A competent programmer with sufficient mathematical sophistication could build a physics engine just from the text and equations—even without the accompanying source code. You can't say this about a lot of books!"

— Philip J. Schneider, Industrial Light + Magic

"Thorough content with an engaging style; the essential mathematics is presented clearly with genuine and effective explanation and appropriate scope. The C++ code samples are a major strength."

*— Dr. John Purdy, Department of Computer
Science, University of Hull, UK*

The Morgan Kaufmann Series in Interactive 3D Technology

The game industry is a powerful and driving force in the evolution of computer technology. As the capabilities of personal computers, peripheral hardware, and game consoles have grown, so has the demand for quality information about the algorithms, tools, and descriptions needed to take advantage of this new technology. To satisfy this demand and establish a new level of professional reference for the game developer, we created the *Morgan Kaufmann Series in Interactive 3D Technology*. Books in the series are written for developers by leading industry professionals and academic researchers, and cover the state of the art in real-time 3D. The series emphasizes practical, working solutions, and solid software-engineering principles. The goal is for the developer to be able to implement real systems from the fundamental ideas, whether it be for games or for other applications.

Game Physics Engine Development
Ian Millington

Artificial Intelligence for Games
Ian Millington

X3D: Extensible 3D Graphics for Web Authors
Don Brutzman and Leonard Daly

3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics, Second Edition
David H. Eberly

3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic
David H. Eberly

Game Physics
David H. Eberly

Better Game Characters by Design: A Psychological Approach
Katherine Isbister

Real-Time Collision Detection
Christer Ericson

Visualizing Quaternions
Andrew J. Hanson

Physically Based Rendering: From Theory to Implementation
Matt Pharr and Greg Humphreys

Collision Detection in Interactive 3D Environments
Gino van den Bergen

Essential Mathematics for Games and Interactive Applications: A Programmer's Guide
James M. Van Verth and Lars M. Bishop

Forthcoming
Real-Time Cameras
Mark Haigh-Hutchinson

GAME PHYSICS
ENGINE
DEVELOPMENT

IAN MILLINGTON

This book is printed on acid-free paper.

To Melanie

About the Author

Ian Millington is a partner at IPR Ventures, a consulting company involved in the development of next-generation technologies for entertainment, modeling, and simulation. Previously, he founded Mindlathe Ltd, the largest specialist AI middleware company in computer games, which worked on a huge range of game genres and technologies. He has an extensive background in artificial intelligence (AI), including PhD research in complexity theory and natural computing. Ian has published academic and professional papers and articles on topics ranging from paleontology to hypertext and is the author of *Artificial Intelligence for Games* (Morgan Kaufmann, 2006).

Contents

CHAPTER

3   THE LAWS OF MOTION

CHAPTER

4   THE PARTICLE PHYSICS ENGINE

LIST OF FIGURES

Preface

When I started writing games, in the 8-bit bedroom coding boom of the eighties, the low budgets and short turnaround times for writing games encouraged innovation and experimentation. This in turn led to some great games (and, it has to be said, a whole heap of unplayable rubbish). Let no one tell you games were better back then! I remember being particularly inspired by two games, both of which used realistic physics as a core of their gameplay.

The first, written by Jeremy Smith and originally published for the United Kingdom's BBC Micro range of home computers, was *Thrust*. Based on the arcade game *Gravitar*, an ivy-leaf shaped ship navigates through underground caverns under the influence of a two-dimensional (2D) physical simulation. The aim is to steal a heavy fuel pod, which is then connected to the ship via a cable. The relatively simple inertial model of the spaceship then becomes a wonderfully complex interaction of two heavy objects. The gameplay was certainly challenging, but had that one-more-time feel that denotes a classic game.

The second game, written by Peter Irvin and Jeremy Smith (again), was *Exile*. This is perhaps the most innovative and impressive game I have ever seen; it features techniques beyond physics, such as procedural content creation that is only now being adopted on a large scale.

Exile's physics extends to every object in the game. Ammunition follows ballistic trajectories—you can throw grenades, which explode sending nearby objects flying; you can carry a heavy object to weigh you down in a strong up-draft; and you can float pleasantly in water. *Exile*'s content qualifies it as the first complete physics engine in a game.

Although *Exile* was released in 1988, I feel like a relative newcomer to the physics coding party. I started writing game physics in 1999 by creating an engine for modeling cars in a driving game. What I thought was going to be a month-long project turned into something of an albatross.

I ran headlong into every physics problem imaginable, from stiff-suspension springs, which sent my car spiraling off to infinity; to wheels that wobbled at high speed because friction moved objects around of their own accord; to hard surfaces that looked like they were made of soft rubber. I tried a whole gamut of approaches, from impulses to Jacobians, from reduced coordinates to faked physics. It was a learning curve unlike anything before or since in my game coding career.

While I was merrily missing my deadlines (driving physics gave way to third-person shooters) and my company examined every middleware physics system to be found, I learned a lot about the pitfalls and benefits of different approaches. The code I wrote, and often abandoned, proved to be useful over the intervening years as it got dusted off and repurposed. I have built several physics engines based on that experience, and have customized them for many applications, so now I think that I have a good sense of how to get the best effects from the simplest approach.

Game development has entered a phase in which physics simulation is a commodity. Almost every game needs physics simulation, and every major development company has an in-house library, or licenses one of the major middleware solutions. Physics, despite being more common than ever before, is still somewhat of a black box. Physics developers do their stuff, and the rest of the team relies on the results.

Most of the information and literature on game physics assumes a level of mathematical and physical sophistication that is uncommon. Some works give you all the physical information, but no architecture for how to apply it. Others contain misinformation and advice that will sting you. Physics engines are complicated beasts, and there are a universe of optimizations and refinements out there, most still waiting to be explored. Before you can wrangle with implementing variations on the Lemke pivot algorithm, however, you need to understand the basics and to have a working body of code to experiment with.

This book was written as a culmination of the first few years of painful experimentation I went through. I wanted it to be a starting point—to be the book I needed seven years ago. The intent is to take you from zero to a working physics engine in one logical and understandable story. It is just the first step on a much longer road, but what's in this book is a sure and dependable step to take—one that's in the right direction.

About the CD-ROM

This book is accompanied by a CD-ROM that contains a library of source code to implement the techniques and demonstrations in this book. The library there is designed to be relatively easy to read, and it includes copious comments and demonstration programs. Support materials, such as updates, errata, and additional features, are available on the companion site for this book at *http://textbooks.elsevier.com/ 012369471X*.

Acknowledgements

My quest to create robust game physics, although difficult, would have been impossible without the contributions of a handful of skilled coders and mathematicians who published papers and articles, gave SIGGRAPH presentations, and released source code. Although there are many more, I am thinking particularly of Chris Hecker,

This page intentionally left blank

1

INTRODUCTION

Physics is a hot topic in computer games. No self-respecting action game can get by without a good physics engine, and the trend is rapidly spreading to other genres, including strategy games and puzzles. This growth has been largely fuelled by middleware companies offering high-powered physics simulation. Many high-profile games feature commercial physics engines.

But commercial packages come at a high price, and for a huge range of developers building a custom physics solution can be cheaper, provide more control, and be more flexible. Unfortunately physics is a topic shrouded in mystery, mathematics, and horror stories.

When I came to build a general physics engine in 2000, I found there was almost no good information available, almost no code to work from, and lots of contradictory information. I struggled through and built a commercial engine, and learned a huge amount in the process. Over the last five years I've applied my own engine and other commercial physics systems to a range of real games. More than five years of effort and experience are contained in this book.

There are other books, websites, and articles on game physics, but there is still almost no reliable information on building a physics engine—that is, a complete simulation technology that can be used in game after game. This book aims to step you through the creation of a physics engine. It goes through a sample physics engine (provided on the CD) line by line, as well as giving you insight into the design decisions that were made in its construction. You can use the engine as is, use it as a base for further experimentation, or make different design decisions and create your own system under the guidance that this book provides.

1

1.1 What Is Game Physics?

Physics is a huge discipline, and academic physics has hundreds of subfields. Each describes some aspect of the physical world, whether it is the way light works or the nuclear reactions inside a star.

Some bits of physics might be useful in games. We could use optics, for example, to simulate the way light travels and bounces and use it to make great-looking graphics. This is the way ray-tracing works, and (although it is still very slow) it had been used in several titles. This isn't what we mean when we talk about game physics. Although they are part of academic physics, they are not part of game physics, and I won't consider them in this book.

Other bits of physics have a more tenuous connection: I can't think of a use for nuclear physics simulation in a game, unless the nuclear reactions were the whole point of the gameplay.

When we talk about physics in a game, we really mean classical mechanics: the laws that govern how large objects move under the influence of gravity and other forces. In academic physics these laws have largely been superseded by new theories: relativity and quantum mechanics. In games they are used to give objects the feel of being solid things, with mass, inertia, bounce, and buoyancy.

Game physics has been around almost since the first games were written. It was first seen in the way particles move: sparks, fireworks, the ballistics of bullets, smoke, and explosions. Physics simulation has also been used to create flight simulators for nearly three decades. Next came car physics, with ever increasing sophistication of tire, suspension, and engine models.

As processing power became available, we saw crates that could be moved around or stacked, walls that could be destroyed and crumble into their constituent blocks. This is rigid body physics, which rapidly expanded to include softer objects like clothes, flags, and rope. Most recently we have seen the rise of the ragdoll: a physical simulation of the human skeleton that allows more realistic trips, falls, and death throes.

In this book we'll cover the full gamut of physics tasks. With a gradually more comprehensive technology suite our physics engine will support particle effects, flight simulation, car physics, crates, destructible objects, cloth and ragdolls, along with many other effects.

1.2 What Is a Physics Engine?

Although physics in games is more than thirty years old, there has been a distinct change in recent years in the way that physics is implemented. Originally each effect was programmed for its own sake, creating a game with only the physics needed for that title. If a game needed arrows to follow trajectories, then the equation of the trajectory could be programmed into the game. It would be useless for simulating anything but the trajectory of arrows, but it would be perfect for that.

This is fine for simple simulations, where the amount of code is small and the scope of the physics is quite limited. As we'll see, a basic particle system can be programmed in only a hundred or so lines of code. When the complexity increases, it can be difficult to get straight to a believable physical effect. In the original Half-Life game, for example, you can push crates around, but the physics code isn't quite right, and the way the crates move looks odd. The difficulty of getting physics to look good, combined with the need for almost the same effects in game after game, encouraged developers to look for general solutions that could be reused.

Reusable technology needs to be quite general: a ballistics simulator that will only deal with arrows can have the behavior of arrows hard-coded into it. If the same code needs to cope with bullets too, then the software needs to abstract away from particular projectiles and simulate the general physics that they all have in common. This is what we call a "physics engine": a common piece of code that knows about physics in general but isn't programmed with the specifics of each game's scenario.

There is an obvious hole here. If we had special code for simulating an arrow, then we need nothing else to simulate an arrow. If we have a general physics engine for simulating any projectile, and we want to simulate an arrow, we also need to tell the engine the characteristics of the thing we are simulating. We need the physical properties of arrows, or bullets, or crates, and so on.

This is an important distinction. The physics engine is basically a big calculator: it does the mathematics needed to simulate physics. But it doesn't know what needs to be simulated. In addition to the engine we also need game-specific data that represents the game level.

Although we'll look at the kind of data we need throughout this book, I won't focus on how the data gets into the game. In a commercial game there will likely be some kind of level-editing tool that allows level designers to place crates, flags, ragdolls, or airplanes: to set their weight, the way they move through the air, their buoyancy, and so on.

The physics engine we'll be developing throughout this book needs gradually more and more data to drive it. I'll cover in depth what kind of data this is, and what reasonable values it can take, but for our purposes we will assume this data can be provided to the engine. It is beyond the scope of the book to consider the toolchain that developers use to author these properties for the specific objects in their game.

1.2.1 Advantages of a Physics Engine

There are two compelling advantages to using a physics engine in your games. First there is the time saving. If you intend to use physics effects in more than one game (and you'll probably be using them in most of your games from now on), then putting the effort into creating a physics engine now pays off when you can simply import it into each new project. A lightweight, general-purpose physics system, of the kind we develop in this book, doesn't have to be difficult to program either. A couple of thousand lines of code will set you up for most of the game effects you need.

The second reason is quality. You will most likely be including more and more physical effects in your game as time goes on. You could implement each of these as you need it: building a cloth simulator for capes and flags, and a water simulator for floating boxes, and a separate particle engine. Each might work perfectly, but you would have a very difficult time combining their effects. When the character with a flowing cloak comes to stand in the water, how will his clothes behave? If the cloak keeps blowing in the wind even when underwater, then the illusion is spoiled.

A physics engine provides you with the ability to have effects interact in believable ways. Remember the movable crates in Half-Life 1? They formed the basis of only one or two puzzles in the game. When it came to Half-Life 2, crate physics was replaced by a full physics engine. This opens up all kinds of new opportunities. The pieces of a shattered crate float on water; objects can be stacked, used as movable shields, and so on.

It's not easy to create a physics engine to cope with water, wind, and clothes, but it's much easier than trying to take three separate ad hoc chunks of code and make them look good together in all situations.

1.2.2   Weaknesses of a Physics Engine

This isn't to say that a physics engine is a panacea. There are reasons that you might not want to use a full physics engine in your game.

The most common reason is one of speed. A general-purpose physics engine is quite processor intensive. Because it has to be general, it can make no assumptions about the kinds of objects it is simulating. When you are working with a very simple game environment, this generality can mean wasted processing power. This isn't an issue on modern consoles or the PC, but on hand-held devices such as phones and PDAs it can be significant. You could create a pool game using a full physics engine on a PC, but the same game on a mobile phone would run faster with some specialized pool physics.

The need to provide the engine with data can also be a serious issue. In a game I worked on recently, we needed no physics other than flags waving in the wind. We could have used a commercial physics engine (one was available to the developer), but the developer would have had to calculate the properties of each flag, its mass, springiness, and so on. This data would then need to be fed into the physics engine to get it to simulate the flags.

There was no suitable level-design tool that could be easily extended to provide this data, so instead we created a special bit of code just for flag simulation; the characteristics of flags were hard-coded in the software, and the designer didn't have to do anything special to support it. We avoided using a physics engine because special-case code was more convenient.

A final reason to avoid physics engines is scope. If you are a one-person hobbyist working on your game in the evenings, then developing a complete physics solution might take time from improving other aspects of your game: the graphics or gameplay, for example. On the other hand, even amateur games need to compete with

commercial titles for attention, and top-quality physics is a must for a top-quality title of any kind.

1.3 Approaches to Physics Engines

There are several different approaches to building a physics engine, ranging from the very simple (and wrong) to the cutting-edge physics engines of top middleware companies. Creating a usable engine means balancing the complexity of the programming task with the sophistication of the effects you need to simulate.

There are a few broad distinctions we can make to categorize different approaches, as described in the following sections.

1.3.1 Types of Object

The first distinction is between engines that simulate full rigid bodies and so-called mass-aggregate engines. Rigid-body engines treat objects as a whole and work out the way they move and rotate. A crate is a single object and can be simulated as a whole. Mass-aggregate engines treat objects as if they were made up of lots of little masses. A box might be simulated as if it were made up of eight masses, one at each corner, connected by rods.

Mass-aggregate engines are easier to program because they don't need to understand rotations. Each mass is located at a single point, and the equations of the motion can be expressed purely in terms of linear motion. The whole object rotates naturally as a result of the constrained linear motion of each component.

Because it is very difficult to make things truly rigid in a physics engine, it is difficult to make really firm objects in a mass-aggregate system. Our eight-mass crate will have a certain degree of flex in it. To make this invisible to the player, extra code is needed to reconstruct the rigid box from the slightly springy set of masses. While the basic mass-aggregate system is very simple to program, these extra checks and corrections are more hit and miss, and I don't feel that they are worth the effort.

Fortunately we can extend a mass-aggregate engine into a full rigid-body system simply by adding rotations. In this book we will develop a mass-aggregate physics engine on the way to a full rigid-body physics engine. Because we are heading for a more robust engine, I won't spend the time creating the correction code for springy aggregates.

1.3.2 Contact Resolution

The second distinction involves the way in which touching objects are processed. As we'll see in this book, a lot of the difficulty in writing a rigid-body physics engine is simulating contacts: locations where two objects touch or are connected. This includes objects resting on the floor, objects connected together, and to some extent collisions.

One approach is to handle these contacts one by one, making sure each works well on its own. This is called the "iterative approach," and it has the advantage of speed. Each contact is fast to resolve, and with only a few tens of contacts the whole set can be resolved quickly. It has the downside that one contact can affect another, and sometimes these interactions can be significant. This is the easiest approach to implement and can form the basis of more complex methods. It is the technique we will use in the engine in this book.

A more physically realistic way is to calculate the exact interaction between different contacts and calculate an overall set of effects to apply to all objects at the same time. This is called a "Jacobian-based" approach, but it suffers from being very time consuming: the mathematics involved is very complex, and solving the equations can involve millions of calculations. Even worse, in some cases there is simply no valid answer, and the developer needs to add special code to fall back on when the equations can't be solved. Several physics middleware packages use this approach, and each has its own techniques for solving the equations and dealing with inconsistencies.

A third option is to calculate a new set of equations based on the contacts and constraints between objects. Rather than use Newton's laws of motion, we can create our own set of laws for just the specific configuration of objects we are dealing with. These equations will be different for every frame, and most of the effort for the physics engine goes into creating them (even though solving them is no picnic either). This is called a "reduced coordinate approach." Some physics systems have been created with this approach, and it is the most common one used in engineering software to do really accurate simulation. Unfortunately, it is very slow and isn't very useful in games applications, where speed and believability are more important than accuracy.

We'll return to other approaches in chapter 18, after we've looked at the physics involved in the first approach.

1.3.3   Impulses and Forces

The third distinction lies in how the engine actually resolves contacts. This takes a little explaining, so bear with me.

When a book rests on a table, the table is pushing it upward with a force equal to the gravity pulling it down. If there were no force from the table to the book, then the book would sink into the table. This force is constantly pushing up on the book as long as it is sitting there. The speed of the book doesn't change.

Contrast this with the way a ball bounces on the ground. The ball collides with the ground, the ground pushes back on the ball, accelerating it upward until it bounces back off the floor with an upward velocity. This change in velocity is caused by a force, but the force acts for such a small fraction of a second that it is easier to think of it as simply a change in velocity. This is called an "impulse."

Some physics engines use forces for resting contacts and impulses for collisions. This is relatively rare because it involves treating forces and impulses differently. More

commonly physics engines treat everything as a force: impulses are simply forces acting over a very small period of time. This is a force-based engine, and it works in the way the real world does. Unfortunately the mathematics of forces is more difficult than the mathematics of impulses. Those engines that are force based tend to be those with a Jacobian or reduced coordinate approach. Several of the leading middleware physics offerings are force based.

Other engines use impulses for everything: the book resting on the table is kept there by lots of miniature collisions rather than a constant force. This is, not surprisingly, called an "impulse-based" engine. In each frame of the game the book receives a little collision that keeps it on the surface of the table until the next frame. If the frame-rate slows down dramatically, things lying on surfaces can appear to vibrate. Under most circumstances, however, it is indistinguishable from a force-based approach. This is the approach we will use in this book: it is easy to implement and has the advantage of being very flexible and adaptable. It has been used in several middleware packages, in the majority of the in-house physics systems I have seen developers create, and it has proved itself in many commercial titles.

1.3.4 What We're Building

In this book I will cover in depth the creation of a rigid-body, iterative, impulse-based physics engine that I've called Cyclone. The engine has been written specifically for the book, although it is based on a commercial physics engine I was involved with writing a few years ago. I am confident that the impulse-based approach is best for developing a simple, robust, and understandable engine for a wide range of different game styles and for using as a basis for adding more complex and exotic features later on.

As we move through the book, I will give pointers for different approaches, and chapter 18 will give some background to techniques for extending the engine to take advantage of more complex simulation algorithms. While we won't cover other approaches in the same depth, the engine is an excellent starting point for any kind of game physics. You will need to understand the content of this book to be able to create a more exotic system.

1.4 The Mathematics of Physics Engines

Creating a physics engine involves a lot of mathematics. If you're the kind of person who is nervous about math, then you may find some of this material hard going. I've tried throughout the book to step through the mathematical background slowly and clearly.

If you have difficulty following the mathematics, don't worry: you can still use the accompanying source code for the corresponding bit. While it is better to understand all of the engine in case you need to tweak or modify it you can still implement and use it quite successfully without fully comprehending the math.

As a quick reference, the mathematical equations and formulae in the book are also collected together in appendix D for easy location when programming a game.

If you are an experienced game developer, then the chances are you will know a fair amount of three-dimensional (3D) mathematics: vectors, matrices, and linear algebra. If you are relatively new to games, then these topics may be beyond your comfort zone.

In this book I will assume you know some mathematics, and I will cover the rest. If I assume you know something, but you aren't confident in using it, then it would be worth getting hold of a reference book or looking for a web tutorial before proceeding, so you can easily follow the text.

1.4.1 THE MATH YOU NEED TO KNOW

I'm going to assume every potential physics developer knows some mathematics.

The most important thing to be comfortable with is algebraic notation. I will introduce new concepts directly in notation, and if you flick through this book, you will see many formulae written into the text.

I'll assume you are happy to read an expression like

$$x = \frac{4}{t} \sin \theta^2$$

and can understand that x, t, and θ are variables and how to combine them to get a result.

I will also assume you know some basic algebra: you should be able to understand that, if the preceding formula is correct, then

$$t = \frac{4}{x} \sin \theta^2$$

These kinds of algebraic manipulations will pop up all through the book without explanation.

Finally I'll assume you are familiar with trigonometry and coordinate geometry: sines, cosines, and tangents, and their relationship to right-angled triangles and to two-dimensional geometry in general. In particular, you should know that if we have the triangle shown in figure 1.1, then these formulae hold:

$$b = a \sin \theta$$
$$c = a \cos \theta$$
$$b = c \tan \theta$$
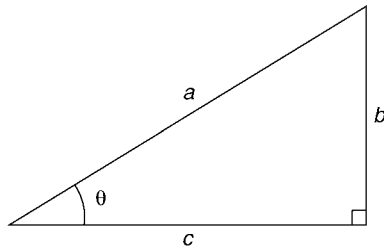
FIGURE 1.1   Trigonometry and coordinate geometry.

Especially when a is of length 1, we will use these results tens of times in the book without further discussion.

1.4.2 THE MATH WE'LL REVIEW

Because the experience of developers varies so much, I will not assume that you are familiar with three-dimensional mathematics to the same extent. This isn't taught in high schools and is often quite specialized to computer graphics. If you have been a game developer for a long time, then you will no doubt be able to skip through these reviews as they arise.

We will cover the way vectors work in the next chapter, including the way a three-dimensional coordinate system relates to the two-dimensional mathematics of high school geometry. I will review the way vectors can be combined, including the scalar and vector product, and their relationship to positions and directions in three dimensions.

We will also review matrices. Matrices are used to transform vectors: moving them in space or changing their coordinate systems. We will also see matrices called "tensors" at a couple of points, which have different uses but the same structure. We will review the mathematics of matrices, including matrix multiplication, transformation of vectors, matrix inversion, and basis changes.

These topics are fundamental to any kind of 3D programming and are used extensively in graphics development and in many AI algorithms too. Most of you will be quite familiar with them, and there are comprehensive books available that cover them in great depth.

Each of these topics is reviewed lightly once in this book, but afterward I'll assume that you are happy to see the results used directly. They are the bread-and-butter topics for physics development, so it would be inconvenient to step through them each time they arise.

If you find later sections difficult, it is worth flicking back in the book and rereading the reviews, or finding a more comprehensive reference to linear algebra or computer graphics and teaching yourself how they work.

1.4.3 The Math We'll Introduce

Finally there is a good deal of mathematics that you may not have discovered unless you have done some physics programming in the past. This is the content I'll try not to assume you know and will cover in some more depth.

At the most well-known end of the spectrum this includes the quaternions, a vectorlike structure that represents the orientation of an object in three-dimensional space. We will take some time to understand why such a strange structure is needed and how it can be manipulated: converted into a matrix, combined with other quaternions, and affected by rotations.

We will also need to cover vector calculus: the way vectors change with time and through space. Most of the book requires only simple calculus—numerical integration and first-order differentiation. The more complex physics approaches of chapter 18 get considerably more exotic, including both partial differentials and differential operators. Fortunately we will have completely built the physics engine by this point, so the content is purely optional.

Finally we will cover some more advanced topics in matrix manipulation. In particular, a difficult chunk of the engine development involves working with changes-of-basis for matrices. This kind of manipulation is rarely needed in graphics development, so it will be covered in some depth in the relevant section.

1.5 The Source Code in This Book

Throughout the book the source code from the Cyclone physics engine is given in the text. The complete engine is available on the accompanying CD, but repeating the code in the text has allowed me to comment more fully on how it works.

The latest Cyclone source, including errata and new features, is available on its own site—*www.procyclone.com*. Check the site from time to time for the latest release of the package.

In each section of the book we will cover the mathematics or concepts needed and then see them in practice in code. I encourage you to try to follow the equations or algorithms in the code, and to find how they have been implemented.

I have used C++ throughout the code. This is by far the most common programming language used for serious game development worldwide. Even those few studios that use other languages for building the final game (including LISP, Lua, and Python) have most of their core engines written in C++.

Some developers still swear by C and believe that using anything else for game development is tantamount to heresy. C compilers have traditionally been more efficient at producing fast-executing code, and the same overhead that makes C++ a more powerful language (in the sense that it does more for you automatically) has the reputation of slowing down the final program. In reality this is a dying mindset rooted in dubious fact. With modern optimizing C++ compilers, and a programming style that takes into consideration problems with the C++ libraries (avoiding some implementations of the hashtable, for example), the final result can be at least as fast and

sometimes faster than C. It is common to code the most speed-critical sections of a game (typically the lowest-level matrix and vector mathematics) in assembly language to take advantage of the architecture of the processor it will be running on. This is well beyond the scope of this book and should only be attempted if you have a decent code profiler telling you that a speed-up would be useful.

If you are developing in a language other than C++, then you will need to translate the code. Appendix C gives some guidance for efficient ways to convert the code into a selection of common languages—ways to ensure that the language features run at a reasonable speed.

I have used an object-oriented design for the source code and have always tried to err on the side of clarity. The code is contained within a `cyclone` namespace, and its layout is designed to make naming clashes unlikely.

There are many parts of the engine that can be optimized, or rewritten to take advantage of mathematics hardware on consoles, graphics cards, and some PC processors. If you need to eke out every ounce of speed from the engine, you will find you need to optimize some of the code to make it less clear and more efficient. Chances are, however, it will be perfectly usable as is. It has a strong similarity to code I have used in real game development projects, which has proved to be fast enough to cope with reasonably complex physics tasks.

There are a number of demonstration programs in the source code, and I will use them as case studies in the course of this book. The demonstrations were created to show off physics rather than graphics, so I've tried to use the simplest practical graphics layer. The source code is based on the GLUT toolkit, which wraps OpenGL in a platform-independent way. The graphics tend to be as simple as possible, calling GLUT's built-in commands for drawing cubes, spheres, and other primitives. This selection doesn't betray any bias on my part (in fact I find OpenGL difficult to optimize for large-scale engines), and you should be able to transfer the physics so that it works with whatever rendering platform you are using.

The license for your use of the source code is liberal, and designed to allow it to be used in your own projects, but it is not copyright-free. Please read through the software license on the CD for more details.

It is my hope that, although the source code will provide a foundation, you'll implement your own physics system as we go (and it therefore will be owned entirely by you). I make decisions throughout this book about the implementation, and the chances are you'll make different decisions at least some of the time. My aim has been to give you enough information to understand the decision and to go a different route if you want to.

1.6 How this Book Is Structured

We will build our physics engine in stages, starting with the simplest engine that is useful and adding new functionality until we have a system capable of simulating almost anything you see in a modern game.

This book is split into the following six parts.
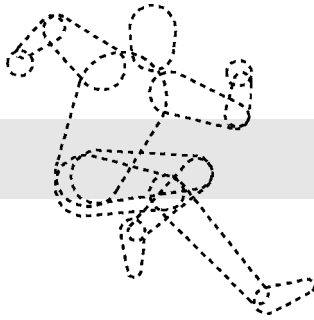
- *Particle Physics* looks at building our initial physics engine, including the vector mathematics and the laws of motion that support it.

- *Mass-Aggregate Physics* turns the particle physics engine into one capable of simulating any kind of object by connecting masses together with springs and rods.

- *Rigid-Body Physics* introduces rotation and the added complexity of rotational forces. Overall the physics engine we end up with is less powerful than the mass-aggregate system we started with, but it is useful in its own right and as a basis for the final stage.

- *Collision Detection* takes a detour from building engines to look at how the collisions and contacts are generated. A basic collision detection library is built that allows us to look at the general techniques.

- *Contact Physics* is the final stage of our engine, adding collision and resting contacts to the engine and allowing us to apply the result to almost any game.

- *What Comes Next* looks beyond the engine we have. In chapter 18 we look at how to extend the engine to take advantage of other approaches, without providing the detailed step-by-step source code to do so.

As we develop each part, the content will be quite theoretical, and it can be difficult sometimes to immediately see the kinds of physical effects that the technology supports. At the end of each part where we add to the engine, there is a chapter covering ways in which it may be used in a game. As we go through the book, we start with engines controlling fireworks and bullets, and end up with ragdolls and cars.

PART I

Particle Physics

This page intentionally left blank

THE MATHEMATICS
OF PARTICLES

B efore we look at simulating the physics of particles, this chapter reviews three-dimensional mathematics. In particular it looks at vector mathematics and vector calculus—the fundamental building blocks on which all our physics code will be built. I'll avoid some of the harder topics we'll only need later. Matrices and quaternions, for example, will not be needed until chapter 9, so I'll postpone reviewing them until that point.

2.1 VECTORS

Most of the mathematics we are taught at school deals with single number—numbers to represent the number of apples we have or the time it takes for a train to make a journey, or the numerical representation of a fraction. We can write algebraic equations that tell us the value of one number in terms of others. If $x = y^2$ and $y = 3$, then we know $x = 9$. This kind of single number on its own is called a "scalar value."

Mathematically a vector is an element of a vector space: a structure that displays certain mathematical properties for addition and multiplication. For our purposes the only vector spaces we're interested in are regular (called Euclidean) 2D and 3D spaces. In this case the vector represents a position in those spaces.

Vectors are usually represented as an ordered list of numbers that can be treated in a similar way to a single number in an algebraic equation. If y is a vector (let's say it contains the numbers 2 and 3), and if $x = 2y$, then x will also be a vector of two numbers, in this case 4 and 6. Vectors can undergo the same mathematical operations

$$a = \begin{bmatrix} 1.4 \\ 2.4 \\ 3 \end{bmatrix} \qquad b = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}$$
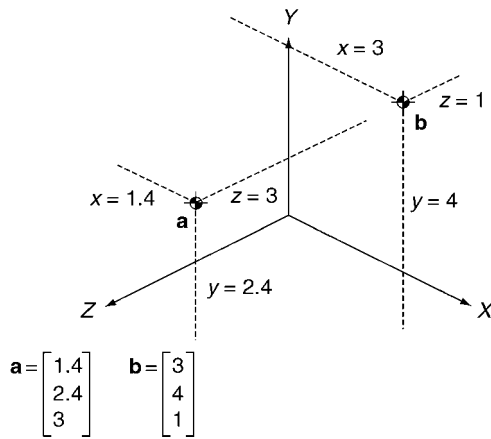
FIGURE 2.1   Three-dimensional coordinates.

as scalars, including multiplication, addition, and subtraction (although the way they do these is slightly different from scalars, and the result isn't always another vector; we'll return to this later).

Note that a vector in this sense, and throughout most of this book, refers only to this mathematical structure. Many programming languages have a `vector` data structure which is some kind of growable array. The name comes from the same source (a set of values rather than just one), but that's where the similarities stop. In particular most languages do not have a built-in vector class to represent the kind of vector we are interested in. On the few occasions in this book where I need to refer to a growable array, I will call it that, to keep the name "vector" reserved for the mathematical concept.

One convenient application of vectors is to represent coordinates in space. Figure 2.1 shows two locations in 3D space. The position can be represented by three coordinate values, one for the distance from a fixed origin point along three axes at right angles to one another. This is a Cartesian coordinate system, named for the mathematician and philosopher Rene Descartes who invented it.

We group the three coordinates together into a vector, written as

$$a = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

where x, y, and z are the coordinate values along the X, Y, and Z axes. Note the *a* notation. This indicates that a is a vector: we will use this throughout the book to make it easy to see what is a vector and what is just a plain number.

Every vector specifies a unique position in space, and every position in space has only one corresponding vector. We can and will use only vectors to represent positions in space.

We can begin to implement a class to represent vectors. I have called this class Vector3 to clearly separate it from any other Vector class in your programming language (seeing the name Vector on its own is particularly confusing for Java programmers).

Excerpt from include/cyclone/precision.h

```
namespace cyclone {

    /**
     * Defines a real number precision. Cyclone can be compiled in
     * single- or double-precision versions. By default single precision
     * is provided.
     */
    typedef float real;
}
```

Excerpt from include/cyclone/core.h

```
namespace cyclone {

    /**
     * Holds a vector in 3 dimensions. Four data members are allocated
     * to ensure alignment in an array.
     */
    class Vector3
    {
    public:
         /** Holds the value along the x axis. */
        real x;

        /** Holds the value along the y axis. */
        real y;

        /** Holds the value along the z axis. */
        real z;

    private:
        /** Padding to ensure 4-word alignment. */
        real pad;

    public:
        /** The default constructor creates a zero vector. */
        Vector3() : x(0), y(0), z(0) {}
```

```
        /**
         * The explicit constructor creates a vector with the given
         * components.
         */
        Vector3(const real x, const real y, const real z)
            : x(x), y(y), z(z) {}

        /** Flips all the components of the vector. */
        void invert()
        {
            x = -x;
            y = -y;
            x = -z;
        }

    };

}
```

There are a few things to note about this source code.

■ All the code is contained within the `cyclone` namespace, as promised in the introduction to the book. This makes it easier to organize code written in C++, and in particular it ensures that names from several libraries will not clash. Wrapping all the code samples in the namespace declaration is a waste of time, however; so in the remaining excerpts in this book, I will not show the namespace explicitly.

■ Also to avoid clashing names, I have placed the header files in the directory `include/cyclone/`, with the intention of having the `include/` directory on the include path for a compiler (see your compiler's documentation for how to achieve this). This means that to include a header we will use an include of the format:

```
#include <cyclone/core.h>
```

or

```
#include "cyclone/core.h"
```

I find this to be a useful way of making sure the compiler knows which header to bring in, especially with large projects that are using multiple libraries, several of which may have the same name for some header files (I have at least four `math.h` headers that I use regularly in different libraries—which is part of my motivation for putting our mathematics code in a header called `core.h`).

- The source code listings show the line numbers. As you can see, the line numbers are not continuous. They represent excerpts from the source code on the CD. In the full source code there are additional comments, more functions, and bits of code we've yet to meet. Each of the source code listings I give should be a self-contained chunk of code. In the preceding code, for example, you shouldn't need any other code to make the listings work; in other excerpts you can add the given code to any previous code for the same class (when we come to add some additional functions to the `Vector3` class, for example). I've added the line numbers simply to allow you to locate where each element is implemented on the CD.

- I have used `real` rather than `float` to reserve the storage for my vector components. The real data type is a `typedef`, contained in its own file (`precision.h`). I've done this to allow the engine to be rapidly compiled in different precisions. In most of the work I've done, `float` precision is fine, but it can be a huge pain to dig through all the code if you find you need to change to `double` precision later. You may have to do this if you end up with numerical rounding problems that won't go away (they are particularly painful if you have objects with a wide range of different masses in the simulation). By consistently using the `real` data type, we can easily change the precision of the whole engine by changing the type definition once. We will add to this file additional definitions for functions (such as `sqrt`) that come in both `float` and `double` form.

- I've added an extra piece of data into the vector structure, called `pad`. This isn't part of the mathematics of vectors and is there purely for performance. On many machines four floating-point values sit more cleanly in memory than three (memory is optimized for sets of four words), so noticeable speed-ups can be achieved by adding this padding.

  Your physics engine shouldn't rely on the existence of this extra value for any of its functionality. If you are programming for a machine that you know is highly memory limited, and doesn't optimize in sets of four words, then you can remove `pad` with impunity.

2.1.1 THE HANDEDNESS OF SPACE

If you are an experienced game developer, you will have spied a contentious assumption in figure 2.1. The figure shows the three axes arranged in a right-handed coordinate system. There are two different ways we can arrange three axes at right angles to one another: in a left-handed way or a right-handed way,[1] as shown in figure 2.2.

You can tell which is which by using your hands: make a gun shape with your hand, thumb and extended forefinger at right angles to one another. Then, keeping your ring finger and pinky curled up, extend your middle finger so it is at right angles

---

[1] Strictly speaking this handedness is called "chirality," and each alternative is an "enantiomorph," although those terms are rarely if ever used in game development.
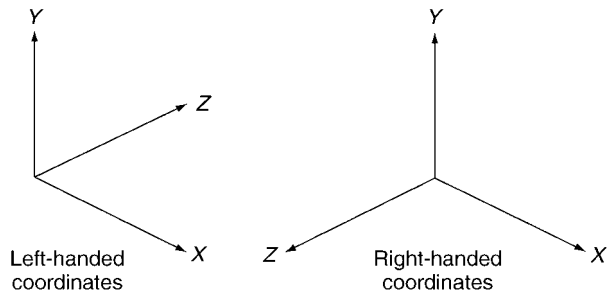
FIGURE 2.2 Left- and right-handed axes.

to the first two. If you label your fingers in order with the axes (thumb is X, forefinger Y, and middle finger Z), then you have a complete set of axes, either right- or left-handed. Some people prefer to think of this in terms of the direction that a screw is turned, but I find making axes with my hands much simpler.

Different game engines, rendering toolkits, and modeling software use either left- or right-handed axes. There is no dependable standard. DirectX favors a left-handed coordinate system, while OpenGL favors a right-handed one, as does the Renderware middleware system. XBox and XBox 360, being DirectX based, are left-handed; GameCube, being rather OpenGL-like, is right-handed; and PlayStation's sample code is right-handed, although most developers create their own rendering code. On any platform you can actually use either with a bit more effort (this is how Renderware uses a right-handed system even on the XBox, for example). For a detailed explanation of different systems and converting between them see Eberly [2003].

There are relatively few places where it matters which system we use: it certainly doesn't change the physics code in any way. I have (fairly arbitrarily) chosen right-handed coordinates throughout this book. Because the code on the CD is designed to work with OpenGL, this makes the sample code slightly easier. In addition most of the commercial engines I've worked with (both middleware and developers' own) have been right-handed.

If you are working on a DirectX-only project and are keen to stay with a left-handed system, then you'll need to make the occasional adjustment in the code. I'll try to indicate places where this is the case.

2.1.2 Vectors and Directions

There is another interpretation of a vector. A vector can represent the change in position. Figure 2.3 shows an object that has moved in space from position a_0 to a_1. We can write down the change in position as a vector where each component of the

$$\boldsymbol{a} = \begin{bmatrix} 1.6 \\ -0.2 \\ -2 \end{bmatrix}$$
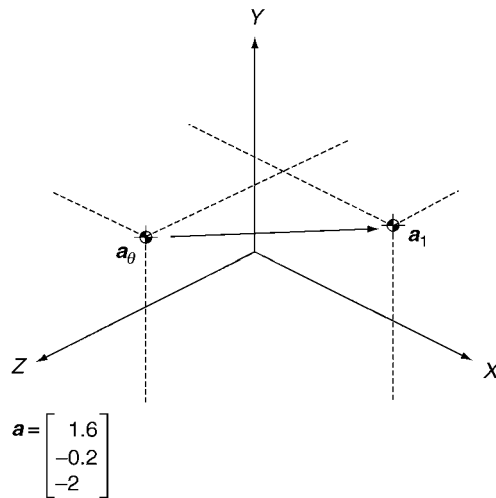
FIGURE 2.3   A vector as a movement in space.

vector is the change along each axis. So

$$\boldsymbol{a} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix}$$

where Δx is the change in the position along the X axis from $\boldsymbol{a}_0$ to $\boldsymbol{a}_1$, given by

$$\Delta x = x_1 - x_0$$

where x_0 is the X coordinate of $\boldsymbol{a}_0$ and x_1 is the X coordinate of $\boldsymbol{a}_1$. Similarly for Δy and Δz.

Position and change in position are really two sides of the same coin. We can think of any position as a change of position from the origin (written as **0**, where each component of the vector is zero) to the target location.

If we think in terms of the geometry of a vector being a movement from the origin to a point in space, then many of the mathematical operations we'll meet in this chapter have obvious and intuitive geometric interpretations. Vector addition and subtraction, multiplication by a scalar, and different types of multiplication can all be understood in terms of how these movements relate. When drawn as in figure 2.3, the visual representation of an operation is often much more intuitive than the mathematical explanation. We'll consider this for each operation we meet.

A change in position, given as a vector, can be split into two elements:

$$a = d\mathbf{n} \qquad\qquad [2.1]$$

where d is the straight-line distance of the change (called the "magnitude" of the vector) and $\mathbf{n}$ is the direction of the change. The vector $\mathbf{n}$ represents a change, whose straight-line distance is always 1, in the same direction as the vector $\mathbf{a}$.

We can find d using the three-dimensional version of Pythagoras's theorem, which has the formula

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

where x, y, and z are the three components of the vector and $|\mathbf{a}|$ is the magnitude of a vector.

We can use equation 2.1 to find $\mathbf{n}$:

$$\widehat{\mathbf{a}} = \mathbf{n} = \frac{1}{d}\mathbf{a} \qquad\qquad [2.2]$$

where $\widehat{\mathbf{a}}$ is a common (but not universal) notation for the unit-length direction of $\mathbf{a}$. The equation is sometimes written as

$$\widehat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|}$$

The process of finding just the direction $\mathbf{n}$ from a vector is called "normalizing"; the result is sometimes called the "normal" form of the vector (i.e., $\mathbf{n}$ is the normal form of $\mathbf{a}$ in the preceding equations). It is a common requirement in several algorithms.

We can add functions to find the magnitude of the vector and its direction and perform a normalization:

Excerpt from include/cyclone/precision.h

```
/** Defines the precision of the square root operator. */
#define real_sqrt sqrtf
```

Excerpt from include/cyclone/core.h

```
class Vector3
{
    // ... Other Vector3 code as before ...

    /** Gets the magnitude of this vector. */
    real magnitude() const
    {
        return real_sqrt(x*x+y*y+z*z);
    }
```

```
/** Gets the squared magnitude of this vector. */
real squareMagnitude() const
{
    return x*x+y*y+z*z;
}

/** Turns a non-zero vector into a vector of unit length. */
void normalize()
{
    real l = magnitude();
    if (l > 0)
    {
        (*this)*=((real)1)/l;
    }
}
};
```

Notice that I've also added a function to calculate the square of the magnitude of a vector. This is a faster process because it avoids the call to sqrt, which can be slow on some machines. There are many cases where we don't need the exact magnitude and where the square of the magnitude will do. For this reason it is common to see a squared magnitude function in a vector implementation.

2.1.3 SCALAR AND VECTOR MULTIPLICATION

In the normalization equations I have assumed we can multiply a scalar $(1/d)$ by a vector. This is a simple process, given by the formula

$$k\boldsymbol{a} = k \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} kx \\ ky \\ kz \end{bmatrix}$$

In other words we multiply a vector by a scalar by multiplying all the components of the vector by the scalar.

To divide a vector by a scalar, we make use of the fact that

$$a \div b = a \times \frac{1}{b}$$

so

$$\frac{\boldsymbol{a}}{k} = \frac{1}{k}\boldsymbol{a}$$

which is how we arrived at the normalization equation 2.2 from equation 2.1.

This formula lets us define the additive inverse of a vector:

$$-\boldsymbol{a} = -1 \times \boldsymbol{a} = \begin{bmatrix} -x \\ -y \\ -z \end{bmatrix}$$

We can overload the multiplication operator *= in C++ to support these operations, with the following code in the Vector3 class.

Excerpt from `include/cyclone/core.h`

```
class Vector3
{
    // ... Other Vector3 code as before ...

    /** Multiplies this vector by the given scalar. */
    void operator*=(const real value)
    {
        x *= value;
        y *= value;
        z *= value;
    }

    /** Returns a copy of this vector scaled to the given value. */
    Vector3 operator*(const real value) const
    {
        return Vector3(x*value, y*value, z*value);
    }
};
```

Geometrically this operation scales the vector, changing its length by the scalar. This is shown in figure 2.4.

2.1.4 Vector Addition and Subtraction

Geometrically, adding two vectors together is equivalent to placing them end to end. The result is the vector from the origin of the first to the end of the second, shown in figure 2.5. Similarly, subtracting one vector from another places the vectors end to end, but the vector being subtracted is placed so that its end touches the end of the first. In other words, to subtract vector $\boldsymbol{b}$ from vector $\boldsymbol{a}$ we first go forward along $\boldsymbol{a}$, and then go backward along $\boldsymbol{b}$.
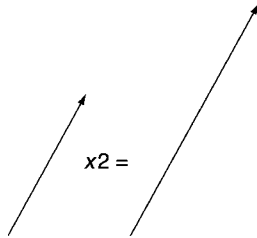
FIGURE 2.4   The geometry of scalar–vector multiplication.
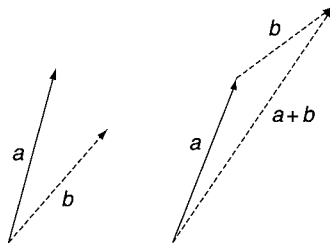
FIGURE 2.5   The geometry of vector addition.

In code it is very easy to add vectors together or subtract them. For two vectors a and b, their sum is given by

$$a + b = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix}$$

where a_x, a_y, and a_z are the x, y, and z components of the vector a: we will normally use this notation rather than x, y, and z to avoid confusion when dealing with more than one vector.

Vector addition is achieved by adding the components of the two vectors together. This can be implemented for the + operator:

```
_____ Excerpt from include/cyclone/core.h _____
class Vector3
{
    // ... Other Vector3 code as before ...

    /** Adds the given vector to this. */
    void operator+=(const Vector3& v)
```

```
    {
        x += v.x;
        y += v.y;
        z += v.z;
    }

    /**
     * Returns the value of the given vector added to this.
     */
    Vector3 operator+(const Vector3& v) const
    {
        return Vector3(x+v.x, y+v.y, z+v.z);
    }
};
```

In the same way vector subtraction is also performed by subtracting the components of each vector:

$$
a - b = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}
$$

which is implemented in the same way as addition:

———— **Excerpt from include/cyclone/core.h** ————

```
class Vector3
{
    // ... Other Vector3 code as before ...

    /** Subtracts the given vector from this. */
    void operator-=(const Vector3& v)
    {
        x -= v.x;
        y -= v.y;
        z -= v.z;
    }

    /**
     * Returns the value of the given vector subtracted from this.
     */
    Vector3 operator-(const Vector3& v) const
    {
        return Vector3(x-v.x, y-v.y, z-v.z);
```

```
        }
};
```

A final, useful version of this is to update a vector by adding a scaled version of another vector. This is simply a combination of vector addition and the multiplication of a vector by a scalar:

$$\boldsymbol{a} + c\boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + c \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x + cb_x \\ a_y + cb_y \\ a_z + cb_z \end{bmatrix}$$

We could do this in two steps with the preceding functions, but having it in one place is convenient:

```
 ────── Excerpt from include/cyclone/core.h ──────
class Vector3
{
    // ... Other Vector3 code as before ...

    /**
     * Adds the given vector to this, scaled by the given amount.
     */
    void addScaledVector(const Vector3& vector, real scale)
    {
        x += vector.x * scale;
        y += vector.y * scale;
        z += vector.z * scale;
    }
};
```

2.1.5 Multiplying Vectors

Seeing how easy it is to add and subtract vectors may lull you into a false sense of security. When we come to multiply two vectors together, life gets considerably more complicated.

There are several ways of multiplying two vectors, and whenever we produce a formula involving vector multiplication, we have to specify which type of multiplication to use. In algebra, for scalar values we can denote the product (multiplication) of two values by writing them together with no intervening symbol (i.e., *ab* means $a \times b$). With vectors this usually denotes one type of multiplication that we need not cover (the vector direct product—see a good mathematical encyclopedia for information). I will not write **ab** but will show the kind of product to use with a unique operator symbol.

2.1.6   THE COMPONENT PRODUCT

The most obvious product is the least useful: the component product, written in this book as ∘ (it does not have a universal standard symbol the way the other products do). It is used in several places in the physics engine, but despite being quite obvious, it is rarely mentioned at all in books about vector mathematics.

This is because it doesn't have a simple geometric interpretation: if the two vectors being multiplied together represent positions, then it isn't clear geometrically how their component product is related to their locations. This isn't true of the other types of product.

The component product is formed in the same way as vector addition and subtraction—by multiplying each component of the vector together:

$$a \circ b = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \circ \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x \\ a_y b_y \\ a_z b_z \end{bmatrix}$$

Note that the end result of the component product is another vector. This is exactly the same as for vector addition and subtraction, and for multiplication by a scalar: all end up with a vector as a result.

Because it is not commonly used, we will implement the component product as a method rather than an overloaded operator. We will reserve overloading the * operator for the next type of product. The method implementation looks like this:

```
                    ─── Excerpt from include/cyclone/core.h ───
class Vector3
{
    // ... Other Vector3 code as before ...

    /**
     * Calculates and returns a component-wise product of this
     * vector with the given vector.
     */
    Vector3 componentProduct(const Vector3 &vector) const
    {
        return Vector3(x * vector.x, y * vector.y, z * vector.z);
    }

    /**
     * Performs a component-wise product with the given vector and
     * sets this vector to its result.
     */
    void componentProductUpdate(const Vector3 &vector)
    {
```

```
            x *= vector.x;
            y *= vector.y;
            z *= vector.z;
        }
};
```

2.1.7 THE SCALAR PRODUCT

By far the most common product of two vectors is the "scalar product." It is different from any of our previous vector operations in that its result is not a vector but rather a single scalar value (hence its name). It is written using a dot symbol: $\boldsymbol{a} \cdot \boldsymbol{b}$, and so is often called the "dot product." For reasons beyond this book it is also called, more mathematically, the "inner product"—a term I will not use again.

The dot product is calculated with the formula

$$\boldsymbol{a} \cdot \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \cdot \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = a_x b_x + a_y b_y + a_z b_z \qquad [2.3]$$

In my vector class I have used the multiplication operator * to represent the dot product (it looks quite like a dot, after all). We could overload the dot operator, but in most C-based languages it controls access to data within an object, and so overloading it is either illegal or a dangerous thing to do.

The scalar product methods have the following form:

Excerpt from include/cyclone/core.h

```
class Vector3
{
    // ... Other Vector3 code as before ...

    /**
     * Calculates and returns the scalar product of this vector
     * with the given vector.
     */
    real scalarProduct(const Vector3 &vector) const
    {
        return x*vector.x + y*vector.y + z*vector.z;
    }

    /**
     * Calculates and returns the scalar product of this vector
     * with the given vector.
     */
```

```
    real operator *(const Vector3 &vector) const
    {
        return x*vector.x + y*vector.y + z*vector.z;
    }
};
```

Notice that there is no in-place version of the operator (because the result is a scalar value, and in most languages an instance of a class can't update itself to be an instance of a different class: the vector can't become a scalar). I have also added a full method version, `scalarProduct`, in case you are more comfortable writing things longhand rather than remembering the slightly odd behavior of the * operator.

The Trigonometry of the Scalar Product

There is an important result for scalar products, which is not obvious from equation 2.3. It relates the scalar product to the length of the two vectors and the angle between them:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_x b_x + a_y b_y + a_z b_z = |\boldsymbol{a}||\boldsymbol{b}| \cos\theta \qquad [2.4]$$

where θ is the angle between the two vectors.

So, if we have two normalized vectors, $\widehat{\boldsymbol{a}}$ and $\widehat{\boldsymbol{b}}$, then the angle between them is given by equation 2.4 as

$$\theta = \cos^{-1}\left(\widehat{\boldsymbol{a}} \cdot \widehat{\boldsymbol{b}}\right)$$

Note the normalized vectors here. If a and b are just regular vectors, then the angle would be given by

$$\theta = \cos^{-1}\left(\frac{\widehat{\boldsymbol{a}} \cdot \widehat{\boldsymbol{b}}}{|\boldsymbol{a}||\boldsymbol{b}|}\right)$$

If you so desire, you can easily convince yourself that equations 2.3 and 2.4 are equivalent by using Pythagoras's theorem and constructing a right-angled triangle where each vector is the hypotenuse. I'll leave this as an exercise for the skeptical.

The Geometry of the Scalar Product

The scalar product arises time and again in physics programming. In most cases it is used because it allows us to calculate the magnitude of one vector in the direction of another.

Figure 2.6 shows vectors in two dimensions (for simplicity—there is no difference in three dimensions). Notice that vector $\widehat{\boldsymbol{a}}$ has unit length. Vector $\boldsymbol{b}$ is almost at right angles to $\widehat{\boldsymbol{a}}$, most of its length points away, and only a small component is in the direction of $\widehat{\boldsymbol{a}}$. Its component is shown, and despite the fact that $\boldsymbol{b}$ is long, its component in the direction of $\widehat{\boldsymbol{a}}$ is small.
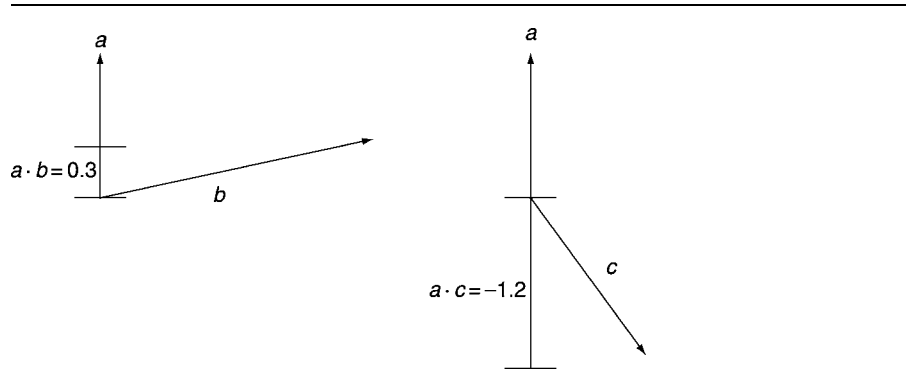
FIGURE 2.6 Geometric interpretation of the scalar product.

Vector c, however, is smaller in magnitude, but it is not pointing at right angles to $\widehat{a}$. Notice that it is pointing in almost the opposite direction to $\widehat{a}$. In this case its component in the direction of $\widehat{a}$ is negative.

We can see this in the scalar products:

$$|\widehat{a}| \equiv 1$$
$$|b| = 2.0$$
$$|c| = 1.5$$
$$\widehat{a} \cdot b = 0.3$$
$$\widehat{a} \cdot b = -1.2$$

If one vector is not normalized, then the size of the scalar product is multiplied by its length (from equation 2.4). In most cases, however, at least one vector, and often both, will be normalized before performing a scalar product.

When you see the scalar product in the physics engines in this book, it will most likely be as part of a calculation that needs to find how much one vector lies in the direction of another.

2.1.8 THE VECTOR PRODUCT

Whereas the scalar product multiplies two vectors together to give a scalar value, the vector product multiplies them to get another vector. In this way it is similar to the component product but is considerably more common.

The vector product is indicated by a multiplication sign, $a \times b$, and so is often called the "cross product." The vector product is calculated with the formula

$$a \times b = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix} \qquad [2.5]$$

This is implemented in our vector class in this way:

```
——— Excerpt from include/cyclone/core.h ———

class Vector3
{
    // ... Other Vector3 code as before ...

    /**
     * Calculates and returns the vector product of this vector
     * with the given vector.
     */
    Vector3 vectorProduct(const Vector3 &vector) const
    {
        return Vector3(y*vector.z-z*vector.y,
                       z*vector.x-x*vector.z,
                       x*vector.y-y*vector.x);
    }

    /**
     * Updates this vector to be the vector product of its current
     * value and the given vector.
     */
    void operator %=(const Vector3 &vector)
    {
        *this = vectorProduct(vector);
    }

    /**
     * Calculates and returns the vector product of this vector
     * with the given vector.
     */
    Vector3 operator%(const Vector3 &vector) const
    {
        return Vector3(y*vector.z-z*vector.y,
                       z*vector.x-x*vector.z,
                       x*vector.y-y*vector.x);
```

```
      }
};
```

To implement this product I have overloaded the `%` operator, simply because it looks most like a cross. This operator is usually reserved for modulo division in most languages, so purists may balk at reusing it for an unrelated mathematical operation. If you are easily offended, you can use the longhand `vectorProduct` method instead. Personally I find the convenience of being able to use operators outweighs any confusion, especially as vectors have no sensible modulo division operation.

The Trigonometry of the Vector Product

Just like the scalar product, the vector product has a trigonometric correspondence. This time the magnitude of the product is related to the magnitude of its arguments and the angle between them, as follows:

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin \theta \qquad [2.6]$$

where θ is the angle between the vectors, as before.

This is the same as the scalar product, replacing the cosine with the sine. In fact we can write

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sqrt{1 - (\boldsymbol{a} \cdot \boldsymbol{b})^2}$$

using the relationship between cosine and sine:

$$\cos^2 \theta + \sin^2 \theta = 1$$

We could use equation 2.6 to calculate the angle between two vectors, just as we did using equation 2.4 for the scalar product. This would be wasteful, however: it is much easier to calculate the scalar product than the vector product. So if we need to find the angle (which we rarely do), then using the scalar product would be a faster solution.

Commutativity of the Vector Product

You may have noticed in the derivation of the vector product that it is not commutative. In other words, $\boldsymbol{a} \times \boldsymbol{b} \neq \boldsymbol{b} \times \boldsymbol{a}$. This is different from each of the previous products of two vectors: both $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{b} \circ \boldsymbol{a}$ and $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{b} \cdot \boldsymbol{a}$.

In fact, by comparing the components in equation 2.5, we can see that

$$\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$$

This equivalence will make more sense once we look at the geometrical interpretation of the vector product.
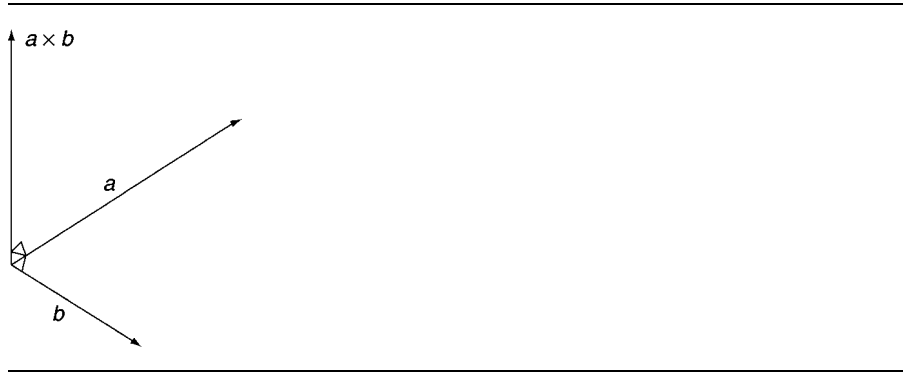
FIGURE 2.7 Geometric interpretation of the vector product.

In practice the non-commutative nature of the vector product means that we need to ensure that the orders of arguments are correct in equations. This is a common error and can manifest itself in the game by objects being sucked into each other with ever increasing velocity or bobbing in and out of surfaces at an ever faster rate.

The Geometry of the Vector Product

Once again using the scalar product as an example, we can interpret the magnitude of the vector product of a vector and a normalized vector. For a pair of vectors, $\hat{a}$ and b, the *magnitude* of the vector product represents the component of b that is *not* in the direction of $\hat{a}$. Again, having a vector a that is not normalized simply gives us a magnitude that is scaled by the length of a. This can be used in some circumstances, but in practice it is a relatively minor result.

Because it is easier to calculate the scalar product than the vector product, if we need to know the component of a vector not in the direction of another vector, we are better off performing the scalar product and then using Pythagoras's theorem to give the result

$$c = \sqrt{1 - s^2}$$

where c is the component of b not in the direction of $\hat{a}$ and s is the scalar product $\hat{a} \cdot b$.

In fact the vector product is very important geometrically, not for its magnitude but for its direction. In three dimensions the vector product will point in a direction that is at right angles (i.e., 90°, also called "orthogonal") to both of its operands. This is illustrated in figure 2.7. There are several occasions in this book where it will be convenient to generate a unit vector that is at right angles to other vectors. We can accomplish this easily using the vector product:

$$r = \widehat{a \times b}$$

Notice in particular that this is only defined in three dimensions. In two dimensions there is no possible vector that is at right angles to two non-parallel vectors. In higher dimensions (which I admit are not very useful for a games programmer), it is also not defined in the same way.

2.1.9 THE ORTHONORMAL BASIS

In some cases we want to construct a triple of mutually orthogonal vectors, where each vector is at right angles to the other two. Typically we want each of the three vectors to be normalized.

Fortunately there is a simple algorithm using the vector product that we can follow. This process starts with two non-parallel vectors. The first of these two will be invariant in direction; let's call this a: we cannot change its direction during the process, but if it is not normalized, we will change its magnitude. The other vector, b, may be changed if it is not already at right angles to the first. The third vector in the triple is not given at all.

1. Find vector c by performing the cross product $c = a \times b$.

2. If vector c has a zero magnitude, then give up: this means that a and b are parallel.

3. Now we need to make sure a and b are at right angles. We can do this by recalculating b based on a, and c using the cross product: $b = c \times a$ (note the order).

4. Normalize each of the vectors to give the output: $\widehat{a}$, $\widehat{b}$, and $\widehat{c}$.

This kind of triple of vectors is called an "orthonormal basis." When we talk about contact detection and contact resolution later in the book, we will need this algorithm several times.

Note that this algorithm is one in which it matters a great deal whether you are working in a left- or right-handed coordinate system. The algorithm is designed for right-handed systems. If you need a left-handed coordinate system, then you can simply change the order of the operands for both the cross products. This will give you a left-handed orthonormal basis.

2.2 CALCULUS

Calculus is a complex field with tendrils that reach into all areas of mathematics. Strictly speaking *calculus* means any kind of formal mathematical system, but when we talk about "the calculus," we normally mean analysis: the study of the behavior of functions. Real analysis is the most common topic of high school and undergraduate calculus classes: the study of functions that operate on real numbers. We are interested in vector analysis (usually widened to "matrix analysis," of which vectors are just one part). Even this subfield of a subfield is huge and contains many branches that have filled countless books on their own.

Fortunately for our purposes we are only interested in a very limited part of the whole picture. We are interested in the way something changes over time: it might

be the position of an object, or the force in a spring, or its rotational speed. The quantities we are tracking in this way are mostly vectors (we'll return to the non-vectors later in the book).

There are two ways of understanding these changes: we describe the change itself, and we describe the results of the change. If an object is changing position with time, we need to be able to understand how it is changing position (i.e., its speed, the direction it is moving in, if it is accelerating or slowing) and the effects of the change (i.e., where it will be when we come to render it during the next frame of the game). These two viewpoints are represented by differential and integral calculus, respectively. We can look at each in turn.

No code will be provided for this section; it is intended as a review of the concepts involved only. The corresponding code makes up most of the rest of this book, very little of which will make sense unless you grasp the general idea of this section.

2.2.1   Differential Calculus

For our purposes we can view the differential of a quantity as the rate at which it is changing. In the majority of this book we are interested in the rate at which it is changing with respect to time. This is sometimes informally called its "speed."

When we come to look at vector calculus, however, speed has a different meaning. It is more common in mathematics and almost ubiquitous in physics programming to call it "velocity."

Velocity

Think about the position of an object, for example. If this represents a moving object, then in the next instance of time, the position of the object will be slightly different. We can work out the velocity at which the object is moving by looking at the two positions. We could simply wait for a short time to pass, find the position of the object again, and use the formula

$$v = \frac{p' - p}{\Delta t} = \frac{\Delta p}{\Delta t}$$

where v is the velocity of the object, p' and p are its positions at the first and second measurements (so Δp is the change in position), and Δt is the time that has passed between the two. This would give us the average velocity over the time period. It wouldn't tell us the exact velocity of the object at any point in time, however.

Figure 2.8 shows the position of two objects at different times. Both objects start at the same place and end at the same place at the same time. Object A travels at a constant velocity, whereas object B stays near its start location for a while and then zooms across the gap very quickly. Clearly they aren't traveling at the same velocity.

If we want to calculate the *exact* velocity of an object, we could reduce the gap between the first and second measurement. As the gap gets smaller, we get a more
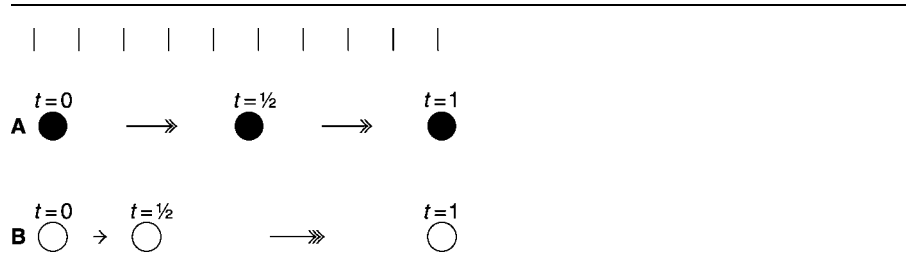
FIGURE 2.8   Same average velocity, different instantaneous velocity.

accurate picture of the velocity of the object at one instant in time. If we could make this gap infinitely small, then we would have the exact answer.

In mathematical notation this is written using the "limit" notation

$$v = \lim_{\Delta t \to 0} \frac{\Delta p}{\Delta t}$$

which simply means that the velocity would be accurately given by the distance traveled divided by the time gap ($\Delta p / \Delta t$), if we could make the gap infinitely small ($\lim_{t \to 0}$). Rather than use the cumbersome limit notation, this is more commonly written with a lowercase d in place of the Δ:

$$v = \lim_{\Delta t \to 0} \frac{\Delta p}{\Delta t} = \frac{\mathrm{d}p}{\mathrm{d}t}$$

Because it is so common in mechanics to be talking about the change with respect to time, this is often simplified even further:

$$v = \frac{\mathrm{d}p}{\mathrm{d}t} = \dot{p}$$

The dot over the p simply means that it is the velocity at which p is changing—its differential with respect to time.

Acceleration

If p is the position of an object and v is its velocity (where $v = \dot{p}$), we can define its acceleration too.

Acceleration is the rate at which velocity is changing. If an object is accelerating quickly, it is rapidly increasing in velocity. In normal English we use the word *slowing* to mean the opposite of acceleration, or *braking* if we are talking about a car. In physics acceleration it can mean any change in velocity, either increasing or decreasing velocity. A positive value for acceleration represents speeding up, a zero

acceleration means no change in velocity at all, and a negative acceleration represents slowing down.

Because acceleration represents the rate at which velocity is changing, following the same process we arrive at

$$a = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{\mathrm{d}v}{\mathrm{d}t}$$

where v in this formula is a velocity itself, also defined in terms of its own limit, as we saw earlier in this section. We could write this as $a = \dot{v}$ if we wanted to, but this causes problems.

As long as I use v for velocity, it is fairly clear, but in general if I write

$$\frac{\mathrm{d}m}{\mathrm{d}t}$$

it would not be obvious whether m is a velocity (and therefore the whole expression is an acceleration) or if it is a position (making the expression a velocity). To avoid this confusion, it is typical to write accelerations in terms of the position only.

This is called the "second differential." Velocity is the first differential of position, and if we differentiate again, we get acceleration; so acceleration is the second differential. Mathematicians often write it in this way:

$$a = \frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}p}{\mathrm{d}t} = \frac{\mathrm{d}^2 p}{\mathrm{d}t^2}$$

which can be confusing if you're not used to differential notation. Don't worry about how we end up with the strange set of squared things—it isn't important for us; it simply indicates a second differential. Fortunately we can completely ignore this format altogether and use the dotted form again—

$$a = \frac{\mathrm{d}^2 p}{\mathrm{d}t^2} = \ddot{p}$$

which is the format we'll use in the remainder of the book.

We could carry on and find the rate at which the acceleration is changing (this is called the "jerk" or sometimes the "jolt," and it is particularly important in the design of roller coasters, among other things). We could go further and find the rate of change of jerk, and so on. It turns out, however, that these quantities are not needed to get a believable physics engine running. As we shall see in the next chapter, Newton discovered that applying a force to an object changes its acceleration: to make believable physics involving forces, all we need to keep track of are the position, velocity, and acceleration.

So, in summary: $\dot{p}$, the velocity of p, is measured at one instant of time, not at an average velocity; and $\ddot{p}$ is the acceleration of p, measured in exactly the same way, and it can be negative to indicate slowing down.

Vector Differential Calculus

So far we've looked at differentiation purely in terms of a single scalar quantity. For full three-dimensional physics we need to deal with vector positions rather than scalars.

Fortunately the simple calculus we've looked at so far works easily in three dimensions (although you must be careful: as a general rule, most of the equations for one-dimensional calculus you find in mathematics reference books cannot be used in three dimensions).

If the position of the object is given as a vector in three dimensions, then its rate of change is also represented by a vector. Because a change in the position on one axis doesn't change the position on any other axes, we can treat each axis as if it were its own scalar differential.

The velocity and the acceleration of a vector depends only on the velocity and acceleration of its components:

$$\dot{a} = \begin{bmatrix} \dot{a}_x \\ \dot{a}_y \\ \dot{a}_z \end{bmatrix}$$

and similarly

$$\ddot{a} = \begin{bmatrix} \ddot{a}_x \\ \ddot{a}_y \\ \ddot{a}_z \end{bmatrix}$$

As long as the formulae we meet do not involve the products of vectors, this matches exactly with the way we defined vector addition and vector–scalar multiplication earlier in the chapter. The upshot of this is that for most formulae involving the differentials of vectors, we can treat the vectors as if they were scalars. We'll see an example of this in the next section.

As soon as we have formulae that involve multiplying vectors together, or that involve matrices, however, things are no longer as simple. Fortunately they are rare in this book.

Velocity, Direction, and Speed

Although in regular English we often use *speed* and *velocity* as synonyms, they have different meanings for physics development. The velocity of an object, as we've seen, is a vector giving the rate at which its position is changing.

The speed of an object is the magnitude of this velocity vector, and the direction that the object is moving in is given by the normalized velocity. So

$$\dot{x} = s\,\widehat{d}$$

where *s* is the speed of the object, and $\widehat{\boldsymbol{d}}$ is its direction of movement. Using the equations for the magnitude and direction of any vector, the speed is given by

$$s = |\dot{\boldsymbol{x}}|$$

and the direction by

$$\widehat{\boldsymbol{d}} = \frac{\dot{\boldsymbol{x}}}{|\dot{\boldsymbol{x}}|}$$

Both the speed and the direction can be calculated from a velocity vector using the `magnitude` and `normalize` methods we developed earlier in the chapter; they do not need additional code.

The speed of an object is rarely needed in physics development: it has an application in calculating aerodynamic drag, but little else. Both the speed and the direction of movement are often used by an animation engine to work out what animation to play as a character moves. This is less common for physically controlled characters.

Mostly this is a terminology issue: it is worth getting into the habit of calling velocity *velocity*, because *speed* has a different meaning.

2.2.2   INTEGRAL CALCULUS

In mathematics, integration is the opposite of differentiation. If we differentiate something, and then integrate it, we get back to where we started.

In the same way that we got the velocity from the position using differentiation, we go the other way in integration. If we know the velocity, then we can work out the position at some point in the future. If we know the acceleration, we can find the velocity at any point in time.

In physics engines, integration is used to update the position and velocity of each object. The chunk of code that performs this operation is called the "integrator."

Although integration in mathematics is an even more complex process than differentiation, involving considerable algebraic manipulation, in game development it is very simple. If we know that an object is moving with a constant velocity (i.e., no acceleration), and we know this velocity along with how much time has passed, we can update the position of the object using the formula

$$p' = p + \dot{p}t \qquad\qquad [2.7]$$

where $\dot{p}$ is the constant velocity of the object over the whole time interval.

This is the integral of the velocity: an equation that gives us the position. In the same way we could update the object's velocity in terms of its acceleration using the formula

$$\dot{p}' = \dot{p} + \ddot{p}t \qquad\qquad [2.8]$$

Equation 2.7 only works for an object that is not accelerating. Rather than finding the position by the first integral of the velocity, we could find it as the second integral

of the acceleration (just as acceleration was the second derivative of the position). This would give us an updated equation of

$$p' = p + \dot{p}t + \ddot{p}\frac{t^2}{2} \tag{2.9}$$

where $\dot{p}$ is the velocity of the object at the start of the time interval, and $\ddot{p}$ is the constant acceleration over the whole time.

It is beyond the scope of this book to describe how these equations are arrived at: you can see any introduction to calculus for the basic algebraic rules for finding equations for differentials and integrals. For our purposes in this book I will provide the equations: they match those found in applied mathematics books for mechanics.

Just as equation 2.7 assumes a constant velocity, equation 2.9 assumes a constant acceleration. We could generate further equations to cope with changing accelerations. As we will see in the next chapter, however, even 2.9 isn't needed when it comes to updating the position; we can make do with the assumption that there is no acceleration.

In mathematics, when we talk about integrating, we mean to convert a formula for velocity into a formula for position; or a formula for acceleration into one for velocity—in other words, to do the opposite of a differentiation. In game development the term is often used slightly differently: *to integrate* means to perform the position or velocity updates. From this point on I will stick to the second use, since we will have no need to do mathematical integration again.

Vector Integral Calculus

Just as we saw for differentiation, vectors can often take the place of scalars in the update functions. Again this is not the case for mathematics in general, and most of the formulae you find in mathematical textbooks on integration will not work for vectors. In the case of the two integrals we will use in this book—equations 2.7 and 2.8—it just so happens that it does. So we can write

$$\boldsymbol{p}' = \boldsymbol{p} + \dot{\boldsymbol{p}}t$$

and perform the calculation on a component-by-component basis:

$$\boldsymbol{p}' = \boldsymbol{p} + \dot{\boldsymbol{p}}t = \begin{bmatrix} p_x + \dot{p}_x t \\ p_y + \dot{p}_y t \\ p_z + \dot{p}_z t \end{bmatrix}$$

This could be converted into code as

```
position += velocity * t;
```

using the overloaded operator forms of + and * we defined earlier. In fact this is exactly the purpose of our addScaledVector method, so we can write

```
position.addScaledVector(velocity, t);
```

and have it done in one operation, rather than taking the risk that our compiler will decide to create and pass around extra vectors on the stack.

We now have almost all the mathematics we need for the particle engine implementation. We will implement the integration step in the next chapter, after we look at the physics involved in simulating particles.

2.3 SUMMARY

Vectors form the basis of all the mathematics in this book. As we've seen, they are easy to manipulate numerically and through simple routines in code. It is important to remember, however, that vectors are geometrical: they represent positions and movements in space. It is very often much simpler to understand the formulae in this book in terms of their corresponding geometric properties rather than look at them numerically.

Describing positions and movements in terms of vectors is fine, but to make a physics engine we'll need to begin to link the two. That is, we'll have to encode into our physics engine the laws of physics that say how position and movement and time are connected. This is the subject of chapter 3.

3
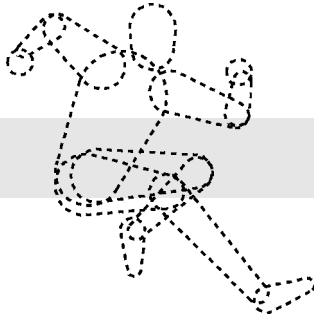
THE LAWS
OF MOTION

Physics engines are based on Newton's laws of motion. In later sections we will begin to use results that were added to Newton's original work, but the fundamentals are his.

Newton created three laws of motion that describe with great accuracy how a point mass behaves. A *point mass* is what we call a "particle," but shouldn't be confused with particle physics, which studies tiny particles such as electrons or photons that definitely do not conform to Newton's laws. For this book we'll use *particle* rather than *point mass*.

Beyond particles we need the physics of rotating, which introduces additional complications that were added to Newton's laws. Even in these cases, however, the point-mass laws still can be seen at work.

Before we look at the laws themselves, and how they are implemented, we need to look at what a particle is within our engine and how it is built in code.

3.1 A Particle

A particle has a position, but no orientation. In other words, we can't tell in what direction a particle is pointing: it either doesn't matter or doesn't make sense. In the former category are bullets: in a game we don't really care in what direction a bullet is pointing; we just care in what direction it is traveling and whether it hits the target. In the second category are pinpricks of light, from an explosion, for example: the light is a dot of light, and it doesn't make sense to ask in which direction a spot of light is pointing.

For each particle we'll need to keep track of various properties: we'll need its current position, its velocity, and its acceleration. We will add additional properties to the particle as we go. The position, velocity, and acceleration are all vectors.

The particle can be implemented with the following structure:

```
                ──── Excerpt from include/cyclone/particle.h ────
/**
 * A particle is the simplest object that can be simulated in the
 * physics system.
 */
class Particle
{
public:
    /**
     * Holds the linear position of the particle in
     * world space.
     */
    Vector3 position;

    /**
     * Holds the linear velocity of the particle in
     * world space.
     */
    Vector3 velocity;

    /**
     * Holds the acceleration of the particle. This value
     * can be used to set acceleration due to gravity (its primary
     * use) or any other constant acceleration.
     */
    Vector3 acceleration;

};
```

Using this structure we can apply some basic physics to create our first physics engine.

3.2 THE FIRST TWO LAWS

There are three law of motion posited by Newton, but for now we will need only the first two. They deal with the way an object behaves in the presence and absence of forces. The first two laws of motion are:

1. An object continues with a constant velocity unless a force acts upon it.

2. A force acting on an object produces acceleration that is proportional to the object's mass.

3.2.1 THE FIRST LAW

The first law (Newton 1) tells us what happens if there are no forces around. The object will continue to move with a constant velocity. In other words, the velocity of the particle will never change, and its position will keep on being updated based on the velocity. This may not be intuitive: moving objects we see in the real world will slow and come to a stop eventually if they aren't being constantly forced along. In this case the object is experiencing a force—the force of drag (or friction if it is sliding along). In the real world we can't get away from forces acting on a body; the nearest we can imagine is the movement of objects in space. What Newton 1 tells us is that if we could remove all forces, then objects would continue to move at the same speed forever.

In our physics engine we could simply assume that there are no forces at work and use Newton 1 directly. To simulate drag we could add special drag forces. This is fine for the simple engine we are building in this part of the book, but it can cause problems with more complex systems. The problem arises because the processor that performs the physics calculations isn't completely accurate. This inaccuracy can lead to objects getting faster of their own accord.

A better solution is to incorporate a rough approximation of drag directly into the engine. This allows us to make sure objects aren't being accelerated by numerical inaccuracy, and it can allow us to simulate some kinds of drag. If we need complicated drag (such as aerodynamic drag in a flight simulator or racing game), we can still do it the long way by creating a special drag force. We call the simple form of drag "damping" to avoid confusion.

To support damping, we add another property to the particle class:

```
──────── Excerpt from include/cyclone/particle.h ────────
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Holds the amount of damping applied to linear
     * motion. Damping is required to remove energy added
     * through numerical instability in the integrator.
     */
    real damping;
};
```

When we come to perform the integration, we will remove a proportion of the object's velocity at each update. The damping parameter controls how velocity is left

after the update. If the damping is zero, then the velocity will be reduced to nothing: this would mean that the object couldn't sustain any motion without a force and would look odd to the player. A value of 1 means that the object keeps all its velocity (equivalent to no damping). If you don't want the object to look like it is experiencing drag, then values near but less than 1 are optimal—0.995, for example.

3.2.2 THE SECOND LAW

The second law (Newton 2) gives us the mechanism by which forces alter the motion of an object. A force is something that changes the *acceleration* of an object (i.e., the rate of change of velocity). An implication of this law is that we cannot do anything to an object to directly change its position or velocity; we can only do that indirectly by applying a force to change the acceleration and wait until the object reaches our target position or velocity. We'll see later in the book that physics engines need to abuse this law to look good, but for now we'll keep it intact.

Because of Newton 2, we will treat the acceleration of the particle different from velocity and position. Both velocity and position keep track of a quantity from frame to frame during the game. They change, but not directly, only by the influence of accelerations. Acceleration, by contrast, can be different from one moment to another. We can simply set the acceleration of an object as we see fit (although we'll use the force equations in Section 3.2.3), and the behavior of the object will look fine. If we directly set the velocity or position, the particle will appear to jolt or jump. Because of this the position and velocity properties will only be altered by the integrator and should not be manually altered (other than setting up the initial position and velocity for an object, of course). The acceleration property can be set at any time, and it will be left alone by the integrator.

3.2.3 THE FORCE EQUATIONS

The second part of Newton 2 tells us how force is related to the acceleration. For the same force, two objects will experience different accelerations depending on their mass. The formula relating the force to the acceleration is the famous

$$f = ma = m\ddot{p} \qquad\qquad [3.1]$$

Where we are trying to find the acceleration, we have

$$\ddot{p} = \frac{1}{m}f \qquad\qquad [3.2]$$

where f is the force and m is the mass.

In a physics engine we typically find the forces applying to each object and then use equation 3.2 to find the acceleration, which can then be applied to the object by the integrator. For the engine we are creating in this part of the book, we can find

the acceleration in advance using this equation, without having to repeat it at every update. In the remainder of the book, however, it will be a crucial step.

So far all the equations have been given in their mathematics textbook form, applied to scalar values. As we saw in the last chapter on calculus, we can convert them easily to use vectors:

$$\ddot{p} = \frac{1}{m} f$$

so the force is a vector, as well as acceleration.

3.2.4 Adding Mass to Particles

Added to the position and velocity we have stored per particle, we need to store its mass so that we can correctly calculate its response to forces. Many physics engines simply add a scalar mass value for each object. There is a better way to get the same effect, however.

First there is an important thing to notice about equation 3.2. If the mass of an object is zero, then the acceleration will be infinite, as long as the force is not zero. This is not a situation that should ever occur: no particle that we can simulate should ever have zero mass. If we try to simulate a zero mass particle, it will cause divide-by-zero errors in the code.

It is often useful, however, to simulate infinite masses. These are objects that no force of any magnitude can move. They are very useful for immovable objects in a game: the walls or floor, for example, cannot be moved during the game. If we feed an infinite mass into equation 3.2, then the acceleration will be zero, as we expect. Unfortunately we cannot represent a true infinity in most computer languages, and the optimized mathematics instructions on all common game processors do not cope well with infinities. We have to get slightly creative. Ideally we want a solution where it is easy to get infinite masses but impossible to get zero masses.

Notice in equation 3.2 that we use 1 over the mass each time. Because we never use the 3.1 form of the equation, we can speed up our calculations by not storing the mass for each object, but 1 over the mass. We call this the "inverse mass." This solves our problem for representing objects of zero or infinite mass: infinite mass objects have a zero inverse mass, which is easy to set. Objects of zero mass would have an infinite inverse mass, which cannot be specified in most programming languages.

We update our particle class to include the inverse mass in this way:

```
          ────────── Excerpt from include/cyclone/particle.h ──────────
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Holds the inverse of the mass of the particle. It
     * is more useful to hold the inverse mass because
```

```
      * integration is simpler and because in real-time
      * simulation it is more useful to have objects with
      * infinite mass (immovable) than zero mass
      * (completely unstable in numerical simulation).
      */
    real inverseMass;
};
```

It is really important to remember that you are dealing with the inverse mass, and not the mass. It is quite easy to set the mass of the particle directly, without remembering, only to see it perform completely inappropriate behavior on screen—barely movable barrels zooming off at the slightest tap, for example.

To help with this, I've made the `inverseMass` data field protected in the `Particle` class on the CD. To set the inverse mass you need to use an accessor function. I have provided functions for `setInverseMass` and `setMass`. Most of the time it is more convenient to use the latter, unless we are trying to set an infinite mass.

3.2.5 Momentum and Velocity

Although Newton 1 is often introduced in terms of velocity, that is a misrepresentation. It is not velocity that is constant in the absence of any forces, but momentum.

Momentum is the product of velocity and mass. Since mass is normally constant, we can assume that velocity is therefore constant by Newton 1. In the event that a traveling object were changing mass, then its velocity would also be changing, even with no forces.

We don't need to worry about this for our physics engine because we'll not deal with any situation where mass changes. It will be an important distinction when we come to look at rotations later, however, because rotating objects can change the way their mass is distributed. Under the rotational form of Newton 1, that means a change in rotational speed, with no other forces acting.

3.2.6 The Force of Gravity

The force of gravity is the most important force in a physics engine. Gravity applies between every pair of objects: attracting them together with a force depends on their mass and their distance apart. It was Newton who also discovered this fact, and along with the three laws of motion he used it to explain the motion of planets and moons with a new level of accuracy.

The formula he gave us is the "law of universal gravitation":

$$f = G\frac{m_1 m_2}{r^2} \tag{3.3}$$

where m_1 and m_2 are the masses of the two objects, r is the distance between their centers, f is the resulting force, and G is the "universal gravitational constant"—a scaling factor derived from observation of planetary motion.

The effects of gravity between two objects the size of a planet are significant; the effects between (relatively) small objects such as a car, or even a building, are small. On earth our experience of gravity is completely dominated by the earth itself. We notice the pull of the moon in the way our tides work, but other than that we only experience gravity pulling us down to the planet's surface.

Because we are only interested in the pull of the earth, we can simplify equation 3.3. First we can assume that m_1 is always constant. Second, and less obviously, we can assume that r is also constant. This is due to the huge distances involved. The distance from the surface of the earth to its center is so huge (6,400 km) that there is almost no difference in gravity between standing at sea level and standing on the top of a mountain. For the accuracy we need in a game, we can therefore assume the r parameter is constant.

Equation 3.3 simplifies to

$$f = mg \qquad\qquad [3.4]$$

where m is the mass of the object we are simulating; f is the force, as before; and g is a constant that includes the universal gravitational constant, the mass of the earth, and its radius:

$$g = G\frac{m_{\text{earth}}}{r^2}$$

The constant, g, is an acceleration, which we measure in meters per second per second. On earth this g constant has a value of around 10 m/s^2. (Scientists sometimes use a value of 9.807 m/s^2, although because of the variations in r and other effects, this is a global average rather than a measured value.)

Notice that the force depends on the mass of the object. If we work out the acceleration using equation 3.2, then we get

$$\ddot{p} = \frac{1}{m}mg = g$$

In other words, no matter what mass the object has, it will always accelerate at the same rate due to gravity. As the legend goes, Galileo dropped heavy and light objects from the Tower of Pisa and showed that they hit the ground at the same time.

What this means for our engine is that the most significant force we need to apply can be applied directly as an acceleration. There is no point using equation 3.4 to calculate a force and then using equation 3.2 to convert it back into an acceleration. In this iteration of the engine we will introduce gravity as the sole force at work on particles, and it will be applied directly as an acceleration.

The Value of g

It is worth noting that, although the acceleration due to gravity on earth is about 10 m/s^2, this doesn't look very convincing on screen. Games are intended to be more exciting than the real world: things happen more quickly and at a higher intensity.

Creating simulations with a g value of 10 m/s^2 can look dull and insipid. Most developers use higher values, from around 15 m/s^2 for shooters (to avoid projectiles being accelerated into the ground too quickly) to 20 m/s^2, which is typical of driving games. Some developers go further and incorporate the facility to tune the g value on an object-by-object basis. Our engine will include this facility.

Gravity typically acts in the down direction, unless you are going for a special effect. In most games the Y axis represents up and down in the game world; and almost exclusively the positive Y axis points up.

The acceleration due to gravity can therefore be represented as a vector with the form

$$\boldsymbol{g} = \begin{bmatrix} 0 \\ -g \\ 0 \end{bmatrix}$$

where g is the value we discussed before, and $\boldsymbol{g}$ is the acceleration vector we will use to update the particle in the next section.

3.3 THE INTEGRATOR

We now have all the equations and background needed to finish the first implementation of the engine. At each frame, the engine needs to look at each object in turn, work out its acceleration, and perform the integration step. The calculation of the acceleration in this case will be trivial: we will use the acceleration due to gravity only.

The integrator consists of two parts: one to update the position of the object, and the other to update its velocity. The position will depend on the velocity and acceleration, while the velocity will depend only on the acceleration.

Integration requires a time interval over which to update the position and velocity: because we update every frame, we use the time interval between frames as the update time. If your engine is running on a console that has a consistent frame rate, then you can hard-code this value into your code (although it is wise not to because in the same console different territories can have different frame rates). If you are running on a PC with a variable frame-rate, then you probably need to time the duration of the frame.

Typically developers will time a frame and then use that value to update the next frame. This can cause noticeable jolts if frame durations are dramatically inconsistent, but the game is unlikely to feel smooth in this case anyway, so it is a common rule of thumb.

In the code on the CD I use a central timing system that calculates the duration of each frame. The physics code we will develop here simply takes in a time parameter when it updates and doesn't care how this value was calculated.

3.3.1 The Update Equations

We need to update both position and velocity; each is handled slightly differently.

Position Update

In chapter 2 we saw that integrating the acceleration twice gives us this equation for the position update:

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}t^2$$

This is a well-known equation seen in high school and undergraduate textbooks on applied mathematics.

We could use this equation to perform the position update in the engine, with code something like

```
object.position += object.velocity * time +
                   object.acceleration * time * time * 0.5;
```

or

```
object.position.addScaledVector(object.velocity, time);
object.position.addScaledVector(object.acceleration, time * time * 0.5);
```

In fact, if we are running the update every frame, then the time interval will be very small (typically 0.033 s for a 30-frames-per-second game). If we look at the acceleration part of this equation, we are taking half of the squared time (which gives 0.0005). This is such a small value that it is unlikely the acceleration will have much of an impact on the change in position of an object.

For this reason we typically ignore the acceleration entirely in the position update and use the simpler form

$$p' = p + \dot{p}t$$

This is the equation we will use in the integrator throughout this book.

If your game regularly uses short bursts of huge accelerations, then you might be better off using the longer form of the equation. If you do intend to use huge accelerations, however, you are likely to get all sorts of other accuracy problems in any case: all physics engines typically become unstable with very large accelerations.

Velocity Update

The velocity update has a similar basic form

$$\dot{p}' = \dot{p} + \ddot{p}t$$

Earlier in the chapter, however, we introduced another factor to alter the velocity: the damping parameter.

The damping parameter is used to remove a bit of velocity at each frame. This is done by simply multiplying the velocity by the damping factor

$$\dot{p}' = \dot{p}d + \ddot{p}t \qquad [3.5]$$

where d is the damping for the object.

This form of the equation hides a problem, however. No matter whether we have a long or a short time interval over which to update, the amount of velocity being removed is the same. If our frame rate suddenly improves, then there will be more updates per second and the object will suddenly appear to have more drag. A more correct version of the equation solves this problem by incorporating the time into the drag part of the equation:

$$\dot{p}' = \dot{p}d^t + \ddot{p}t \qquad [3.6]$$

where the damping parameter d is now the proportion of the velocity retained each second, rather than each frame.

Calculating one floating-point number to the power of another is a slow process on most modern hardware. If you are simulating a huge number of objects, then it is normally best to avoid this step. For a particle physics engine designed to simulate thousands of sparks, for example, use equation 3.5, or even remove damping altogether.

Because we are heading toward an engine designed for simulating a smaller number of rigid bodies, I will use the full form in this book, as given in equation 3.6. A different approach favored by many engine developers is to use equation 3.5 with a damping value very near to 1—so small that it will not be noticeable to the player but big enough to be able to solve the numerical instability problem. In this case a variation in frame rate will not make any visual difference. Drag forces can then be created and applied as explicit forces that will act on each object (as we'll see in chapter 5).

Unfortunately, this simply moves the problem to another part of the code—the part where we calculate the size of the drag force. For this reason I prefer to make the damping parameter more flexible and allow it to be used to simulate visible levels of drag.

3.3.2   The Complete Integrator

We can now implement our integrator unit. The code looks like this:

```
———— Excerpt from include/cyclone/precision.h ————
/** Defines the precision of the power operator. */
#define real_pow powf
```

```
————— Excerpt from include/cyclone/particle.h —————
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Integrates the particle forward in time by the given amount.
     * This function uses a Newton-Euler integration method, which is a
     * linear approximation of the correct integral. For this reason it
     * may be inaccurate in some cases.
     */
    void integrate(real duration);
};
```

```
————— Excerpt from src/particle.cpp —————
#include <assert.h>
#include <cyclone/particle.h>

using namespace cyclone;

void Particle::integrate(real duration)
{
    assert(duration > 0.0);

    // Update linear position.
    position.addScaledVector(velocity, duration);

    // Work out the acceleration from the force.
    Vector3 resultingAcc = acceleration;
    resultingAcc.addScaledVector(forceAccum, inverseMass);

    // Update linear velocity from the acceleration.
    velocity.addScaledVector(resultingAcc, duration);

    // Impose drag.
    velocity *= real_pow(damping, duration);
}
```

I have added the integration method to the Particle class because it simply updates the particles' internal data. It takes just a time interval and updates the position and velocity of the particle, returning no data.

3.4 Summary

In two short chapters we've gone from coding vectors to a first complete physics engine.

The laws of motion are elegant, simple, and incredibly powerful. The fundamental connections that Newton discovered drive all the physical simulations in this book. Calculating forces, and integrating position and velocity based on force and integrating position and velocity based on force and time, are the fundamental steps of all physics engines, complex or simple.

Although we now have a physics engine that can actually be used in games (and is equivalent to the systems used in many hundreds of published games), it isn't yet suitable for a wide range of physical applications. In chapter 4 we'll look at some of the applications it can support and some of its limitations.

4

THE PARTICLE
PHYSICS ENGINE

We now have our first working physics engine capable of simulating the movement of particles under gravity.

Considering that it is such a simple piece of code, I've spent a long time talking about the theory behind it. This will become important later in this book when we repeat the same kind of logic for the rotation of objects.

At the moment our engine is fairly limited. It can only deal with isolated particles, and they cannot interact in any way with their environment. Although these are serious limitations that will be addressed in the next part of the book, we can still do some useful things with what we have.

In this chapter we will look at how to set up the engine to process ballistics: bullets, shells, and the like. We will also use the engine to create a fireworks display. Both of these applications are presented in skeleton form here, with no rendering code. They can be found with full source code on the CD.

4.1 BALLISTICS

One of the most common applications of physics simulation in games is to model ballistics. This has been the case for two decades, predating the current vogue for physics engines.

In our ballistics simulation, each weapon fires a particle. Particles can represent anything from bullets to artillery shells, from fireballs to laser-bolts. Regardless of the object being fired, we will call this a "projectile."

Each weapon has a characteristic muzzle velocity: the speed at which the projectile is emitted from the weapon. This will be very fast for a laser-bolt and probably considerably slower for a fireball. For each weapon the muzzle velocity used in the game is unlikely to be the same as its real-world equivalent.

4.1.1 Setting Projectile Properties

The muzzle velocity for the slowest real-world guns is in the order of 250 m/s, whereas tank rounds designed to penetrate armor plate by their sheer speed can move at 1,800 m/s. The muzzle velocity of an energy weapon, such as a laser, would be the speed of light: 300,000,000 m/s. Even for relatively large game levels, any of these values is far too high. A level that represents a square kilometer is huge by the standard of modern games: clearly a bullet that can cross this in half a second would be practically invisible to the player. If this is the aim, then it is better not to use a physics simulation, but to simply cast a ray through the level the instant that the weapon is shot and check to see whether it collides with the target.

Instead, if we want the projectile's motion to be visible, we use muzzle velocities that are in the region of 5 to 25 m/s for a human-scale game. (If your game represents half a continent, and each unit is the size of a city, then it would be correspondingly larger.) This causes two consequences that we have to cope with.

First, the mass of the particle should be larger than in real life, especially if you are working with the full physics engine later in this book and you want impacts to look impressive (being able to shoot a crate and topple it over, for example). The effect that a projectile has when it impacts depends on both its mass and its velocity: if we drop the velocity, we should increase the mass to compensate. The equation that links energy, mass, and speed is

$$e = ms^2$$

where e is the energy and s is the *speed* of the projectile (this equation doesn't work with vectors, so we can't use velocity). If we want to keep the same energy, we can work out the change in mass for a known change in speed:

$$\Delta m = (\Delta s)^2$$

Real-world ammunition ranges from a gram in mass up to a few kilograms for heavy shells and even more for other tactical weapons (bunker-busting shells used during the second Gulf War were more than 1000 kg in weight). A typical 5 g bullet that normally travels at 500 m/s might be slowed to 25 m/s. This is a Δs of 20. To get the same energy we need to give it 400 times the weight: 2 kg.

Second, we have to decrease the gravity on projectiles. Most projectiles shouldn't slow too much in flight, so the damping parameter should be near 1. Shells and mortars may arch under gravity, but other types of projectiles should barely feel the effect. If they were traveling at very high speed, then they wouldn't have time to be pulled down by gravity to a great extent, but since we've slowed them down, gravity will have
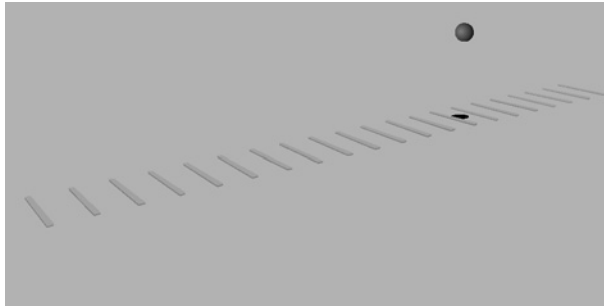
FIGURE 4.1   Screenshot of the **ballistic** demo.

longer to do its work. Likewise, if we are using a higher gravity coefficient in the game, it will make the ballistic trajectory far too severe: well-aimed projectiles will hit the ground only a few meters in front of the character. To avoid this we lower the gravity. For a known change in speed we can work out a "realistic" gravity value using the formula:

$$g_{bullet} = \frac{1}{\Delta s} g_{normal}$$

where g_{normal} is the gravity that you want to simulate. This would be 10 m/s^2 for most games (which is earth gravity, not the general gravity being used elsewhere in the simulation, which is typically higher). For our bullet example we have a g_{bullet} of 0.5 m/s^2.

4.1.2 IMPLEMENTATION

The **ballistic** demo on the CD (shown in figure 4.1) gives you the choice of four weapons: a pistol, an artillery piece, a fireball, and a laser gun (indicated by name at the bottom of the screen). When you click the mouse, a new round is fired. The code that creates a new round and fires it looks like this:

Excerpt from src/demos/ballistic/ballistic.cpp

```cpp
// Set the properties of the particle.
switch(currentShotType)
{
case PISTOL:
    shot->particle.setMass(2.0f); // 2.0kg
    shot->particle.setVelocity(0.0f, 0.0f, 35.0f); // 35m/s
    shot->particle.setAcceleration(0.0f, -1.0f, 0.0f);
    shot->particle.setDamping(0.99f);
    break;
```

```
case ARTILLERY:
    shot->particle.setMass(200.0f); // 200.0kg
    shot->particle.setVelocity(0.0f, 30.0f, 40.0f); // 50m/s
    shot->particle.setAcceleration(0.0f, -20.0f, 0.0f);
    shot->particle.setDamping(0.99f);
    break;

case FIREBALL:
    shot->particle.setMass(1.0f); // 1.0kg - mostly blast damage
    shot->particle.setVelocity(0.0f, 0.0f, 10.0f); // 5m/s
    shot->particle.setAcceleration(0.0f, 0.6f, 0.0f); // Floats up
    shot->particle.setDamping(0.9f);
    break;

case LASER:
    // Note that this is the kind of laser bolt seen in films,
    // not a realistic laser beam!
    shot->particle.setMass(0.1f); // 0.1kg - almost no weight
    shot->particle.setVelocity(0.0f, 0.0f, 100.0f); // 100m/s
    shot->particle.setAcceleration(0.0f, 0.0f, 0.0f); // No gravity
    shot->particle.setDamping(0.99f);
    break;
}

// Set the data common to all particle types.
shot->particle.setPosition(0.0f, 1.5f, 0.0f);
shot->startTime = TimingData::get().lastFrameTimestamp;
shot->type = currentShotType;

// Clear the force accumulators.
shot->particle.clearAccumulator();
```

Notice that each weapon configures the particle with a different set of values. The surrounding code is skipped here for brevity: you can refer to the source code on the CD to see how and where variables and data types are defined.

The physics update code looks like this:

─────── **Excerpt from src/demos/ballistic/ballistic.cpp** ───────

```
// Update the physics of each particle in turn.
for (AmmoRound *shot = ammo; shot < ammo+ammoRounds; shot++)
{
    if (shot->type != UNUSED)
    {
        // Run the physics.
```

```
            shot->particle.integrate(duration);

            // Check if the particle is now invalid.
            if (shot->particle.getPosition().y < 0.0f ||
                shot->startTime+5000 < TimingData::get().lastFrameTimestamp ||
                shot->particle.getPosition().z > 200.0f)
            {
                // We simply set the shot type to be unused, so the
                // memory it occupies can be reused by another shot.
                shot->type = UNUSED;
            }
        }
    }
}
```

It simply calls the integrator on each particle in turn. After it has updated the particle, it checks to see whether the particle is below 0 height, in which case it is removed. The particle will also be removed if it is a long way from the firing point (100 m) or if it has been in flight for more than 5 seconds.

In a real game you would use some kind of collision detection system to check on whether the projectile had collided with anything. Additional game logic could then be used to reduce the hit points of the target character or add a bullet-hole graphic to a surface.

Because we have no detailed collision model at this stage, it is difficult to show the effect of the energy in each projectile. When combined with the collisions and contacts in later parts of this book, this is obvious. I've provided a version of the demo (see screenshot in figure 4.2) called **bigballistics** that includes objects to shoot at, which are simulated using the full physics engine. You can clearly see in this version the realism of impacts with the different types of projectile.
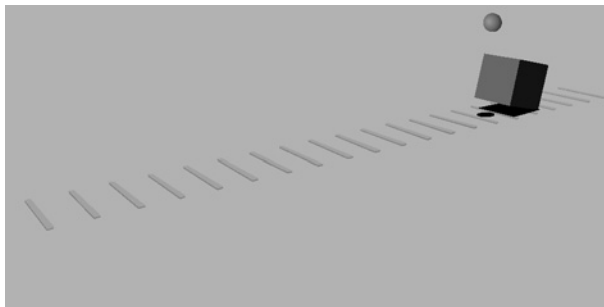
FIGURE 4.2   Screenshot of the **bigballistic** demo.

FIGURE 4.3   Screenshot of the **fireworks** demo.

4.2 Fireworks

Our second example may appear less useful but demonstrates a common application of particle physics used in the majority of games. Fireworks are just a very ostentatious application of a particle system that could be used to display explosions, flowing water, and even smoke and fire.

The **fireworks** demo on the CD allows you to create an interactive fireworks display. You can see a display in progress in figure 4.3.

4.2.1 The Fireworks Data

In our fireworks display we need to add extra data to the basic particle structure. First we need to know what kind of particle it represents. Fireworks consist of a number of payloads: the initial rocket may burst into several lightweight mini-fireworks that explode again after a short delay. We represent the type of firework by an integer value.

Second we need to know the age of the particle. Fireworks consist of a chain reaction of pyrotechnics, with carefully timed fuses. A rocket will first ignite its rocket motor; then, after a short time of flight, the motor will burn out as the explosion stage detonates. This may scatter additional units, which all have a fuse of the same length, allowing the final bursts to occur at roughly the same time (not exactly the same time, however, as that would look odd). To support this we keep an age for each particle and update it at each frame.

The Firework structure can be implemented in this way:

```
                ── Excerpt from src/demos/fireworks/fireworks.cpp ──
/**
 * Fireworks are particles, with additional data for rendering and
 * evolution.
 */
```

```
class Firework : public cyclone::Particle
{
public:
    /** Fireworks have an integer type, used for firework rules. */
    unsigned type;

    /**
     * The age of a firework determines when it detonates. Age gradually
     * decreases; when it passes zero the firework delivers its payload.
     * Think of age as fuse-left.
     */
    cyclone::real age;
};
```

I've used an object-oriented approach here and made the Firework structure a subclass of the particle structure. This allows me to add just the new data without changing the original particle definition.

4.2.2 The Fireworks Rules

To define the effect of the whole fireworks display, we need to be able to specify how one type of particle changes into another. We do this as a set of rules: for each firework type we store an age and a set of data for additional fireworks that will be spawned when the age is passed. This is held in a Rules data structure with this form:

```
                    ── Excerpt from src/demos/fireworks/fireworks.cpp ──
/**
 * Firework rules control the length of a firework's fuse and the
 * particles it should evolve into.
 */
struct FireworkRule
{
    /** The type of firework that is managed by this rule. */
    unsigned type;

    /** The minimum length of the fuse. */
    cyclone::real minAge;

    /** The maximum length of the fuse. */
    cyclone::real maxAge;

    /** The minimum relative velocity of this firework. */
    cyclone::Vector3 minVelocity;
```

```
    /** The maximum relative velocity of this firework. */
    cyclone::Vector3 maxVelocity;

    /** The damping of this firework type. */
    cyclone::real damping;

    /**
     * The payload is the new firework type to create when this
     * firework's fuse is over.
     */
    struct Payload
    {
        /** The type of the new particle to create. */
        unsigned type;

        /** The number of particles in this payload. */
        unsigned count;
    };

    /** The number of payloads for this firework type. */
    unsigned payloadCount;

    /** The set of payloads. */
    Payload *payloads;
};
```

Rules are provided in the code, and they are defined in one function that controls the behavior of all possible fireworks. This is a sample of that function:

___ **Excerpt from src/demos/fireworks/fireworks.cpp** ___

```
void FireworksDemo::initFireworkRules()
{
    // Go through the firework types and create their rules.
    rules[0].setParameters(
        1, // type
        3, 5, // age range
        cyclone::Vector3(-5, -5, -5), // min velocity
        cyclone::Vector3(5, 5, 5), // max velocity
        0.1 // damping
        );

    // ... and so on for other firework types ...
}
```

In a game development studio it is often the art staff who need to decide how the particles in a game will behave. In this case it is inconvenient to have the rules defined in code. Many developers incorporate some kind of text file format into their engine, which can read in easy-to-edit particle rules created by level designers or artists. Some developers have gone further to create specific tools where an artist can interactively tweak the behavior of particles in a WYSIWYG environment. This tool then saves out some file format that defines the rules, to be later read in by the engine.

4.2.3 The Implementation

At each frame, each firework has its age updated and is checked against the rules. If its age is past the threshold, then it will be removed and more fireworks will be created in its place (the last stage of the chain reaction spawns no further fireworks).

Using the particle update function to perform the physics, the firework update function now looks like this:

Excerpt from src/demos/fireworks/fireworks.cpp

```cpp
/**
 * Fireworks are particles, with additional data for rendering and
 * evolution.
 */
class Firework : public cyclone::Particle
{
public:
    /**
     * Updates the firework by the given duration of time. Returns true
     * if the firework has reached the end of its life and needs to be
     * removed.
     */
    bool update(cyclone::real duration)
    {
        // Update our physical state
        integrate(duration);

        // We work backward from our age to zero.
        age -= duration;
        return (age < 0);
    }
};
```

Notice that if we don't have any spare firework slots when a firework explodes into its components, then not all the new fireworks will be calculated. In other words, when resources are tight, older fireworks get priority.

This allows us to put a hard limit on the number of fireworks being processed, which can avoid having the physics slow down when things get busy. Many developers use a different strategy in their engines: they give priority to newly spawned particles and remove old particles to make way. This gives a slightly less pleasing effect in the **fireworks** demo, so I've avoided it.

The code that actually creates new fireworks looks like this:

──────── **Excerpt from src/demos/fireworks/fireworks.cpp** ────────

```cpp
/**
 * Firework rules control the length of a firework's fuse and the
 * particles it should evolve into.
 */
struct FireworkRule
{

    /**
     * Creates a new firework of this type and writes it into the given
     * instance. The optional parent firework is used to base position
     * and velocity on.
     */
    void create(Firework *firework, const Firework *parent = NULL) const
    {
        cyclone::Random r;
        firework->type = type;
        firework->age = r.randomReal(minAge, maxAge);
        if (parent) firework->setPosition(parent->getPosition());

        // The velocity is the particle's velocity.
        cyclone::Vector3 vel = parent->getVelocity();
        vel += r.randomVector(minVelocity, maxVelocity);
        firework->setVelocity(vel);

        // We use a mass of 1 in all cases (no point having fireworks
        // with different masses, since they are only under the influence
        // of gravity).
        firework->setMass(1);

        firework->setDamping(damping);

        firework->setAcceleration(cyclone::Vector3::GRAVITY);

        firework->clearAccumulator();
    }
};
```

```
void FireworksDemo::create(unsigned type, const Firework *parent)
{
    // Get the rule needed to create this firework.
    FireworkRule *rule = rules + (type - 1);

    // Create the firework.
    rule->create(fireworks+nextFirework, parent);

    // Increment the index for the next firework.
    nextFirework = (nextFirework + 1) % maxFireworks;
}
```

As fireworks are spawned, they have their particle properties set, with velocities determined with a random component.

Notice that I've used high damping values for several of the firework types. This allows them to drift back down to the ground slowly, which is especially important for fireworks that need to hang in the air before exploding.

At each frame, all of the currently active fireworks are updated. This is performed by a simple loop that first checks to see whether the firework should be processed (fireworks with a type of 0 are defined to be inactive).

Excerpt from src/demos/fireworks/fireworks.cpp

```
for (Firework *firework = fireworks;
     firework < fireworks+maxFireworks;
     firework++)
{
    // Check if we need to process this firework.
    if (firework->type > 0)
    {
        // Does it need removing?
        if (firework->update(duration))
        {
            // Find the appropriate rule.
            FireworkRule *rule = rules + (firework->type-1);

            // Delete the current firework (this doesn't affect its
            // position and velocity for passing to the create function,
            // just whether or not it is processed for rendering or
            // physics.
            firework->type = 0;

            // Add the payload.
            for (unsigned i = 0; i < rule->payloadCount; i++)
            {
```

```
                    FireworkRule::Payload * payload = rule->payloads + i;
                    create(payload->type, payload->count, firework);
            }
        }
    }
}
```

These code fragments are taken from the **fireworks** demo on the CD. You can create your own fireworks display using the number keys to launch new fireworks (there are nine basic firework types).

Exactly the same kind of particle system is used in many game engines. By setting the gravity of particles to a very low value, or even having gravity pull some kinds of particle upward, we can create smoke, fire, flowing water, explosions, sparks, rain, and many, many other effects.

The difference between each type of particle is simply one of rendering. Particles are normally drawn as a flat bitmap on screen rather than as a three-dimensional model. This is the approach I've used in the demo.

Most production particle systems also allow particles to rotate. Not the full three-dimensional rotation we will cover later in this book, but a screen rotation so that each particle bitmap is not drawn with the same orientation on screen. It can be useful to have this rotation change over time. I will not try to implement the technique in this book. It is a relatively easy tweak to add a constant speed rotation to particles, and I'll leave it as an exercise for those who need it.

4.3 Summary

The particle physics engine is primarily suitable for special effects—namely, the ballistics of projectile weapons and particle systems and the visual effects for explosions.

In this chapter we've used a particle system to render fireworks. There are tens of other uses too. Most games have some kind of particle system at work (often completely separate from the main physics engine). By setting particles with different properties for gravity, drag, and initial velocity, it is possible to simulate everything from flowing water to smoke, from fireballs to fireworks.

Eventually, however, single particles won't be enough. We'll need full three-dimensional objects. In part II of this book we'll look at one way to simulate objects: by building structures out of particles connected by springs, rods, and cables. To handle these structures we'll need to consider more forces than just gravity on particles. Chapter 5 introduces this.

PART II

Mass-Aggregate Physics

This page intentionally left blank
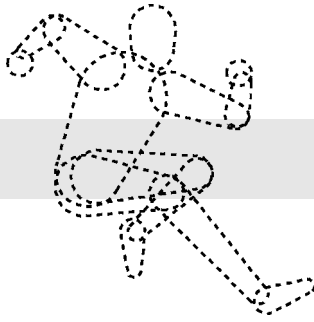
<div style="text-align: right; font-size: 4em;">5</div>

ADDING GENERAL FORCES

I n part I we built a particle physics engine that included the force of gravity. We looked at the mathematics of forces in chapter 3, which let us simulate any force we like by calculating the resulting acceleration.

In this chapter we will extend our physics engine so it can cope with multiple different forces acting at the same time. We will assume that gravity is one force, although this can be removed or set to zero if required. We will also look at force generators: code that can calculate forces based on the current state of the game world.

5.1 D'ALEMBERT'S PRINCIPLE

Although we have equations for the behavior of an object when a force is acting on it, we haven't considered what happens when more than one force is acting. Clearly the behavior is going to be different than if either force acts alone: one force could be acting in the opposite direction to another, or reinforcing it in parallel. We need a mechanism to work out the overall behavior as a result of all forces.

D'Alembert's principle comes to the rescue here. The principle itself is more complex and far-reaching than we'll need to consider here. It relates quantities in a different formulation of the equations of motion. But it has two important implications that we'll make use of in this book. The first applies here; the second will arise in chapter 10.

For particles D'Alembert's principle implies that, if we have a set of forces acting on an object, we can replace all those forces with a single force, which is calculated by

$$f = \sum_i f_i$$

In other words, we simply add the forces together using vector addition, and we apply the single force that results.

To make use of this result, we use a vector as a force accumulator. In each frame we zero the vector and add each applied force in turn using vector addition. The final value will be the resultant force to apply to the object. We add a method to the particle that is called at the end of each integration step to clear the accumulator of the forces that have just been applied:

<div align="center">— Excerpt from include/cyclone/particle.h —</div>

```
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Holds the accumulated force to be applied at the next
     * simulation iteration only. This value is zeroed at each
     * integration step.
     */
    Vector3 forceAccum;

    /**
     * Clears the forces applied to the particle. This will be
     * called automatically after each integration step.
     */
    void clearAccumulator();
};
```

<div align="center">— Excerpt from src/particle.cpp —</div>

```
void Particle::integrate(real duration)
{
    assert(duration > 0.0);

    // Update linear position.
    position.addScaledVector(velocity, duration);

    // Work out the acceleration from the force.
    Vector3 resultingAcc = acceleration;
    resultingAcc.addScaledVector(forceAccum, inverseMass);
```

```
    // Update linear velocity from the acceleration.
    velocity.addScaledVector(resultingAcc, duration);

    // Impose drag.
    velocity *= real_pow(damping, duration);

    // Clear the forces.
    clearAccumulator();
}

void Particle::clearAccumulator()
{
    forceAccum.clear();
}
```

We then add a method that can be called to add a new force into the accumulator:

```
——————— Excerpt from include/cyclone/particle.h ———————
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Adds the given force to the particle, to be applied at the
     * next iteration only.
     *
     * @param force The force to apply.
     */
    void addForce(const Vector3 &force);
};
```

```
——————— Excerpt from src/particle.cpp ———————
void Particle::addForce(const Vector3 &force)
{
    forceAccum += force;
}
```

This accumulation stage needs to be completed just before the particle is integrated. All the forces that apply need to have the chance to add themselves to the accumulator.

We can do this by manually adding code to our frame update loop that adds the appropriate forces. This is appropriate for forces that will only occur for a few frames. Most forces will apply to an object over an extended period of time.

We can make it easier to manage these long-term forces by creating a registry. A force registers itself with a particle and then will be asked to provide a force at each frame. I call these "force generators."

5.2 Force Generators

We have a mechanism for applying multiple forces to an object. We now need to work out where these forces come from. The force of gravity is fairly intuitive: it is always present for all objects in the game.

Some forces arise because of the behavior of an object—a dedicated drag force, for example. Other forces are a consequence of the environment that an object finds itself in: a buoyancy force for a floating object and the blast force from an explosion are examples. Still other types of force are a result of the way objects are connected together; we will look at forces that behave like springs in the next chapter. Finally there are forces that exist because the player (or an AI-controlled character) has requested them: the acceleration force in a car or the thrust from a jetpack, for example.

Another complication is the dynamic nature of some forces. The force of gravity is easy because it is always constant. We can calculate it once and leave it set for the rest of the game. Most other forces are constantly changing. Some change as a result of the position or velocity of an object: drag is stronger at higher speeds, and a spring's force is greater the more it is compressed. Others change because of external factors: an explosion dissipates, or a player's jetpack burst will come to a sudden end when he releases the thrust button.

We need to be able to deal with a range of different forces with very different mechanics for their calculation. Some might be constant, others might apply some function to the current properties of the object (such as position and velocity), some might require user input, and others might be time based.

If we simply programmed all these types of force into the physics engine, and set parameters to mix and match them for each object, the code would rapidly become unmanageable. Ideally we would like to be able to abstract away the details of how a force is calculated and allow the physics engine to simply work with forces in general. This would allow us to apply any number of forces to an object, without the object knowing the details of how those forces are calculated.

I will do this through a structure called a "force generator." There can be as many different types of force generator as there are types of force, but each object doesn't need to know how a generator works. The object simply uses a consistent interface to find the force associated with each generator: these forces can then be accumulated and applied in the integration step. This allows us to apply any number of forces, of any type we choose, to the object. It also allows us to create new types of force for new games or levels, as we need to, without having to rewrite any code in the physics engine.

Not every physics engine has the concept of force generators: many require hand-written code to add forces or else limit the possible forces to a handful of common

options. Having a general solution is more flexible and allows us to experiment more quickly.

To implement this we will use an object-oriented design pattern called an "interface." Some languages, such as Java, have this built in as part of the language; in others it can be approximated with a regular class. Before we look at the force generator code, I will briefly review the concept of an interface, and its relative, polymorphism.

5.2.1 Interfaces and Polymorphism

In programming, an *interface* is a specification of how one software component interacts with others. In an object-oriented language it normally refers to a class: an interface is a specification of the methods, constants, data types, and exceptions (i.e., errors) that a class will expose. The interface itself is not a class; it is a specification that any number of classes can fulfill. When a class fulfills the specification, it is said that it implements the interface (in fact, Java uses the explicit `implements` keyword to denote a class that implements an interface).

What is powerful about interfaces is their use in polymorphism. *Polymorphism* is the ability of a language to use some software component on the basis that it fulfills some specification. For our purposes the components are classes and the specification is an interface. If we write some code that needs to use another part of the system, we can define an interface for the interaction and make the calling code use only the elements in the interface. We can later change what the code is interacting with, and as long as it implements the same interface, the calling code will never know the difference.

This replaceability is key for our needs: we have an interface for a force generator, and through polymorphism we don't need to know what kind of force the force generator represents, as long as it implements the interface we need to extract the relevant information. This is a helpful way to avoid having different parts of the program extremely reliant on the way other parts are implemented: we create an interface, and as long as a class implements it, the calling code need know nothing more about it.

In C++ there is no dedicated interface structure in the language. Instead we use a base class, with a selection of pure virtual functions. This ensures that we can't create an instance of the base class. Each class that derives from the base class then has to implement all its methods before it can be instantiated.

5.2.2 Implementation

The force generator interface needs to provide only a current force. This can then be accumulated and applied to the object. The interface we will use looks like this:

```
                    Excerpt from include/cyclone/pfgen.h
/**
 * A force generator can be asked to add a force to one or more
 * particles.
```

```
 */
class ParticleForceGenerator
{
public:

    /**
     * Overload this in implementations of the interface to calculate
     * and update the force applied to the given particle.
     */
    virtual void updateForce(Particle *particle, real duration) = 0;
};
```

The updateForce method is passed for the duration of the frame for which the force is needed and a pointer to the particle requesting the force. The duration of the frame is needed for some force generators (we will meet a spring-force generator that depends critically on this value in chapter 6).

We pass the pointer of the object into the function so that a force generator does not need to keep track of an object itself. This also allows us to create force generators that can be attached to several objects at the same time. As long as the generator instance does not contain any data that is specific to a particular object, it can simply use the object passed in to calculate the force. Both of the example force generators described in sections 5.2.3 and 5.2.4 have this property.

The force generator does not return any value. We could have it return a force to add to the force accumulator, but then force generators would have to return some force (even if it were zero), and that would remove the flexibility we'll use later in the book when we support wholly different kinds of force. Instead, if a force generator wants to apply a force, it can call the addForce method to the object it is passed.

As well as the interface for force generators, we need to be able to register which force generators affect which particles. We could add this into each particle, with a data structure such as a linked list or a growable array of generators. This would be a valid approach, but it has performance implications: either each particle needs to have lots of wasted storage (using a growable array) or new registrations will cause lots of memory operations (creating elements in linked lists). For performance and modularity I think it is better to decouple the design and have a central registry of particles and force generators. The one I have provided looks like this:

—————————— **Excerpt from include/cyclone/pfgen.h** ——————————
```
#include <vector>

namespace cyclone {
    /**
     * Holds all the force generators and the particles they apply to.
     */
    class ParticleForceRegistry
```

```
{
protected:

    /**
     * Keeps track of one force generator and the particle it
     * applies to.
     */
    struct ParticleForceRegistration
    {
        Particle *particle;
        ParticleForceGenerator *fg;
    };

    /**
     * Holds the list of registrations.
     */
    typedef std::vector<ParticleForceRegistration> Registry;
    Registry registrations;

public:
    /**
     * Registers the given force generator to apply to the
     * given particle.
     */
    void add(Particle* particle, ParticleForceGenerator *fg);

    /**
     * Removes the given registered pair from the registry.
     * If the pair is not registered, this method will have
     * no effect.
     */
    void remove(Particle* particle, ParticleForceGenerator *fg);

    /**
     * Clears all registrations from the registry. This will
     * not delete the particles or the force generators
     * themselves, just the records of their connection.
     */
    void clear();

    /**
     * Calls all the force generators to update the forces of
     * their corresponding particles.
     */
```

```
              void updateForces(real duration);
        };
}
```

I have used the C++ standard template library's growable array, vector. The implementations of the first three methods are simple wrappers around corresponding methods in the vector data structure.

At each frame, before the update is performed, the force generators are all called. They will be adding forces to the accumulator which can be used later to calculate each particle's acceleration:

—————— **Excerpt from src/pfgen.cpp** ——————
```cpp
#include <cyclone/pfgen.h>

using namespace cyclone;

void ParticleForceRegistry::updateForces(real duration)
{
    Registry::iterator i = registrations.begin();
    for (; i != registrations.end(); i++)
    {
        i->fg->updateForce(i->particle, duration);
    }
}
```

5.2.3 A Gravity Force Generator

We can replace our previous implementation of gravity by a force generator. Rather than apply a constant acceleration at each frame, gravity is represented as a force generator attached to each particle. The implementation looks like this:

—————— **Excerpt from include/cyclone/pfgen.h** ——————
```cpp
/**
 * A force generator that applies a gravitational force. One instance
 * can be used for multiple particles.
 */
class ParticleGravity : public ParticleForceGenerator
{
    /** Holds the acceleration due to gravity. */
    Vector3 gravity;

public:

    /** Creates the generator with the given acceleration. */
```

```
        ParticleGravity(const Vector3 &gravity);

        /** Applies the gravitational force to the given particle. */
        virtual void updateForce(Particle *particle, real duration);
};
```

──── **Excerpt from src/pfgen.cpp** ────
```
void ParticleGravity::updateForce(Particle* particle, real duration)
{
    // Check that we do not have infinite mass.
    if (!particle->hasFiniteMass()) return;

    // Apply the mass-scaled force to the particle.
    particle->addForce(gravity * particle->getMass());
}
```

Notice that the force is calculated based on the mass of the object passed into the updateForce method. The only piece of data stored by the class is the acceleration due to gravity. One instance of this class could be shared among any number of objects.

5.2.4 A Drag Force Generator

We could also implement a force generator for drag. *Drag* is a force that acts on a body and depends on its velocity. A full model of drag involves more complex mathematics that we can easily perform in real time. Typically in game applications we use a simplified model of drag where the drag acting on a body is given by

$$f_{\text{drag}} = -\widehat{\boldsymbol{p}}\big(k_1|\widehat{\boldsymbol{p}}| + k_2|\widehat{\boldsymbol{p}}|^2\big) \qquad [5.1]$$

where k_1 and k_2 are two constants that characterize how strong the drag force is. They are usually called the "drag coefficients," and they depend on both the object and the type of drag being simulated.

The formula looks complex but is simple in practice. It says that the force acts in the opposite direction to the velocity of the object (this is the $-\widehat{\boldsymbol{p}}$ part of the equation: $\widehat{\boldsymbol{p}}$ is the normalized velocity of the particle), with a strength that depends on both the *speed* of the object and the *square* of the speed.

Drag that has a k_2 value will grow faster at higher speeds. This is the case with the aerodynamic drag that keeps a car from accelerating indefinitely. At slow speeds the car feels almost no drag from the air, but for every doubling of the speed the drag almost quadruples. The implementation for the drag generator looks like this:

──── **Excerpt from include/cyclone/pfgen.h** ────
```
/**
 * A force generator that applies a drag force. One instance
```

```
 * can be used for multiple particles.
 */
class ParticleDrag : public ParticleForceGenerator
{
    /** Holds the velocity drag coefficient. */
    real k1;

    /** Holds the velocity squared drag coefficient. */
    real k2;

public:

    /** Creates the generator with the given coefficients. */
    ParticleDrag(real k1, real k2);

    /** Applies the drag force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

Excerpt from **src/pfgen.cpp**

```
void ParticleDrag::updateForce(Particle* particle, real duration)
{
    Vector3 force;
    particle->getVelocity(&force);

    // Calculate the total drag coefficient.
    real dragCoeff = force.magnitude();
    dragCoeff = k1 * dragCoeff + k2 * dragCoeff * dragCoeff;

    // Calculate the final force and apply it.
    force.normalize();
    force *= -dragCoeff;
    particle->addForce(force);
}
```

Once again the force is calculated based only on the properties of the object it is passed. The only pieces of data stored by the class are the values for the two constants. As before, one instance of this class could be shared among any number of objects that have the same drag coefficients.

This drag model is considerably more complex than the simple damping we used in chapter 3. It can be used to model the kind of drag that a golf ball experiences in flight, for example. For the aerodynamics needed in a flight simulator, however, it may not be sufficient; we will return to flight simulator aerodynamics in chapter 11.

5.3 Built-in Gravity and Damping

Using the generators discussed earlier we can replace both the damping and the acceleration due to gravity with force generators. This is a valid approach and one used by many different engines. It allows us to remove the special code that processes damping, and it means we don't need to store an acceleration due to gravity with the object; it can be calculated among all the other forces during transient force accumulation.

Although it has some advantages in simplicity, this is not the approach I will use. Directly applying the damping and acceleration due to gravity, in the way we did in chapter 3, is fast. If we have to calculate forces for them each time, we waste extra time performing calculations for which we already know the answer.

To avoid this I keep damping and acceleration unchanged. If we need more complex drag, we can set a damping value nearer to 1 and add a drag force generator. Similarly, if we need some exotic form of gravity (for an orbiting space ship, for example), we could create a gravity force generator that provides the correct behavior and set the acceleration due to gravity to be 0.
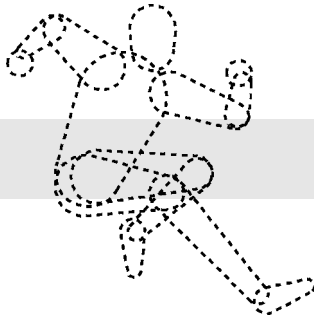
5.4 Summary

Forces are easily combined by adding their vectors together, and the total acts as if it were the only force applied to an object. This is D'Alembert's principle, and it allows us to support any number of general forces without having to know anything about how the forces are generated.

Throughout this book we'll see various kinds of force generators that simulate some kind of physical property by calculating a force to apply to an object. The code we've created in this chapter allows us to manage those forces, combining them and applying them before integrating.

Drag and gravity are important force generators, but they only replicate the functionality we had in our particle physics engine. To move toward a mass-aggregate physics engine we need to start linking particles together. Chapter 6 introduces springs, and other springlike connections, using the force generator structure we've built in this chapter.

This page intentionally left blank

6

SPRINGS AND SPRINGLIKE THINGS

One of the most useful forces we can create for our engine is a spring force. Although springs have an obvious use in driving games (for simulating the suspension of a car), they come into their own in representing soft or deformable objects of many kinds. Springs and particles alone can produce a whole range of impressive effects such as ropes, flags, cloth garments, and water ripples. Along with the hard constraints we'll cover in the next chapter, they can represent almost any kind of object.

To extend our engine to support springs, this chapter will first cover the theory of springs and then look at how they can be created for our engine. Finally we'll look at a major problem in the simulation of springs.

6.1 HOOK'S LAW

Hook's law gives us the mathematics of springs. Hook discovered that the force exerted by a string depends only on the distance the spring is extended or compressed from its rest position. A spring extended twice as far will exert twice the force. The formula is therefore

$$f = -k\Delta l$$

where Δl is the distance the spring is extended or compressed, and k is the "spring constant," a value that gives the stiffness of the spring. The force given in this equation is felt *at both ends* of the spring. In other words, if two objects are connected by a spring, then they will each be attracted together by the same force, given by the preceding equation.

Notice that we have used Δl in the equation. This is because, at rest, with no forces acting to extend or compress the spring, it will have some natural length. This is also called the "rest length" and has the symbol l_0. If the spring is currently at length l, then the force generated by the spring is

$$f = -k(l - l_0)$$

So far we have considered Hook's law only in terms of a one-dimensional spring. When it comes to three dimensions, we need to generate a force vector rather than a scalar. The corresponding formula for the force is

$$f = -k\big(|d| - l_0\big)\widehat{d} \qquad [6.1]$$

where d is the vector from the end of the spring attached to the object we're generating a force for, to the other end of the spring. It is given by

$$d = x_A - x_B \qquad [6.2]$$

where x_A is the position of the end of the spring attached to the object under consideration, and x_B is the position of the other end of the spring.

Equation 6.1 states that the force should be in the direction of the other end of the spring (the $\widehat{d}$ component), with a magnitude given by the spring coefficient multiplied by the amount of extension of the spring—the $-k(|d| - l_0)$ part. The $|d|$ element is the magnitude of the distance between the ends of the spring, which is simply the length of the spring, making $-k(|d| - l_0)$ just a different way of writing $-k(l - l_0)$.

Because equation 6.1 is defined in terms of one end of the spring only (the end attached to the object we are currently considering), we can use it unmodified for the other end of the spring, when we come to process the object attached there. Alternatively, because the two ends of the spring always pull toward each other with the same magnitude of force, we know that if the force on one end is f, then the force on the other will be $-f$.

In the force generator described in this chapter we will calculate the force separately for each object, and do not make use of this fact. A more optimized approach might use the same force generator for both objects involved, and cache the force calculation from one to save time recalculating it for the other. A force generator of this kind is provided in the source code on the CD.
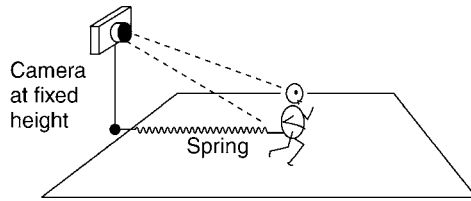
FIGURE 6.1   The game's camera attached to a spring.

6.1.1 The Limit of Elasticity

Real springs only behave this way within a range of lengths: this range is called their "limit of elasticity." If you continue to extend a metal spring, eventually you will exceed its elasticity and it will deform. Similarly, if you compress a spring too much, its coils will touch and further compression is impossible.

We could encode these limits into our force generator to produce a realistic model of a spring. In the vast majority of cases, however, we don't need this sophistication. Players will see a spring doing its most springlike thing; they are unlikely to notice whether it behaves correctly at its limits of elasticity. One exception to this is the case of springs that cannot compress beyond a certain limit. This is the case with car suspensions: they hit their "stop." After being compressed to this point, they no longer act like springs but rather like a collision between two objects. We will cover this kind of hard constraint in the next chapter: it can't be easily modeled using a spring.

6.1.2 Springlike Things

It's not only a coiled metal spring that can be simulated using equation 6.1: Hook's law applies to a huge range of natural phenomena. Anything that has an elastic property will usually have some limit of elasticity in which Hook's law applies.

The applications are limitless. We can implement elastic bungees as springs. We could simulate the buoyancy of water in the same way, connecting the submerged object to the nearest point on the surface with an invisible spring. Some developers even use springs to control the camera as it follows a game character, by applying a spring from the camera to a point just behind the character (see figure 6.1).

6.2 Springlike Force Generators

We will implement four force generators based on spring forces. Although each has a slightly different way of calculating the current length of the spring, they all use Hook's law to calculate the resulting force.

This section illustrates a feature of many physics systems. The core processing engine remains generic, but it is surrounded by helper classes and functions (in this

case the different types of spring force generators) that are often quite similar to one another. In the remainder of this book I will avoid going through similar variations in detail; you can find several suites of similar classes in the source code on the CD. This first time, however, it is worth looking at some variations in detail.

6.2.1 A Basic Spring Generator

The basic spring generator simply calculates the length of the spring using equation 6.2, and then uses Hook's law to calculate the force. It can be implemented like this:

—————— **Excerpt from include/cyclone/precision.h** ——————
```
/** Defines the precision of the absolute magnitude operator. */
#define real_abs fabsf
```

—————— **Excerpt from include/cyclone/pfgen.h** ——————
```
/**
 * A force generator that applies a spring force.
 */
class ParticleSpring : public ParticleForceGenerator
{
    /** The particle at the other end of the spring. */
    Particle *other;

    /** Holds the spring constant. */
    real springConstant;

    /** Holds the rest length of the spring. */
    real restLength;

public:

    /** Creates a new spring with the given parameters. */
    ParticleSpring(Particle *other,
        real springConstant, real restLength);

    /** Applies the spring force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

—————— **Excerpt from src/pfgen.cpp** ——————
```
void ParticleSpring::updateForce(Particle* particle, real duration)
{
    // Calculate the vector of the spring.
    Vector3 force;
```

```
    particle->getPosition(&force);
    force -= other->getPosition();

    // Calculate the magnitude of the force.
    real magnitude = force.magnitude();
    magnitude = real_abs(magnitude - restLength);
    magnitude *= springConstant;

    // Calculate the final force and apply it.
    force.normalize();
    force *= -magnitude;
    particle->addForce(force);
}
```

The generator is created with three parameters. The first is a pointer to the object at the other end of the spring, the second is the spring constant, and the third is the rest length of the spring. We can create and add the generator using this code:

```
Particle a, b;
ParticleForceRegistry registry;

ParticleSpring ps(&b, 1.0f, 2.0f);
registry.add(&a, ps);
```

Because it contains data that depends on the spring, one instance cannot be used for multiple objects, in the way that the force generators from chapter 5 were. Instead we need to create a new generator for each object.[1]

Notice also that the force generator (like the others we have met) creates a force for only one object. If we want to link two objects with a spring, then we'll need to create and register a generator for each.

```
Particle a, b;
ParticleForceRegistry registry;

ParticleSpring psA(&b, 1.0f, 2.0f);
registry.add(&a, psA);
```

---

1. Strictly speaking, we can reuse the force generator. If we have a set of springs, all connected to the same object and having the same values for rest length and spring constant, we can use one generator for all of them. Rather than try to anticipate these situations in practice, it is simpler to assume that instances cannot be reused.
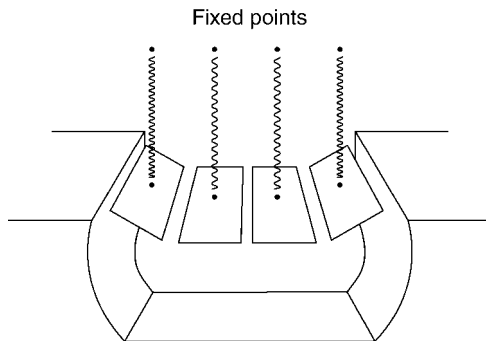
FIGURE 6.2 A rope-bridge held up by springs.

```
ParticleSpring psB(&a, 1.0f, 2.0f);
registry.add(&b, psB);
```

6.2.2 AN ANCHORED SPRING GENERATOR

In many cases we don't want to link two objects together with a spring; rather we want one end of the spring to be at a fixed point in space. This might be the case for the supporting cables on a springy rope-bridge, for example. One end of the spring is attached to the bridge, the other is fixed in space; see figure 6.2 for an example of this.

In this case the form of the spring generator we created earlier will not work. We can modify it so the generator expects a fixed location rather than an object to link to. The force generator code is also modified to use the location directly rather than looking it up in an object. The anchored force generator implementation looks like this:

```
                 ─── Excerpt from include/cyclone/pfgen.h ───
/**
 * A force generator that applies a spring force, where
 * one end is attached to a fixed point in space.
 */
class ParticleAnchoredSpring : public ParticleForceGenerator
{
    /** The location of the anchored end of the spring. */
    Vector3 *anchor;

    /** Holds the spring constant. */
    real springConstant;

    /** Holds the rest length of the spring. */
```

```
        real restLength;

public:

    /** Creates a new spring with the given parameters. */
    ParticleAnchoredSpring(Vector3 *anchor,
        real springConstant, real restLength);

    /** Applies the spring force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

```
─────── Excerpt from src/pfgen.cpp ───────
void ParticleAnchoredSpring::updateForce(Particle* particle,
                                         real duration)
{
    // Calculate the vector of the spring.
    Vector3 force;
    particle->getPosition(&force);
    force -= *anchor;

    // Calculate the magnitude of the force.
    real magnitude = force.magnitude();
    magnitude = real_abs(magnitude - restLength);
    magnitude *= springConstant;

    // Calculate the final force and apply it.
    force.normalize();
    force *= -magnitude;
    particle->addForce(force);
}
```

If we wanted to connect the game's camera to the player's character, this is an approach we would use. Instead of an anchor point that never moves, however, we would recalculate and reset the anchor point at each frame based on the position of the character. The previous implementation needs no modification (other than a setAnchor method to give the new value); we would just need to perform the update of the anchor point somewhere in the game loop.

6.2.3 An Elastic Bungee Generator

An elastic bungee only produces pulling forces. You can scrunch it into a tight ball and it will not push back out, but it behaves like any other spring when extended. This is useful for keeping a pair of objects together: they will be pulled together if they stray too far, but they can get as close as they like without being separated.

The generator can be implemented like this:

```
_____ Excerpt from include/cyclone/pfgen.h _____

/**
 * A force generator that applies a spring force only
 * when extended.
 */
class ParticleBungee : public ParticleForceGenerator
{
    /** The particle at the other end of the spring. */
    Particle *other;

    /** Holds the spring constant. */
    real springConstant;

    /**
     * Holds the length of the bungee at the point it begins to
     * generate a force.
     */
    real restLength;

public:

    /** Creates a new bungee with the given parameters. */
    ParticleBungee(Particle *other,
        real springConstant, real restLength);

    /** Applies the spring force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

```
_____ Excerpt from src/pfgen.cpp _____

void ParticleBungee::updateForce(Particle* particle, real duration)
{
    // Calculate the vector of the spring.
    Vector3 force;
    particle->getPosition(&force);
    force -= other->getPosition();

    // Check if the bungee is compressed.
    real magnitude = force.magnitude();
    if (magnitude <= restLength) return;

    // Calculate the magnitude of the force.
```

```
        magnitude = springConstant * (restLength - magnitude);

        // Calculate the final force and apply it.
        force.normalize();
        force *= -magnitude;
        particle->addForce(force);
}
```

I have added a factory function to this class as well to allow us to easily connect two objects with a bungee.

This implementation assumes that the elastic connects to two objects. In exactly the same way as for the simple spring generator, we could create a version of the code that connects an object to a fixed anchor point in space. The modifications we would need are exactly the same as we saw earlier, so I will not write them out in longhand again. There is a sample implementation of an anchored bungee generator on the CD.

6.2.4 A Buoyancy Force Generator

A buoyancy force is what keeps an object afloat. Archimedes first worked out that the buoyancy force is equal to the weight of the water that an object displaces.

The first part of figure 6.3 shows a block submerged in the water. The block has a mass of 0.5 kg. Pure water has a density of 1000 kg/m^3; in other words, a cubic meter of water has a mass of about one metric ton. The block in the figure has a volume of 0.001 m^3, so it is displacing the same amount of water. The mass of this water would therefore be 1 kg.

Weight isn't the same as mass in physics. *Mass* is the property of an object that makes it resist acceleration. The mass of an object will always be the same. *Weight* is the force that gravity exerts on an object. As we have already seen, force is given by the equation

$$f = mg$$
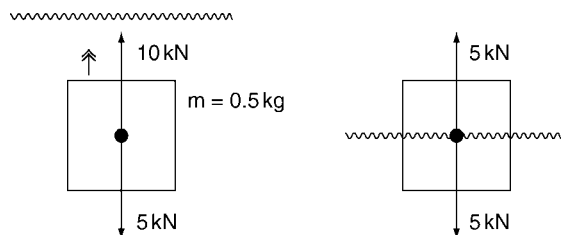
FIGURE 6.3   A buoyant block submerged and partially submerged.

where f is the weight, m is the mass, and g is the acceleration due to gravity. This means that on different planets, the same object will have different weights (but the same mass) because g changes.

On earth, we assume $g = 10$ m/s^2, so an object with a weight of 1 kg will have a weight of $1 \times 10 = 10$ kN. The kN unit is a unit of weight: kilograms, kg, are *not* a unit of weight, despite what your bathroom scales might say! This causes scientists who work on space prejects various problems: because g is different, they can no longer convert English units such as pounds to kilograms using the conversion factors found in science reference books. Pounds are a measure of weight, and kilograms are a measure of mass.

So, back to buoyancy. Our block in the first part of figure 6.3 has a buoyancy force of 10 kN. In the second part of the figure only half is submerged, so using the same calculations, it has a buoyancy force of 5 kN.

Although we don't need to use it for our force generator, it is instructive to look at the weight of the object too. In both cases the weight of the block is the same: 5 kN (a mass of 0.5 kg multiplied by the same value of g). So in the first part of the figure, the buoyancy force will push the block upward; in the second part of the figure the weight is exactly the same as the buoyancy, so the object will stay at the same position—floating.

Calculating the exact buoyancy force for an object involves knowing exactly how it is shaped because the shape affects the volume of water displaced, which is used to calculate the force. Unless you are designing a physics engine specifically to model the behavior of different-shaped boat hulls, it is unlikely that you will need this level of detail.

Instead we can use a springlike calculation as an approximation. When the object is near to the surface, we use a spring force to give it buoyancy. The force is proportional to the depth of the object, just as the spring force is proportional to the extension or compression of the spring. As we saw in figure 6.3, this will be accurate for a rectangular block that is not completely submerged. For any other object it will be slightly inaccurate, but not enough to be noticeable.

When the block is completely submerged, it behaves in a slightly different way. Pushing it deeper into the water will not displace any more water, so as long as we assume water has the same density, the force will be the same. The point-masses we are dealing with in this part of the book have no size, so we can't tell how big they are to determine whether they are fully submerged. We can simply use a fixed depth instead: when we create the buoyancy force, we give a depth at which the object is considered to be fully submerged. At this point the buoyancy force will not increase for deeper submersion.

By contrast, when the object is lifted out of the water, some part of it will still be submerged until it reaches its maximum submersion depth above the surface. At this point the last part of the object will have left the water. In this case there will be no buoyancy force at all, no matter how high we lift the object: it simply is displacing no more water.

The formula for the force calculation is therefore

$$f = \begin{cases} 0 & \text{when } d \leqslant 0 \\ v\rho & \text{when } d \geqslant 1 \\ dv\rho & \text{otherwise} \end{cases}$$

where s is the submersion depth (the depth at which the object is completely submerged), ρ is the density of the liquid, v is the volume of the object, and d is the amount of the object submerged, given as a proportion of its maximum submersion depth (i.e., when it is fully submerged, $d \geqslant 1$, and when it is fully out of the water, $d \leqslant 0$):

$$d = \frac{y_o - y_w - s}{2s}$$

where y_o is the y coordinate of the object and y_w is the y coordinate of the liquid plane (assuming it is parallel to the XZ plane). This can be implemented like this:

Excerpt from include/cyclone/pfgen.h

```
/**
 * A force generator that applies a buoyancy force for a plane of
 * liquid parallel to XZ plane.
 */
class ParticleBuoyancy : public ParticleForceGenerator
{
    /**
     * The maximum submersion depth of the object before
     * it generates its maximum buoyancy force.
     */
    real maxDepth;

    /**
     * The volume of the object.
     */
    real volume;

    /**
     * The height of the water plane above y=0. The plane will be
     * parallel to the XZ plane.
     */
    real waterHeight;

    /**
     * The density of the liquid. Pure water has a density of
     * 1000 kg per cubic meter.
     */
```

```
    real liquidDensity;

public:

    /** Creates a new buoyancy force with the given parameters. */
    ParticleBuoyancy(real maxDepth, real volume, real waterHeight,
        real liquidDensity = 1000.0f);

    /** Applies the buoyancy force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

<div align="center">Excerpt from src/pfgen.cpp</div>

```cpp
void ParticleBuoyancy::updateForce(Particle* particle, real duration)
{
    // Calculate the submersion depth.
    real depth = particle->getPosition().y;

    // Check if we're out of the water.
    if (depth >= waterHeight + maxDepth) return;
    Vector3 force(0,0,0);

    // Check if we're at maximum depth.
    if (depth <= waterHeight - maxDepth)
    {
        force.y = liquidDensity * volume;
        particle->addForce(force);
        return;
    }

    // Otherwise we are partly submerged.
    force.y = liquidDensity * volume *
        (depth - maxDepth - waterHeight) / 2 * maxDepth;
    particle->addForce(force);
}
```

In this code I have assumed that the buoyancy is acting in the up direction. I have used only the *y* component of the object's position to calculate the length of the spring for Hook's law. The generator takes four parameters: the maximum depth parameter, as discussed earlier; the volume of the object; the height of the surface of the water; and the density of the liquid in which it is floating. If no density parameter is given, then water, with a density of 1000 kg/m^3, is assumed (ocean water has a density of around 1020 to 1030 kg/m^3 up to 1250 kg/m^3 for the Dead Sea).

This generator applies to only one object because it contains the data for the object's size and volume. One instance could be given to multiple objects with the same size and volume, floating in the same liquid, but it is probably best to create a new instance per object to avoid confusion.

6.3 STIFF SPRINGS

In real life almost everything acts as a spring. If a rock falls onto the ground, the ground gives a little, like a very stiff spring. With a model of spring behavior we could simulate anything. Collisions between objects could be modeled in a similar way to the buoyancy force: the objects would be allowed to pass into one another (called "interpenetration"), and a spring force would push them back out again.

With the correct spring parameters for each object, this method would give us perfect collisions. It is called the "penalty" method and has been used in many physics simulators, including several used in games.

If life were so simple, however, this book could be two hundred pages shorter. In fact, to avoid having everything in a game look really spongy as things bounce around on soggy springs, we have to increase the spring constant so that it is very high. If you try to do that, and run the engine, you will see everything go haywire: objects will almost instantly disappear off to infinity, and your program may even crash with numerical errors. This is the problem of stiff springs, and it makes penalty methods almost useless for our needs.

6.3.1 THE PROBLEM OF STIFF SPRINGS

To understand why stiff springs cause problems we need to break down the behavior of a spring into short time steps. Figure 6.4 shows a spring's behavior over several time steps. In the first step, the spring is extended, and we calculate the force at that point.

The force is applied to the end of the spring using the update function from chapter 3:

$$\dot{p}' = \dot{p} + \ddot{p}t$$
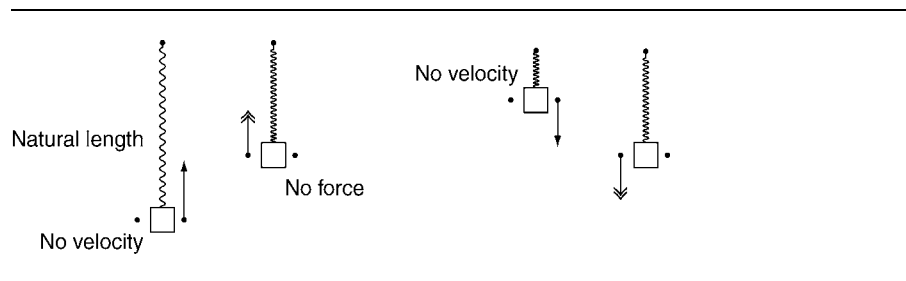
FIGURE 6.4   A non-stiff spring over time.

In other words, the force is converted into an acceleration: the acceleration of the end of the spring *at that instant of time*. This acceleration is then applied to the object *for the whole time interval*. This would be accurate if the object didn't move—that is, if the spring were held at a constant extension over the whole time period.

In the real world, as soon the spring has moved a bit, a tiny fraction of the time interval later, the force will have decreased slightly. So applying the same force for the whole time interval means we have applied too much force. In figure 6.4 we see that this doesn't matter very much. Even though the force is too great, the end doesn't move far before the next time frame, and then a lower force is applied for the next time frame, and so on. The overall effect is that the spring behaves normally, but it is slightly stiffer than the spring constant we specified.

Figure 6.5 shows the same problem but with a much stiffer spring. Now the force in the first frame is enough to carry the end of the spring past the rest length and to compress the spring. In reality the movement of the spring wouldn't do this: it would begin to move inward, having had a huge instantaneous force applied, but this force would drop rapidly as the end moved in.

The figure shows the spring has compressed more than it was extended originally. In the next time frame it moves in the opposite direction but has an even greater force applied, so it overshoots and is extended even farther. In each time frame the spring will oscillate with ever growing forces, until the end of the spring ends up at infinity. Clearly this is not accurate.

The longer the time frame we use the more likely this is to happen. If your game uses springs and variable frame-rates, you need to take care that your spring constants aren't too large when used on a very slow machine. If a player switches all the graphics options on, and slows her machine down to ten frames per second (or slower), you don't want all your physics to explode!
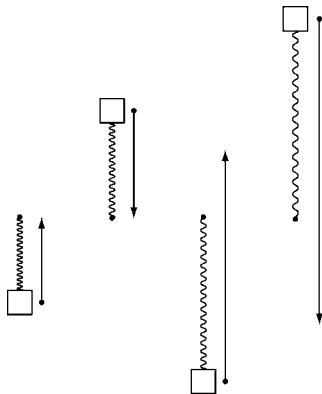
FIGURE 6.5   A stiff spring over time.

We can solve this problem to some extent by forcing small time periods for the update, or we could use several smaller updates for each frame we render. Neither approach buys us much, however. The kinds of spring stiffness needed to simulate realistic collisions just aren't possible in the framework we have built so far.

Instead we will have to use alternative methods to simulate collisions and other hard constraints.

6.3.2 Faking Stiff Springs

This section will implement a more advanced spring force generator which uses a different method of calculating spring forces to help with stiff springs. It provides a "cheat" for making stiff springs work. In the remaining chapters of this book we will look at more robust techniques for simulating constraints, collisions, and contacts.

You can safely skip this section: the mathematics are not explored in detail; there are restrictions on where we can use fake stiff springs, and the formulation is not always guaranteed to work. In particular, while they can fake the effect reasonably on their own, when more than one is combined, or when a series of objects is connected to them, the physical inaccuracies in the calculation can interact nastily and cause serious problems. In the right situation, however, they can be a useful addition to your library of force generators.

We can solve this problem to some extent by predicting how the force will change over the time interval, and use that to generate an average force. This is sometimes called an "implicit spring," and a physics engine that can deal with varying forces in this way is implicit, or semi-implicit. For reasons we'll see at the end of the chapter, we can't do anything more than fake the correct behavior, so I have called this approach "fake implicit force generation."

In order to work out the force equation, we need to understand how a spring will act if left to its own devices.

Harmonic Motion

A spring that had no friction or drag would oscillate forever. If we stretched such a spring to a particular extension and then released it, its ends would accelerate together. It would pass its natural length and begin to compress. When its ends were compressed to exactly the same degree as they were extended initially, it would begin to accelerate apart. This would continue forever. This kind of motion is well known to physicists; it is called "simple harmonic motion." The position of one end of the spring obeys the equation

$$\ddot{\boldsymbol{p}} = -\chi^2 \boldsymbol{p} \qquad [6.3]$$

where k is the spring constant, m is the mass of the object, and χ is defined, for convenience in the following equations, to be

$$\chi = \sqrt{\frac{k}{m}}$$

This kind of equation is called a "differential equation." It links the different differentials together, sometimes with the original quantity: in this case the second differential $\ddot{\boldsymbol{p}}$ and the original $\boldsymbol{p}$. Differential equations can sometimes be solved to give an expression for just the original quantity. In our case the equation can be solved to give us an expression that links the position with the current time.[2] The expression is solved to give

$$\boldsymbol{p}_t = \boldsymbol{p}_0 \cos(\chi t) + \frac{\dot{\boldsymbol{p}}_0}{\chi} \sin(\chi t) \qquad [6.4]$$

where $\boldsymbol{p}_0$ is the position of the end of the spring *relative to the natural length* at the start of the prediction, and $\dot{\boldsymbol{p}}_0$ is the velocity at the same time.

We can substitute into equation 6.4 the time interval we are interested in (i.e., the duration of the current frame), and work out where the spring would end up if it were left to do its own thing. We can then create a force that is just big enough to get it to the correct location over the duration of the frame. If the final location needs to be $\boldsymbol{p}_t$, then the force to get it there would be

$$\boldsymbol{f} = m\ddot{\boldsymbol{p}}$$

and the acceleration $\ddot{\boldsymbol{p}}$ is given by

$$\ddot{\boldsymbol{p}} = (\boldsymbol{p}_t - \boldsymbol{p}_0)\frac{1}{t^2} - \dot{\boldsymbol{p}}_0 \qquad [6.5]$$

Note that, although this gets the particle to the correct place, it doesn't get it there with the correct speed. We'll return to the problems caused by this failing at the end of the section.

Damped Harmonic Motion

A real spring experiences drag as well as spring forces. The spring will not continue to oscillate forever to the same point. Its maximum extension will get less each time, until eventually it settles at the rest length. This gradual decrease is caused by the drag that the spring experiences.

When we run our physics engine normally, the drag will be incorporated in the damping parameter. When we predict the behavior of the spring using the previous formula this does not happen.

---

2. Not all differential equations have a simple solution, although most simple equations of the preceding kind do. Solving differential equations can involve applying a whole range of different techniques and is beyond the scope of this book. When necessary I will provide the answers needed for the physics simulator. If you want to understand more about how I get these answers, you can refer to any undergraduate-level calculus text for more details.

We can include the damping in the equations to give a damped harmonic oscillator. The differential equation 6.3 becomes

$$\ddot{\boldsymbol{p}} = -k\boldsymbol{p} - d\dot{\boldsymbol{p}}$$

where k is the spring constant (no need for χ in this case) and d is a drag coefficient (it matches the k_1 coefficient from equation 5.1 in the previous chapter). This equation doesn't allow for drag that is proportional to the velocity squared—that is, the k_2 value from equation 5.1. If we added this, the mathematics would become considerably more complex for little visible improvement (remember, we're faking this in any case). So we stick with the simplest kind of drag.

Solving the differential equation gives an expression for the position at any time in the future:

$$\boldsymbol{p}_t = \left[\boldsymbol{p}_0 \cos(\gamma t) + \boldsymbol{c} \sin(\gamma t) \right] e^{-\frac{1}{2} dt}$$

where γ is a constant given by

$$\gamma = \frac{1}{2} \sqrt{4k - d^2}$$

and $\boldsymbol{c}$ is a constant given by

$$\boldsymbol{c} = \frac{d}{2\gamma} \boldsymbol{p}_0 + \frac{1}{\gamma} \dot{\boldsymbol{p}}_0$$

Substituting the time interval for t in these equations, as before, we can get a value for $\boldsymbol{p}_t$, and calculate the acceleration required using equation 6.5, as we did for regular harmonic motion.

Implementation

The code to implement a fake implicit spring force generator looks like this:

```
─────── Excerpt from include/cyclone/precision.h ───────
/** Defines the precision of the sine operator. */
#define real_sin sinf

/** Defines the precision of the cosine operator. */
#define real_cos cosf

/** Defines the precision of the exponent operator. */
#define real_exp expf
```

```
─────── Excerpt from include/cyclone/pfgen.h ───────
/**
 * A force generator that fakes a stiff spring force, and where
 * one end is attached to a fixed point in space.
```

```
 */
class ParticleFakeSpring : public ParticleForceGenerator
{
    /** The location of the anchored end of the spring. */
    Vector3 *anchor;

    /** Holds the spring constant. */
    real springConstant;

    /** Holds the damping on the oscillation of the spring. */
    real damping;

public:

    /** Creates a new spring with the given parameters. */
    ParticleFakeSpring(Vector3 *anchor, real springConstant,
        real damping);

    /** Applies the spring force to the given particle. */
    virtual void updateForce(Particle *particle, real duration);
};
```

Excerpt from src/pfgen.cpp

```
void ParticleFakeSpring::updateForce(Particle* particle, real duration)
{
    // Check that we do not have infinite mass.
    if (!particle->hasFiniteMass()) return;

    // Calculate the relative position of the particle to the anchor.
    Vector3 position;
    particle->getPosition(&position);
    position -= *anchor;

    // Calculate the constants and check whether they are in bounds.
    real gamma = 0.5f * real_sqrt(4 * springConstant - damping*damping);
    if (gamma == 0.0f) return;
    Vector3 c = position * (damping / (2.0f * gamma)) +
        particle->getVelocity() * (1.0f / gamma);

    // Calculate the target position.
    Vector3 target = position * real_cos(gamma * duration) +
        c * real_sin(gamma * duration);
    target *= real_exp(-0.5f * duration * damping);
```

```
    // Calculate the resulting acceleration and therefore the force
    Vector3 accel = (target - position) * (1.0f / duration*duration) -
        particle->getVelocity() * duration;
    particle->addForce(accel * particle->getMass());
}
```

The force generator looks like the anchored regular spring generator we created ear-
lier in the chapter, with one critical difference: it no longer has a natural spring length.
This, and the fact that we have used an anchored generator rather than a spring ca-
pable of attaching two objects, is a result of some of the mathematics used here. The
consequence is that we must always have a rest length of zero.

Zero Rest Lengths

If a spring has a zero rest length, then any displacement of one end of the spring results
in extension of the spring. If we fix one end of the spring, then there will always be a
force in the direction of the anchored end.

  For a spring where both ends of the spring are allowed to move, the direction of
the force is much more difficult to determine. The previous formulae assume that the
force can be expressed in terms of the location of the object only. If we didn't anchor
the spring, then we would have to include the motion of the other end of the spring
in the equation, which would make it insoluble.

  A similar problem occurs if we anchor one end but use a non-zero rest length. In
one dimension a non-zero rest length is equivalent to moving the equilibrium point
along a bit, as shown in figure 6.6. The same is true in three dimensions, but because
the spring is allowed to swivel freely, this equilibrium point is now in motion with the
same problems as for a non-anchored spring.

  So the equations only work well for keeping an object at a predetermined, fixed
location. Just as for the previous anchored springs, we can move this location manu-
ally from frame to frame, as long as we don't expect the force generator to cope with
the motion in its prediction.
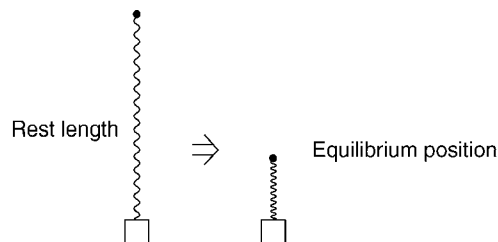
FIGURE 6.6   The rest length and the equilibrium position.

Velocity Mismatches

So far we have talked only about position. Equation 6.5 calculates the force needed to get the object to its predicted position. Unfortunately it will not get there with an accurate velocity (although it will often be close). Could this equation end up increasing the velocity of the object each time, getting faster and faster and exploding out to infinity?

For damped harmonic motion, when the anchor point is not in motion, the velocity resulting from performing this kind of prediction will never mount up to achieve this. The mathematics involved in demonstrating this is complex, so I'll leave it as an exercise for the skeptical.

Even though we won't get exploding velocities, the mismatch between the resulting velocity and the correct velocity causes the spring to behave with an inconsistent spring constant. Sometimes it will be stiffer than we specified, sometimes it will be looser. In most cases it is not noticeable, but it is an inescapable consequence of faking the force in the way we have done.

Interacting with Other Forces

Another major limitation of the faked spring generator is the way it interacts with other forces.

The previous equations assume that the object is moving freely, not under the influence of any other forces. The spring force will decrease over the course of the time interval, because the spring is moving toward its rest length. If we have another force that is keeping the spring extended or compressed at a constant length, then the force *will* be constant, and the original force generator will give a perfect result, no matter what the spring constant is.

We could theoretically incorporate all the other forces into the prediction for the spring generator, and then it would return exactly the correct force. Unfortunately, to correctly work out the force, we'd need to know the behavior of all the objects being simulated. Simulating the behavior of all the objects is, of course, the whole purpose of the physics engine. So the only way we could get this to work is to put a full physics engine in the force calculations. This is not practical (in fact, strictly speaking it is impossible because in that engine we'd need another one, and so on ad infinitum).

For springs that are intended to be kept extended (such as the springs holding up our rope-bridge earlier in the chapter), faked spring forces will be too small, often considerably too small. In practice it is best to try to find a blend of techniques to get the effect you want, using different spring force generators for different objects in the game.

I have used this faked force generator successfully to model the spring in a character's hair (and other wobbly body parts). The rest position is given by the original position of a hair-vertex in the 3D model, and the spring force attracts the actual drawn vertex to this rest position. As the character moves, her hair bobs naturally. This method is ideally suited to the problem because the vertices don't have any other forces on them (a natural "flop" caused by gravity is incorporated by the artist in

the model design), and they need to have very high spring coefficients to avoid looking too bouncy. The **hairbounce** demo on the CD gives a simple example of this in action.

6.4 SUMMARY

A surprising number of physical effects can be modeled using Hook's law. Some effects, such as buoyancy, have such similar properties to a spring that they are most simply supported using the same code.

We've built a set of force generators that can be used alongside the remainder of the book to model anything that should appear elastic or buoyant. But we've also seen the start of a problem that motivates the whole of the rest of the book: springs with high spring constants (i.e., those that have a fast and strong bounce) are difficult to simulate on a frame-by-frame basis. When the action of the spring is faster than the time between simulated frames, then the spring can get unruly and out of control.

If it weren't for this problem, we could simulate almost anything using springlike forces. All collisions, for example, could be easily handled. Even though we were able to fake stiff springs in some cases, the solution wasn't robust enough to cope with stiff springs in the general case, so we need to find alternative approaches (involving significantly more complex code) to handle the very fast bounce of a collision. Chapter 7 looks at this: building a set of special-case code for handling collisions and hard constraints such as rods and inelastic cables.

This page intentionally left blank

7

Hard Constraints

In the last chapter we looked at springs both as a force generator and as one way of having multiple objects affect one another. This is the first time we've had objects that move based on the motion of other objects.

While springs can be used to represent many situations, they can behave badly. When we want objects to be tightly coupled together, the spring constant we need is practically impossible to simulate. For situations where objects are linked by stiff rods, or kept apart by hard surfaces, springs are not a viable option.

In this chapter I'll talk about hard constraints. Initially we'll look at the most common hard constraint—collisions and contact between objects. The same mathematics can be used for other kinds of hard constraints, such as rods or unstretchable cables, that can be used to connect objects together.

To cope with hard constraints in our physics engine we'll need to leave the comfortable world of force generators. All the engines we're building in this book treat hard constraints different from force generators. At the end of the book, in chapter 18, we'll look briefly at alternative approaches that unify them all into one again.

7.1 Simple Collision Resolution

To cope with hard constraints we'll add a collision resolution system to our engine. For the sake of this part of the book, a "collision" refers to any situation in which two objects are touching. In normal English we think about collisions being violent processes where two objects meet with some significant closing velocity. For our purposes this is also true, but two objects that just happen to be touching can be thought of as being in a collision with no closing velocity. The same process we use to resolve

103

high-speed collisions will be used to resolve resting contact. This is a significant assumption that needs justifying, and I'll return to it later in the chapter and at various points further into the book. To avoid changing terminology later, I'll use the terms *collision* and *contact* interchangeably during this chapter.

When two objects collide, their movement after the collision can be calculated from their movement before the collision: this is collision resolution. We resolve the collision by making sure the two objects have the correct motion that would result from the collision. Because collision happens in such a small instant of time (for most objects we can't see the process of collision happening; it appears to be instant), we go in and directly manipulate the motion of each object.

7.1.1   THE CLOSING VELOCITY

The laws governing the motion of colliding bodies depend on their closing velocity. The *closing velocity* is the total speed at which the two objects are moving together.

Note also that this is a closing velocity, rather than a speed, even though it is a scalar quantity. Speeds have no direction; they can only have positive (or zero) values. Velocities can have direction. If we have vectors as velocities, then the direction is the direction of the vector; but if we have a scalar value, then the direction is given by the sign of the value. Two objects that are moving apart from each other will have a closing velocity that is less than zero, for example.

We calculate the closing velocity of two objects by finding the component of their velocity in the direction from one object to another:

$$v_c = \dot{\boldsymbol{p}}_a \cdot (\widehat{\boldsymbol{p}_b - \boldsymbol{p}_a}) + \dot{\boldsymbol{p}}_b \cdot (\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b})$$

where v_c is the closing velocity (a scalar quantity), $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ are the positions of objects a and b, the dot (˙) is the scalar product, and $\widehat{\boldsymbol{p}}$ is the unit-length vector in the same direction as p. This can be simplified to give

$$v_c = -(\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b) \cdot (\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b}) \qquad [7.1]$$

Although it is just a convention, it is more common to change the sign of this quantity. Rather than a closing velocity, we have a separating velocity. The closing velocity is the velocity of one object relative to another, in the direction between the two objects.

In this case two objects that are closing in on each other will have a negative relative velocity, and objects that are separating will have a positive velocity. Mathematically this is simply a matter of changing the sign of equation 7.1 to give

$$v_s = (\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b) \cdot (\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b}) \qquad [7.2]$$

where v_s is the separating velocity, which is the format we'll use in the rest of this book. You can stick with closing velocities if you like: it is simply a matter of preference, although you'll have to flip the sign of various quantities in the engine to compensate.

7.1.2 THE COEFFICIENT OF RESTITUTION

As we saw in the last chapter, when two objects collide, they compress together, and the springlike deformation of their surfaces causes forces to build up that bring the objects apart. This all happens in a very short space of time (too fast for us to simulate frame by frame, although long enough to be captured on very high-speed film). Eventually the two objects will no longer have any closing velocity. Although this behavior is springlike, in reality there is more going on.

All kinds of things can be happening during this compression, and the peculiarities of the materials involved can cause very complicated interactions to take place. In reality the behavior does not conform to that of a damped spring, and we can't hope to capture the subtleties of the real process.

In particular the spring model assumes that momentum is conserved during the collision:

$$m_a \dot{\boldsymbol{p}}_a + m_b \dot{\boldsymbol{p}}_b = m_a \dot{\boldsymbol{p}}'_a + m_b \dot{\boldsymbol{p}}'_b \qquad [7.3]$$

where m_a is the mass of object a, $\dot{\boldsymbol{p}}_a$ is the velocity of object a before the collision, and $\dot{\boldsymbol{p}}'_a$ is the velocity after the collision.

Fortunately the vast majority of collisions behave almost like the springlike ideal. We can produce perfectly believable behavior by assuming the conservation of momentum, and we will use equation 7.3 to model our collisions.

Equation 7.3 tells us about the total velocity before and after the collision, but it doesn't tell us about the individual velocities of each object. The individual velocities are linked together using the closing velocity, according to the equation

$$v'_s = -cv_s$$

where v'_s is the separating velocity after the collision, v_s is the separating velocity before the collision, and c is a constant called the "coefficient of restitution."

The coefficient of restitution controls the speed at which the objects will separate after colliding. It depends on the materials in collision. Different pairs of material will have different coefficients. Some objects such as billiard balls or a tennis ball on a racket bounce apart. Other objects stick together when they collide—a snowball and someone's face.

If the coefficient is 1, then the objects will bounce apart with the same speed as when they were closing. If the coefficient is 0, then the objects will coalesce and travel together (i.e., their separating velocity will be 0). Regardless of the coefficient of restitution, equation 7.3 will still hold: the total momentum will be the same.

Using the two equations, we can get values for $\dot{\boldsymbol{p}}'_a$ and $\dot{\boldsymbol{p}}'_b$.

7.1.3 THE COLLISION DIRECTION AND THE CONTACT NORMAL

So far we've talked in terms of collisions between two objects. Often we also want to be able to support collisions between an object and something we're not physically

simulating. This might be the ground, the walls of a level, or any other immovable object. We could represent these as objects of infinite mass, but it would be a waste of time: by definition they never move.

If we have a collision between one object and some piece of immovable scenery, then we can't calculate the separating velocity in terms of the vector between the location of each object: we only have one object. In other words we can't use the $(\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b})$ term in equation 7.2; we need to replace it.

The $(\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b})$ term gives us the direction in which the separating velocity is occurring. The separating velocity is calculated by the dot product of the relative velocity of the two objects and this term. If we don't have two objects, we can ask that the direction be given to us explicitly. It is the direction in which the two objects are colliding and is usually called the "collision normal" or "contact normal." Because it is a direction, the vector should always have a magnitude of 1.

In cases where we have two particles colliding, the contact normal will always be given by

$$\widehat{\boldsymbol{n}} = (\widehat{\boldsymbol{p}_a - \boldsymbol{p}_b})$$

By convention we always give the contact normal from object a's perspective. In this case, from a's perspective, the contact is incoming from b, so we use $\boldsymbol{p}_a - \boldsymbol{p}_b$. To give the direction of collision from b's point of view we could simply multiply by -1. In practice we don't do this explicitly, but factor this inversion into the code used to calculate the separating velocity for b. You'll notice this in the code we implement later in the chapter: a minus sign appears in b's calculations.

When a particle is colliding with the ground, we only have an object a (the particle) and no object b. In this case from object a's perspective, the contact normal will be

$$\widehat{\boldsymbol{n}} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

assuming that the ground is level at the point of collision.

When we leave particles and begin to work with full rigid bodies, having an explicit contact normal becomes crucial even for inter-object collisions. Without preempting later chapters, figure 7.1 gives a taste of the situation we might come across. Here the two colliding objects, by virtue of their shapes, have a contact normal in almost exactly the opposite direction from that we'd expect if we simply considered their locations. The objects arch over each other, and the contact is acting to prevent them from moving apart rather than keeping them together. At the end of this chapter we'll look at similar situations for particles, which can be used to represent rods and other stiff connections.

With the correct contact normal, equation 7.2 becomes

$$v_s = (\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b) \cdot \widehat{\boldsymbol{n}} \tag{7.4}$$
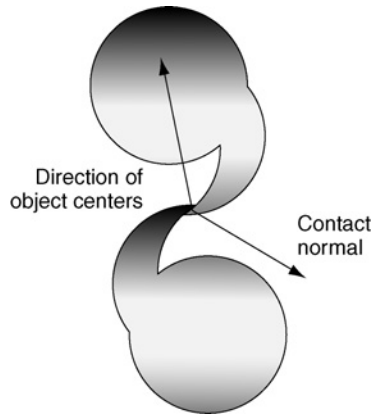
FIGURE 7.1   Contact normal is different from the vector between objects in contact.

7.1.4 IMPULSES

The change we must make to resolve a collision is a change in velocity only. So far in the physics engine we've only made changes to velocity using acceleration. If the acceleration is applied for a long time, there will be a larger change in velocity. Here the changes are instant: the velocities immediately take on new values.

Recall that applying a force changes the acceleration of an object. If we instantly change the force, the acceleration instantly changes too. We can think of acting on an object to change its velocity in a similar way. Rather than a force, this is called an "impulse": an instantaneous change in velocity. In the same way that we have

$$f = m\ddot{p}$$

for forces, we have

$$g = m\dot{p} \tag{7.5}$$

for impulses, g. Impulses are often written with the letter p; I will use g to avoid confusion with the position of the object p.

There is a major difference, however, between force and impulse. An object has no acceleration unless it is being acted on by a force: we can work out the total acceleration by combining all the forces using D'Alembert's principle. On the other hand, an object will continue to have a velocity even if no impulses (or forces) are acting on it. The impulse therefore changes the velocity; it is not completely responsible for the velocity. We can combine impulses using D'Alembert's principle, but the result will

be the total change in velocity, not the total velocity:

$$\dot{p}' = \dot{p} + \frac{1}{m} \sum_n g_i$$

where $g_1 \ldots g_n$ is the set of all impulses acting on the object. In practice we won't accumulate impulses in the way we did forces. We will apply impulses as they arise during the collision resolution process. Each will be applied one at a time using the equation

$$\dot{p}' = \dot{p} + \frac{1}{m} g$$

The result of our collision resolution will be an impulse to apply to each object. The impulse will be immediately applied and will instantly change the velocity of the object.

7.2 COLLISION PROCESSING

To handle collisions we will create a new piece of code—the ContactResolver. It has the job of taking a whole set of collisions and applying the relevant impulses to the objects involved. Each collision is provided in a Contact data structure, which looks like this:

```
                    ── Excerpt from include/cyclone/pcontacts.h ──
/**
 * A contact represents two objects in contact (in this case
 * ParticleContact representing two particles). Resolving a
 * contact removes their interpenetration, and applies sufficient
 * impulse to keep them apart. Colliding bodies may also rebound.
 *
 * The contact has no callable functions, it just holds the
 * contact details. To resolve a set of contacts, use the particle
 * contact resolver class.
 */
class ParticleContact
{
public:
    /**
     * Holds the particles that are involved in the contact. The
     * second of these can be NULL, for contacts with the scenery.
     */
    Particle* particle[2];

    /**
```

```
     * Holds the normal restitution coefficient at the contact.
     */
    real restitution;

    /**
     * Holds the direction of the contact in world coordinates.
     */
    Vector3 contactNormal;
};
```

The structure holds a pointer to each object involved in the collision; a vector representing the contact normal, from the first object's perspective; and a data member for the coefficient of restitution for the contact. If we are dealing with a collision between an object and the scenery (i.e., there is only one object involved), then the pointer for the second object will be NULL.

To resolve one contact we implement the collision equations from earlier in the section to give

Excerpt from include/cyclone/pcontacts.h

```
class ParticleContact
{
    // ... Other ParticleContact code as before ...

protected:
    /**
     * Resolves this contact, for both velocity and interpenetration.
     */
    void resolve(real duration);

    /**
     * Calculates the separating velocity at this contact.
     */
    real calculateSeparatingVelocity() const;

private:
    /**
     * Handles the impulse calculations for this collision.
     */
    void resolveVelocity(real duration);
};
```

Excerpt from src/pcontacts.cpp

```
#include <cyclone/pcontacts.h>
void ParticleContact::resolve(real duration)
```

```
{
    resolveVelocity(duration);
}

real ParticleContact::calculateSeparatingVelocity() const
{
    Vector3 relativeVelocity = particle[0]->getVelocity();
    if (particle[1]) relativeVelocity -= particle[1]->getVelocity();
    return relativeVelocity * contactNormal;
}

void ParticleContact::resolveVelocity(real duration)
{
    // Find the velocity in the direction of the contact.
    real separatingVelocity = calculateSeparatingVelocity();

    // Check whether it needs to be resolved.
    if (separatingVelocity > 0)
    {
        // The contact is either separating or stationary - there's
        // no impulse required.
        return;
    }

    // Calculate the new separating velocity.
    real newSepVelocity = -separatingVelocity * restitution;


    real deltaVelocity = newSepVelocity - separatingVelocity;

    // We apply the change in velocity to each object in proportion to
    // its inverse mass (i.e., those with lower inverse mass [higher
    // actual mass] get less change in velocity).
    real totalInverseMass = particle[0]->getInverseMass();
    if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

    // If all particles have infinite mass, then impulses have no effect.
    if (totalInverseMass <= 0) return;

    // Calculate the impulse to apply.
    real impulse = deltaVelocity / totalInverseMass;

    // Find the amount of impulse per unit of inverse mass.
    Vector3 impulsePerIMass = contactNormal * impulse;
```

```
      // Apply impulses: they are applied in the direction of the contact,
      // and are proportional to the inverse mass.
      particle[0]->setVelocity(particle[0]->getVelocity() +
          impulsePerIMass * particle[0]->getInverseMass()
          );
      if (particle[1])
      {
            // Particle 1 goes in the opposite direction.
            particle[1]->setVelocity(particle[1]->getVelocity() +
            impulsePerIMass * -particle[1]->getInverseMass()
            );
      }
}
```

This directly changes the velocities of each object to reflect the collision.

7.2.1 COLLISION DETECTION

The collision points will normally be found using a collision detector. A collision detector is a chunk of code responsible for finding pairs of objects that are colliding or single objects that are colliding with some piece of immovable scenery.

In our engine the end result of the collision detection algorithm is a set of Contact data structures, filled with the appropriate information. Collision detection obviously needs to take the geometries of the objects into account: their shape and size. So far in the physics engine, we've assumed we are dealing with particles, which lets us avoid taking geometry into account at all.

This is a distinction we'll keep intact even with more complicated 3D objects: the physics simulation system (that part of the engine that handles laws of motion, collision resolution, and forces) will not need to know the details of the shape of the objects it is dealing with. The collision detection system is responsible for calculating any properties that are geometrical, such as when and where two objects are touching, and the contact normal between them.

There is a whole range of algorithms used for working out contact points, and we'll implement a range of useful collision detection routines for full 3D objects in chapter 12. For now we'll assume this is a magic process hidden inside a black box.

There is one exception: I'll cover the simplest possible collision detection for particles represented as small spheres in the next chapter. This will allow us to build some useful physics systems with just the mass-aggregate engine we are constructing. Other than that I'll leave the details until after we've looked at full 3D rigid bodies in chapter 10.

Some collision detection algorithms can take into account the way objects are moving and try to predict likely collisions in the future. Most simply look through the set of objects and check to see whether any two objects are interpenetrating.
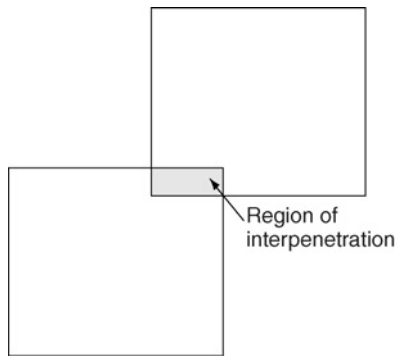
FIGURE 7.2   Interpenetrating objects.

Two objects are interpenetrating if they are partially embedded in each other, as shown in figure 7.2. When we're processing a collision between partially embedded objects, it is not enough to only change their velocity. If the objects are colliding with a small coefficient of restitution, their separation velocity might be almost zero. In this case they will never move apart, and the player will see the objects stuck together in an impossible way.

As part of resolving the collisions, we need to resolve the interpenetration.

7.2.2 RESOLVING INTERPENETRATION

When two objects are interpenetrating, we move them apart just enough to separate them. We expect the collision detector to tell us how far the objects have interpenetrated, as part of the Contact data structure it creates. The calculation of the interpenetration depth depends on the geometries of the objects colliding, and as we saw earlier, this is the domain of the collision detection system rather than the physics simulator.

We add a data member to the Contact data structure to hold this information:

Excerpt from **include/cyclone/pcontacts.h**

```
class ParticleContact
{
    // ... Other ParticleContact code as before ...

    /**
     * Holds the depth of penetration at the contact.
     */
    real penetration;
};
```
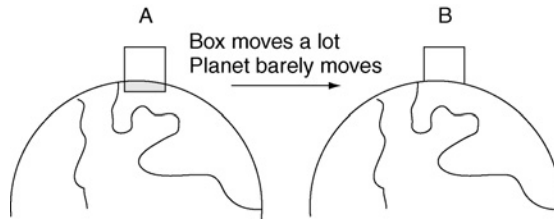
FIGURE 7.3   Interpenetration and reality.

Note that, just like the closing velocity, the penetration depth has both size and sign. A negative depth represents two objects that have no interpenetration. A depth of zero represents two objects that are merely touching.

To resolve the interpenetration we check the interpenetration depth. If it is already zero or less, then we need take no action; otherwise, we can move the two objects apart just far enough so that the penetration depth becomes zero. The penetration depth should be given in the direction of the contact normal. If we move the objects in the direction of the contact normal, by a distance equal to the penetration depth, then the objects will no longer be in contact. The same occurs when we have just one object involved in the contact (i.e., it is interpenetrating with the scenery of the game): the penetration depth is in the direction of the contact normal.

So we know the total distance that the objects must be moved (i.e., the depth) and the direction in which they will be moving; we need to work out how much each individual object should be moved.

If we have only one object involved in the contact, then this is simple: the object needs to move the entire distance. If we have two objects, then we have a whole range of choices. We could simply move each object by the same amount: by half of the interpenetration depth. This would work in some situations but can cause believability problems. Imagine we are simulating a small box resting on a planet's surface. If the box is found slightly interpenetrating the surface, should we move the box and the planet out of the way by the same amount?

We have to take into account how the interpenetration came to be in the first place, and what would have happened in the same situation in reality. Figure 7.3 shows the box and planet, in penetration, as if real physics were in operation. We'd like to get as near to the situation in part B of the figure as possible.

To do this we move two objects apart in inverse proportion to their mass. An object with a large mass gets almost no change, and an object with a tiny mass gets to move a lot. If one of the objects has infinite mass, then it will not move: the other object gets moved the entire way.

The total motion of each object is equal to the depth of interpenetration:

$$\Delta p_a + \Delta p_b = d$$

where Δp_a is the scalar distance that object *a* will be moved (we'll return to the direction later). The two distances are related to each other according to the ratio of their masses:

$$m_a \Delta p_a = m_b \Delta p_b$$

which combined gives us

$$\Delta p_a = \frac{m_b}{m_a + m_b} d$$

and

$$\Delta p_b = \frac{m_a}{m_a + m_b} d$$

Combining these with the direction from the contact normal, we get a total change in the vector position of

$$\Delta \boldsymbol{p}_a = \frac{m_b}{m_a + m_b} d\boldsymbol{n}$$

and

$$\Delta \boldsymbol{p}_b = -\frac{m_a}{m_a + m_b} d\boldsymbol{n}$$

where $\boldsymbol{n}$ is the contact normal. (Note the minus sign in the second equation: this is because the contact normal is given from object *a*'s perspective.)

We can implement the interpenetration resolution equations with this function:

Excerpt from include/cyclone/pcontacts.h

```
class ParticleContact
{
    // ... Other ParticleContact code as before ...

    /**
     * Handles the interpenetration resolution for this contact.
     */
    void resolveInterpenetration(real duration);
};
```

Excerpt from src/pcontacts.cpp

```
void ParticleContact::resolve(real duration)
{
    resolveVelocity(duration);
    resolveInterpenetration(duration);
}

void ParticleContact::resolveInterpenetration(real duration)
{
```

```
    // If we don't have any penetration, skip this step.
    if (penetration <= 0) return;

        // The movement of each object is based on its inverse mass, so
        // total that.
    real totalInverseMass = particle[0]->getInverseMass();
    if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

    // If all particles have infinite mass, then we do nothing.
    if (totalInverseMass <= 0) return;

    // Find the amount of penetration resolution per unit of inverse mass.
    Vector3 movePerIMass = contactNormal *
        (-penetration / totalInverseMass);

    // Apply the penetration resolution.
    particle[0]->setPosition(particle[0]->getPosition() +
        movePerIMass * particle[0]->getInverseMass());
    if (particle[1])
    {
        particle[1]->setPosition(particle[1]->getPosition() +
            movePerIMass * particle[1]->getInverseMass());
    }
}
```

We now have code to apply the change in velocity at a collision and to resolve objects that are interpenetrating. If you implement and run the contact resolution system, it will work well for medium-speed collisions, but objects resting (a particle resting on a table, for example) may appear to vibrate and may even leap into the air occasionally.[1]

To have a complete and stable contact resolution system we need to reconsider what happens when two objects are touching but have a very small or zero closing velocity.

---

1. I said medium-speed here because very high-speed collisions are notoriously difficult to cope with. The physics simulation we've provided will usually cope (except for insanely high speeds where a lack of floating-point accuracy starts to cause problems), but collision detectors can start to provide strange results: it is possible for two objects to pass right through each other before the collision detector realizes they have even touched. If it does detect a collision, they may be at least halfway through each other and be separating again, in which case they have a positive separating velocity and no impulse is generated. We'll return to these issues when we create our collision detection system later in this book, although we will not be able to resolve them fully: they are a feature of very high-speed collision detection.

7.2.3 RESTING CONTACTS

Consider the situation shown in figure 7.4. We have a particle resting on the ground. It is experiencing only one force, gravity. In the first frame the particle accelerates downward. Its velocity increases, but its position stays constant (it has no velocity at the start of the frame). In the second frame the position is updated, and the velocity increases again. Now it is moving downward and has begun to interpenetrate with the ground. The collision detector picks up on the interpenetration and generates a collision.

The contact resolver looks at the particle and sees that it has a penetrating velocity of

$$\dot{p} = 2\ddot{p}t$$

Applying the collision response, the particle is given a velocity of

$$\dot{p}' = c\dot{p} = c2\ddot{p}t$$

and is moved out of interpenetration. In frame 3, therefore, it has an upward velocity, which will carry it off the ground and into the air. The upward velocity will be small, but it may be enough to be noticed. In particular, if frame 1 or 2 is abnormally long, the velocity will have a chance to significantly build up and send the particle skyward. If you implement this algorithm for a game with a variable frame-rate and then slow down the frame-rate (by dragging a window around, for example, or having email arrive in the background), any resting objects will suddenly jump.

To solve this problem we can do two things. First we need to detect the contact earlier. In the example two frames have passed before we find out that there is a problem. If we set our collision detector so it returns contacts that are nearly, but not quite interpenetrating, then we get a contact to work with after frame 1.

Second we need to recognize when an object has velocity that could only have arisen from its forces acting for one frame. After frame 1, the velocity of the particle is caused solely by the force of gravity acting on it for one frame. We can work out what the velocity would be if only the force acted on it, by simply multiplying the force by the frame duration. If the actual velocity of the object is less than or equal to this value (or even slightly above it, if we acknowledge that rounding errors can creep in), we
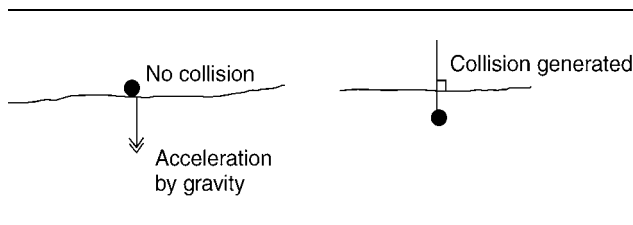
FIGURE 7.4   Vibration on resting contact.

know that the particle was stationary at the previous frame. In this case the contact is likely to be a resting contact rather than a colliding contact. Instead of performing the impulse calculation for a collision, we can apply the impulse that would result in a zero separating velocity.

This is what would happen for a resting contact: no closing velocity would have time to build up, so there would be no separating velocity after the contact. In our case we see that the velocity we do have is likely to be only a by-product of the way we split time into frames, and we can therefore treat the object as if it had a zero velocity before the contact. The particle is given a zero velocity. This happens at every frame: in effect the particle always remains at frame 1 in figure 7.4.

You could also look at this a different way. We are performing a collision with a zero coefficient of restitution at each frame. These series of micro-collisions keep the objects apart. For this reason an engine that handles resting contact in this way is sometimes called a "micro-collision engine."

Velocity and the Contact Normal

When we have two objects in resting contact, we are interested in their relative velocity rather than the absolute velocity of either. The two objects might be in resting contact with each other in one direction, but moving across each other in another direction. A box might be resting on the ground even though it is skidding across the surface at the same time. We want the vibrating contacts code to cope with pairs of objects that are sliding across each other. This means we can't use the absolute velocity of either object.

To cope with this situation the velocity and acceleration calculations are all performed in the direction of the contact normal only. We first find the velocity in this direction and test to see whether it could have been solely caused by the component of the acceleration in the same direction. If so, then the velocity is changed so there is no separating or closing velocity in this direction. There still may be relative velocity in any other direction, but it is ignored.

We can add special-case code to the collision processing function in this way:

```
──────── Excerpt from src/pcontacts.cpp ────────
void ParticleContact::resolveVelocity(real duration)
{
    // Find the velocity in the direction of the contact.
    real separatingVelocity = calculateSeparatingVelocity();

    // Check whether it needs to be resolved.
    if (separatingVelocity > 0)
    {
        // The contact is either separating or stationary - there's
        // no impulse required.
        return;
    }
```

```
// Calculate the new separating velocity.
real newSepVelocity = -separatingVelocity * restitution;

// Check the velocity build-up due to acceleration only.
Vector3 accCausedVelocity = particle[0]->getAcceleration();
if (particle[1]) accCausedVelocity -= particle[1]->getAcceleration();
real accCausedSepVelocity = accCausedVelocity *
    contactNormal * duration;

// If we've got a closing velocity due to acceleration build-up,
// remove it from the new separating velocity.
if (accCausedSepVelocity < 0)
{
    newSepVelocity += restitution * accCausedSepVelocity;

    // Make sure we haven't removed more than was
    // there to remove.
    if (newSepVelocity < 0) newSepVelocity = 0;
}

real deltaVelocity = newSepVelocity - separatingVelocity;

// We apply the change in velocity to each object in proportion to
// its inverse mass (i.e., those with lower inverse mass [higher
// actual mass] get less change in velocity).
real totalInverseMass = particle[0]->getInverseMass();
if (particle[1]) totalInverseMass += particle[1]->getInverseMass();

// If all particles have infinite mass, then impulses have no effect.
if (totalInverseMass <= 0) return;

// Calculate the impulse to apply.
real impulse = deltaVelocity / totalInverseMass;

// Find the amount of impulse per unit of inverse mass.
Vector3 impulsePerIMass = contactNormal * impulse;

// Apply impulses: they are applied in the direction of the contact,
// and are proportional to the inverse mass.
particle[0]->setVelocity(particle[0]->getVelocity() +
    impulsePerIMass * particle[0]->getInverseMass()
    );
if (particle[1])
```

```
    {
        // Particle 1 goes in the opposite direction.
        particle[1]->setVelocity(particle[1]->getVelocity() +
        impulsePerIMass * -particle[1]->getInverseMass()
            );
    }
}
```

To keep two objects in resting contact we are applying a small change in velocity at each frame. The change is just big enough to correct the increase in velocity that would arise from their settling into each other over the course of one frame.

Other Approaches to Resting Contact

The micro-collision approach I described above is only one of many possibilities. Resting contact is one of two key challenges to get right in a physics engine (the other being friction: in fact the two often go together). There are many routes of attack as well as countless variations and tweaks.

My solution is somewhat ad hoc. Effectively we second-guess the mistakes of a rough implementation and then try to correct it after the event. This has the flavor of a hack, and despite being easy to implement and suitable for adding in friction (which we'll do in chapter 15), it is frowned on by engineering purists.

A more physically realistic approach would be to recognize that in reality a force would be applied on the particle from the ground. This reaction force pushes the object back so that its total acceleration in the vertical direction becomes zero. No matter how hard the particle pushes down, the ground will push up with the same force. We can create a force generator that works in this way, making sure there can be no acceleration into the ground.

This works okay for particles that can have only one contact with the ground. For more complex rigid bodies the situation becomes considerably more complex. We may have several points of contact between an object and the ground (or worse, we might have a whole series of contacts between an object and immovable resting points). It isn't immediately clear how to calculate the reaction forces at each contact so that the overall motion of the object is correct. We'll return to reaction forces in some depth in chapter 15, and to more complex resolution methods in chapter 18.

7.3 THE CONTACT RESOLVER ALGORITHM

The collision resolver receives a list of contacts from the collision detection system and needs to update the objects being simulated to take account of the contacts. We have three bits of code for performing this update:

1. The collision resolution function that applies impulses to objects to simulate their bouncing apart.

2. The interpenetration resolution function that moves objects apart so that they aren't partially embedded in one another.

3. The resting contact code that sits inside the collision resolution function and keeps an eye out for contacts that might be resting rather than colliding.

Which of these functions needs calling for a contact depends on its separating velocity and interpenetration depth. Interpenetration resolution only needs to occur if the contact has a penetration depth greater than zero. Similarly we might need to perform interpenetration resolution only, with no collision resolution, if the objects are interpenetrated but separating.

Regardless of the combination of functions needed, each contact is resolved one at a time. This is a simplification of the real world. In reality each contact would occur at a slightly different instant of time, or be spaced out over a range of time. Some contacts would apply their effects in series; others would combine and act simultaneously on the objects they affect. Some physics engines will try to accurately replicate this, treating sequential contacts in their correct order and resolving resting contacts all at the same time. In section 7.3.2, we'll look at an alternative resolution scheme that honors sequential series. In chapter 18 we'll look at systems to perform simultaneous resolution of multiple contacts.

For our engine we'd like to keep things simple and do neither. We'd like to resolve all the contacts one at a time at the end of a frame. We can still get very believable results with this scheme, with a considerably less complex and error-prone implementation. To get the best results, however, we need to make sure the contacts are resolved in the right order.

7.3.1 Resolution Order

If an object has two simultaneous contacts, as shown in figure 7.5, then changing its velocity to resolve one contact may change its separating velocity at the other contact. In the figure, if we resolve the first contact, then the second contact stops being a collision at all: it is now separating. If we resolve the second contact only, however, the first contact still needs to be resolved: the change in velocity isn't enough to save it.

To avoid doing unnecessary work in situations like this, we resolve the most severe contact first: the contact with the lowest separating velocity (i.e., the most negative). As well as convenient, this is also the most physically realistic thing we can do. In the figure, if we compare the behavior of the full three-object situation with the behavior we'd have if we removed one of the two lower blocks, we would find the final result to be most similar to the case where we have block A but not block B. In other words, the most severe collisions tend to dominate the behavior of the simulation. If we have to prioritize which collisions to handle, it should be those that give the most realism.

Figure 7.5 illustrates a complication in our contact resolution algorithm. If we handle one collision, then we might change the separating velocity for other contacts. We can't just sort the contacts by their separating velocity and then handle them in
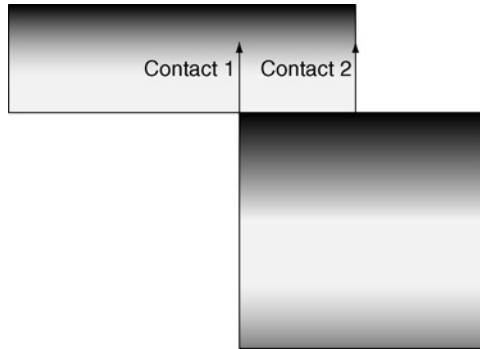
FIGURE 7.5   Resolving one contact may resolve another automatically.

order. Once we have handled the first collision, the next contact may have a positive separating velocity and not need any processing.

There is also another, more subtle problem that doesn't tend to arise in many particle situations. We could have a situation where we resolve one contact and then another, but the resolution of the second puts the first contact back into collision, so we need to re-resolve it. Fortunately it can be shown that for certain types of simulation (particularly those with no friction, although some friction situations can also work), this looping will eventually settle into a correct answer. We'll not need to loop around forever, and we'll not end up with a situation where the corrections get bigger and bigger until the whole simulation explodes. Unfortunately this could take a long time to reach, and there is no accurate way to estimate how long it will take. To avoid getting stuck we place a limit on the number of resolutions that can be performed at each frame.

The contact resolver we will use follows this algorithm:

1. Calculate the separating velocity of each contact, keeping track of the contact with the lowest (i.e., most negative) value.

2. If the lowest separating velocity is greater than or equal to zero, then we're done: exit the algorithm.

3. Process the collision response algorithm for the contact with the lowest separating velocity.

4. If we have more iterations, then return to step 1.

The algorithm will automatically reexamine contacts that it has previously resolved, and it will ignore contacts that are separating. It resolves the most severe collision at each iteration.

The number of iterations should be at least the number of contacts (to give them all a chance of getting seen to at least once) and can be greater. For simple particle

simulations, having the same number of iterations as there are contacts can often work fine. I tend to use double the number of contacts as a rule of thumb, but more is needed for complex, interconnected sets of contacts. You could also give the algorithm no iteration limit and see how it performs. This is a good approach to debugging when difficult situations arise.

You may have noticed that I've ignored interpenetration so far. We could combine interpenetration resolution with collision resolution. A better solution, in practice, is to separate the two into distinct phases. First we resolve the collisions, in order, using the previous algorithm. Second we resolve the interpenetrations.

Separating the two resolution steps allows us to use a different order for resolving interpenetration than for velocity. Once again we want to get the most realistic results. We can do this by resolving the contacts in order of severity, as before. If we combine the two stages, we're tied to a suboptimal order for one or the other kind of resolution.

The interpenetration resolution follows the same algorithm as for collision resolution. As before, we need to recalculate all the interpenetration depths between each iteration. Recall that interpenetration depths are provided by the collision detector. We don't want to perform collision detection again after each iteration, as it is far too time consuming. To update the interpenetration depth we keep track of how much we moved the two objects at the previous iteration. The objects in each contact are then examined. If either object was moved in the last frame, then its interpenetration depth is updated by finding the component of the move in the direction of the contact normal.

Putting this all together we get the contact resolver function:

Excerpt from `include/cyclone/pcontacts.h`

```
/**
 * The contact resolution routine for particle contacts. One
 * resolver instance can be shared for the whole simulation.
 */
class ParticleContactResolver
{
protected:
    /**
     * Holds the number of iterations allowed.
     */
    unsigned iterations;

    /**
     * This is a performance tracking value - we keep a record
     * of the actual number of iterations used.
     */
    unsigned iterationsUsed;

public:
```

```
    /**
     * Creates a new contact resolver.
     */
    ParticleContactResolver(unsigned iterations);

    /**
     * Sets the number of iterations that can be used.
     */
    void setIterations(unsigned iterations);

    /**
     * Resolves a set of particle contacts for both penetration
     * and velocity.
     */
    void resolveContacts(ParticleContact *contactArray,
        unsigned numContacts,
        real duration);
};
```

———— Excerpt from src/pcontacts.cpp ————

```
void
ParticleContactResolver::resolveContacts(ParticleContact *contactArray,
                                         unsigned numContacts,
                                         real duration)
{
    iterationsUsed = 0;
    while(iterationsUsed < iterations)
    {
        // Find the contact with the largest closing velocity;
        real max = 0;
        unsigned maxIndex = numContacts;
        for (unsigned i = 0; i < numContacts; i++)
        {
            real sepVel = contactArray[i].calculateSeparatingVelocity();
            if (sepVel < max)
            {
                max = sepVel;
                maxIndex = i;
            }
        }

        // Resolve this contact.
        contactArray[maxIndex].resolve(duration);
```

```
            iterationsUsed++;
        }
    }
}
```

The number of iterations we use to resolve interpenetration might not necessarily be the same as the number used in resolving collisions. We could implement the function to use a different limit in each case.

In practice there is rarely any need to have different values: we can pass the same for both. As a simulation gets more complex, with interacting objects, the number of collision iterations needed will increase at roughly the same rate as the number of interpenetration iterations. In the preceding function I've used one iteration limit for both parts.

The recalculation of the closing velocity and interpenetration depth at each iteration is the most time consuming part of this algorithm. For very large numbers of contacts this can dominate the execution speed of the physics engine. In practice most of the updates will have no effect: one contact may have no possible effect on another contact. In chapter 16 we'll return to this issue and optimize the way collisions are resolved.

7.3.2 Time-Division Engines

There is another approach to creating a physics engine that avoids having to resolve interpenetration or generate a sensible resolution order for the contacts. Rather than have one single update of the physics engine per frame, we could have many updates, punctuated by collisions.

The theory goes like this:

- When there are no collisions, objects are moving around freely, using just the laws of motion and force generators we saw in the last chapter.
- When a collision occurs, it is at the exact point that two objects touch. At this stage there is no interpenetration.
- If we can detect exactly when a collision occurs, we can use the normal laws of motion up to this point, then stop, then perform the impulse calculations, and then start up with the normal laws of motion again.
- If there are numerous collisions, we process them in order; and between each collision, we update the world using the normal laws of motion.

In practice this kind of engine has this algorithm:

1. Let the start time be the current simulation time and the end time be the end of the current update request.

2. Perform a complete update for the whole time interval.

3. Run the collision detector and collect a list of collisions.

4. If there are no collisions, we are done: exit the algorithm.

5. For each collision work out the exact time of the first collision.

6. Choose the first collision to have occurred.

7. If the first collision occurs after the end time, then we're done: exit the algorithm.

8. Remove the complete update from step 2, and perform an update from the start time to the first collision time.

9. Process the collision, applying the appropriate impulses (no interpenetration resolution is needed, because at the instant of collision the objects are only just touching).

10. Set the start time to be the first collision time, keep the end time unchanged, and return to step 1.

This gives an accurate result and avoids the problems with interpenetration resolution. It is a commonly used algorithm in engineering physics applications where accuracy is critical. Unfortunately it is very time consuming.

For each collision we run the collision detector again and rerun the regular physics update each time. We still need to have special-case code to cope with resting contacts; otherwise, the resting contacts will be returned as the first collision at every iteration. Even without resting contacts, numerical errors in the collision detection calculations can cause a never-ending cycle—a constant stream of collisions occurring at the same time, which causes the algorithm to loop endlessly.

For almost all game projects this approach isn't practical. A once-per-frame update is a better solution, where all the contacts are resolved for velocity and interpenetration.

The "almost" case I am thinking of is pool, snooker, or billiards games. In these cases the sequence of collisions and the position of balls when they collide is critical. A pool game using once-per-frame physics might be believable when two balls collide, but strange effects can appear when the cue ball hits a tightly packed (but not touching) bunch of balls. For a serious simulation it is almost essential to follow the preceding algorithm, with the advantage that if you are writing from scratch, it is easier to implement without the interpenetration code (not to mention the simplifications you can get because all the balls have the same mass).

You can see this in pool simulation games running on older PCs. When you break off, there is a fraction of a second pause when the cue ball hits the pack, as the thousands of internal collisions are detected and handled sequentially.

For a simple arcade pool game, if you already have a once-per-frame physics engine available, it is worth a try: it may be good enough to do the job.

7.4 Collisionlike Things

Just as for springs, we will look at several types of connections that can be modeled using the techniques in this chapter.

You can think of a collision as acting to keep two objects at least some minimum distance apart. A contact is generated between two objects if they ever get too close. By the same token, we can use contacts to keep objects together.

7.4.1 CABLES

A cable is a constraint that forces two objects to be no more than its length apart. If we have two objects connected by a light cable, they will feel no effects as long as they are close together. When the cable is pulled taut, the objects cannot separate further. Depending on the characteristics of the cable, the objects may appear to bounce off this limit, in the same way that objects colliding might bounce apart. The cable has a characteristic coefficient of restitution that controls this bounce effect.

We can model cables by generating contacts whenever the ends of the cable separate too far. The contact is very much like those used for collisions, except that its contact normal is reversed: it pulls the objects together rather than bouncing them apart. The interpenetration depth of the contact corresponds to how far the cable has been stretched beyond its limit.

We can implement a contact generator for a cable in this way:

```
/**
 * Links connect two particles together, generating a contact if
 * they violate the constraints of their link. It is used as a
 * base class for cables and rods, and could be used as a base
 * class for springs with a limit to their extension.
 */
class ParticleLink
{
public:
    /**
     * Holds the pair of particles that are connected by this link.
     */
    Particle* particle[2];

protected:
    /**
     * Returns the current length of the cable.
     */
    real currentLength() const;

public:
    /**
     * Fills the given contact structure with the contact needed
     * to keep the link from violating its constraint. The contact
     * pointer should point to the first available contact in a
     * contact array, where limit is the maximum number of
     * contacts in the array that can be written to. The method
     * returns the number of contacts that have been written. This
     * format is common to contact-generating functions, but this
```

— Excerpt from include/cyclone/plinks.h —

```
      * class can only generate a single contact, so the
      * pointer can be a pointer to a single element. The limit
      * parameter is assumed to be at least one (zero isn't valid),
      * and the return value is either 0, if the cable wasn't
      * overextended, or one if a contact was needed.
      */
    virtual unsigned fillContact(ParticleContact *contact,
                                 unsigned limit) const = 0;
};


/**
 * Cables link a pair of particles, generating a contact if they
 * stray too far apart.
 */
class ParticleCable : public ParticleLink
{
public:
    /**
     * Holds the maximum length of the cable.
     */
    real maxLength;

    /**
     * Holds the restitution (bounciness) of the cable.
     */
    real restitution;

public:
    /**
     * Fills the given contact structure with the contact needed
     * to keep the cable from overextending.
     */
    virtual unsigned fillContact(ParticleContact *contact,
                                 unsigned limit) const;
};
```

───────────────── **Excerpt from src/plinks.cpp** ─────────────────

```
real ParticleLink::currentLength() const
{
    Vector3 relativePos = particle[0]->getPosition() -
                          particle[1]->getPosition();
    return relativePos.magnitude();
}
```

```
unsigned ParticleCable::fillContact(ParticleContact *contact,
                                    unsigned limit) const
{
    // Find the length of the cable.
    real length = currentLength();

    // Check whether we're overextended.
    if (length < maxLength)
    {
        return 0;
    }

    // Otherwise return the contact.
    contact->particle[0] = particle[0];
    contact->particle[1] = particle[1];

    // Calculate the normal.
    Vector3 normal = particle[1]->getPosition() - particle[0]
        ->getPosition();
    normal.normalize();
    contact->contactNormal = normal;

    contact->penetration = length-maxLength;
    contact->restitution = restitution;

    return 1;
}
```

This code acts as a collision detector: it examines the current state of the cable and can return a contact if the cable has reached its limit. This contact should then be added to all the others generated by the collision detector and processed in the normal contact resolver algorithm.

7.4.2 Rods

Rods combine the behaviors of cables and collisions. Two objects linked by a rod can neither separate nor get closer together. They are kept at a fixed distance apart.

We can implement this in the same way as the cable contact generator. At each frame we look at the current state of the rod and generate either a contact to bring the ends inward or a contact to keep them apart.

We need to make two modifications to what we've seen so far, however. First we should always use a zero coefficient of restitution. It doesn't make sense for the two ends to bounce either together or apart. They should be kept at a constant distance from each other, so their relative velocity along the line between them should be zero.

Second, if we apply just one of the two contacts (to separate or to close) at each frame, we will end up with a vibrating rod. On successive frames the rod is likely to be too short, and then too long, and each contact will drag it backward and forward. To avoid this we generate both contacts at every frame. If either of the contacts is not needed (i.e., the separating velocity is greater than zero, or there is no interpenetration), then it will be ignored. Having the extra contact there helps to keep the contact resolver algorithm from overcompensating, so the rod will be more stable. The downside of this approach is that for complex assemblies of rods the number of iterations needed to reach a really stable solution can rise dramatically. If you have a low iteration limit, the vibration can return.

We can implement our contact generator in this way:

Excerpt from include/cyclone/plinks.h

```cpp
/**
 * Rods link a pair of particles, generating a contact if they
 * stray too far apart or too close.
 */
class ParticleRod : public ParticleLink
{
public:
    /**
     * Holds the length of the rod.
     */
    real length;

public:
    /**
     * Returns the current length of the cable.
     */
    real currentLength() const;

    /**
     * Fills the given contact structure with the contact needed
     * to keep the rod from extending or compressing.
     */
    virtual unsigned fillContact(ParticleContact *contact,
                                 unsigned limit) const;
};
```

Excerpt from src/plinks.cpp

```cpp
unsigned ParticleRod::fillContact(ParticleContact *contact,
                                  unsigned limit) const
{
    // Find the length of the rod.
    real currentLen = currentLength();
```

```
    // Check whether we're overextended.
    if (currentLen == length)
    {
        return 0;
    }

    // Otherwise return the contact.
    contact->particle[0] = particle[0];
    contact->particle[1] = particle[1];

    // Calculate the normal.
    Vector3 normal = particle[1]->getPosition() - particle[0]
        ->getPosition();
    normal.normalize();

    // The contact normal depends on whether we're extending
    // or compressing.
    if (currentLen > length) {
        contact->contactNormal = normal;
        contact->penetration = currentLen - length;
    } else {
        contact->contactNormal = normal * -1;
        contact->penetration = length - currentLen;
    }

    // Always use zero restitution (no bounciness).
    contact->restitution = 0;

    return 1;
}
```

The code always generates two contacts, which should be added to the list returned by the collision detector and passed to the contact resolver.

7.5 Summary

We've now built a set of physics code that can connect particles together using both hard constraints, such as rods and cables, and elastic constraints, such as springs and bungees.

Rods and cables behave similarly to collisions between separate objects. Cables can cause the particles joined together to bounce toward one another, just in the same way that particles bounce off one another when they collide. In the same way rods

cause connected particles to stay together, moving with a fixed separation distance. This is equivalent to collisions with no bounce—when the particles stick together and their closing velocity is reduced to zero.

Supporting both hard and elastic connections between particles allows us to build interesting structures and simulate them in the game.

This forms our second complete physics engine: the mass-aggregate engine. Unlike the particle engine we built first, the mass-aggregate engine is rare in published games. It has been used to good effect in many two-dimensional platform games. While it has largely been superseded by the more complex engines described later in this book, it is still useful in some games in its own right. Chapter 8 looks at its strengths and some applications.

This page intentionally left blank
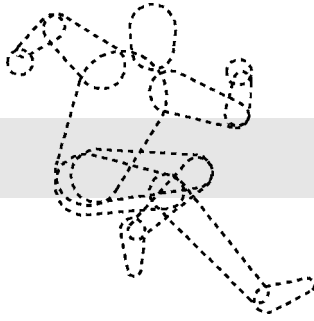
8

THE MASS-AGGREGATE PHYSICS ENGINE

We've now built a mass-aggregate physics engine, capable of both particle simulation and constructions made of many objects connected by rods, cables, and springs. It is time to test the engine on some example scenarios.

The engine still has limits, however; in particular, it can't describe the way objects rotate. We'll look at ways around this, faking the rotation of objects in terms of mass aggregates. It is a useful technique for some applications and can eliminate the need for more advanced physics.

8.1 OVERVIEW OF THE ENGINE

The mass-aggregate physics engine has three components:

1. The particles themselves keep track of their position, movement, and mass. To set up a simulation we need to work out what particles are needed and set their initial position velocity. We also need to set their inverse mass. The acceleration of an object due to gravity is also held in the rigid body (this could be removed and replaced by a force, if you so desire).

2. The force generators are used to keep track of forces that apply over several frames of the game.

3. The collision system accumulates a set of contact objects and passes them to the contact resolver. Any bit of code can generate new contacts. We have considered two: a collision detector and rod or cable constraints.

At each frame we take each particle, calculate its internal data, call its force generators, and then call its integrator to update its position and velocity. We then accumulate the contacts on the particle and pass all the contacts for all the particles into the collision resolver.

To make this process easier we will construct a simple structure to hold any number of rigid bodies. We hold the rigid bodies in a linked list, exactly as we did for force generators. This linked list is contained in a World class, representing the whole physically simulated world:

Excerpt from include/cyclone/pworld.h

```cpp
/**
 * Keeps track of a set of particles, and provides the means to
 * update them all.
 */
class ParticleWorld
{

/**
 * Holds one particle in the linked list of particles.
 */
struct ParticleRegistration
{
    Particle *particle;
    ParticleRegistration *next;
};

/**
 * Holds the list of registrations.
 */
ParticleRegistration* firstParticle;
public:

/**
 * Creates a new particle simulator that can handle up to the
 * given number of contacts per frame. You can also optionally
 * give a number of contact-resolution iterations to use. If you
 * don't give a number of iterations, then twice the number of
 * contacts will be used.
 */
```

```
            ParticleWorld(unsigned maxContacts, unsigned iterations=0);
    };
```

At each frame the startFrame method is first called, which sets up each object ready for the force accumulation:

```
                  Excerpt from include/cyclone/pworld.h
    /**
     * Keeps track of a set of particles, and provides the means to
     * update them all.
     */
    class ParticleWorld
    {
    /**
     * Initializes the world for a simulation frame. This clears
     * the force accumulators for particles in the world. After
     * calling this, the particles can have their forces for this
     * frame added.
     */
    void startFrame();
    };
```

```
                     Excerpt from src/pworld.cpp
void ParticleWorld::startFrame()
{
    ParticleRegistration *reg = firstParticle;
    while (reg)
    {
        // Remove all forces from the accumulator.
        reg->particle->clearAccumulator();

        // Get the next registration.
        reg = reg->next;
    }
}
```

Additional forces can be applied after calling this method.

We will also create another system to register contacts. Just as we saw for force generators, we create a polymorphic interface for contact detectors.

```
                 Excerpt from include/cyclone/pcontacts.h
/**
 * This is the basic polymorphic interface for contact generators
 * applying to particles.
```

```
 */
class ParticleContactGenerator
{
public:
    /**
     * Fills the given contact structure with the generated
     * contact. The contact pointer should point to the first
     * available contact in a contact array, where limit is the
     * maximum number of contacts in the array that can be written
     * to. The method returns the number of contacts that have
     * been written.
     */
    virtual unsigned addContact(ParticleContact *contact,
                                unsigned limit) const = 0;
};
```

Each of these gets called in turn from the world and can contribute any contacts it finds back to the world by calling its addContact method.

To execute the physics, the runPhysics method is called. This calls all the force generators to apply their forces and then performs the integration of all objects, runs the contact detectors, and resolves the resulting contact list:

```
─────── Excerpt from include/cyclone/pworld.h ───────
/**
 * Keeps track of a set of particles, and provides the means to
 * update them all.
 */
class ParticleWorld
{
    /// ... previous ParticleWorld code as before ...

    /**
     * Holds the force generators for the particles in this world.
     */
    ParticleForceRegistry registry;

    /**
     * Holds the resolver for contacts.
     */
    ParticleContactResolver resolver;

    /**
     * Holds one registered contact generator.
     */
```

```
struct ContactGenRegistration
{
    ParticleContactGenerator *gen;
    ContactGenRegistration *next;
};

/**
 * Holds the list of contact generators.
 */
ContactGenRegistration *firstContactGen;

/**
 * Holds the list of contacts.
 */
ParticleContact *contacts;

/**
 * Holds the maximum number of contacts allowed (i.e., the
 * size of the contacts array).
 */
unsigned maxContacts;

public:
/**
 * Calls each of the registered contact generators to report
 * their contacts. Returns the number of generated contacts.
 */
unsigned generateContacts();

/**
 * Integrates all the particles in this world forward in time
 * by the given duration.
 */
void integrate(real duration);

/**
 * Processes all the physics for the particle world.
 */
void runPhysics(real duration);
};
```

_____ **Excerpt from src/pworld.cpp** _____

```
unsigned ParticleWorld::generateContacts()
{
```

```
    unsigned limit = maxContacts;
    ParticleContact *nextContact = contacts;

    ContactGenRegistration * reg = firstContactGen;
    while (reg)
    {
        unsigned used = reg->gen->addContact(nextContact, limit);
        limit -= used;
        nextContact += used;

        // We've run out of contacts to fill. This means we're missing
        // contacts.
        if (limit <= 0) break;

        reg = reg->next;
    }

    // Return the number of contacts used.
    return maxContacts - limit;
}

void ParticleWorld::integrate(real duration)
{
    ParticleRegistration *reg = firstParticle;
    while (reg)
    {
        // Remove all forces from the accumulator.
        reg->particle->integrate(duration);

        // Get the next registration.
        reg = reg->next;
    }
}

void ParticleWorld::runPhysics(real duration)
{
    // First apply the force generators.
    registry.updateForces(duration);

    // Then integrate the objects.
    integrate(duration);

    // Generate contacts.
    unsigned usedContacts = generateContacts();
```

```
    // And process them.
    if (calculateIterations) resolver.setIterations(usedContacts * 2);
    resolver.resolveContacts(contacts, usedContacts, duration);
}
```

We add a call to startFrame at the start of each frame of the game, and a call to runPhysics wherever we want the physics to occur. A typical game loop might look like this:

```
void loop()
{
  while (1) {
    // Prepare the objects for this frame.
    world.startFrame();

    // Calls to other parts of the game code.
    runGraphicsUpdate();
    updateCharacters();

    // Update the physics.
    world.runPhysics();

    if (gameOver) break;
  }
}
```

8.2 Using the Physics Engine

We will look at a useful application of the mass-aggregate engine: creating structures out of particle masses and hard constraints. Using this technique we can create and simulate many larger objects. The possibilities are endless: crates; mechanical devices; even chains and vehicles; or, with the addition of springs, soft deformable blobs.

8.2.1 Rope-Bridges and Cables

Sagging bridges, cables, and tilting platforms are all stalwarts of the platform game genre, as well as having applications in other genres.

We can set up a bridge using pairs of particles suspended by cables. Figure 8.1 shows an arrangement that has this effect. Each pair of particles along the bridge is linked with a rod constraint to keep them connected with their neighbors. Pairs of
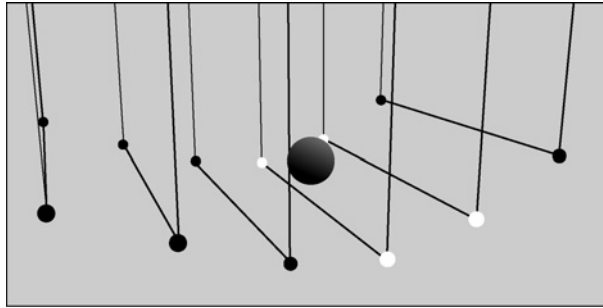
FIGURE 8.1 Screenshot of the **bridge** demo.

particles are likewise linked together to give the bridge some strength. The cables are cable constraints descending from a fixed point in space.

On the CD the **bridge** demo shows this setup in operation. You can move an object (representing a character) over the bridge. The collision detector applies contacts to the nearest particles when the object is above them. Notice the bridge stretch and conform to the presence of the heavy object. In the demo the constraints are shown as lines in the simulation.

The collision detector needs some explanation. Because we have only particles in our simulation, but we want to give the impression of a bridge, it is not the collision between particles that interests us but the collision between the character and the planks of the bridge. We will return later in the book to a more robust way of doing this. For the purpose of this chapter I have created a custom collision detector.

The detector treats the character as a sphere and checks to see whether it intersects with any of the planks. A *plank* is a line segment between one pair of particles. If the object does intersect, then a contact is generated between the character object and the nearest of the plank particles. The contact normal is set based on the position of the object and the line of the plank.

Tilting platforms can use the same theory. Figure 8.2 shows a suitable structure. On the CD the **platform** demo shows this in operation: the platform will, by default, tilt in one direction. A weight can be added to the opposite end, causing it to tilt. The particles that make up the pivot of the platform have been set with infinite mass to avoid their moving. If the platform were intended to be mobile, they could be set with a mass similar to the other particles.

The simulation setup is similar to the **bridge** demo; you can see the full source code for both on the CD.

8.2.2 FRICTION

One key limitation of this approach is the lack of friction in our contact model. It was a deliberate choice to leave out friction at this stage: we'll implement it as part of
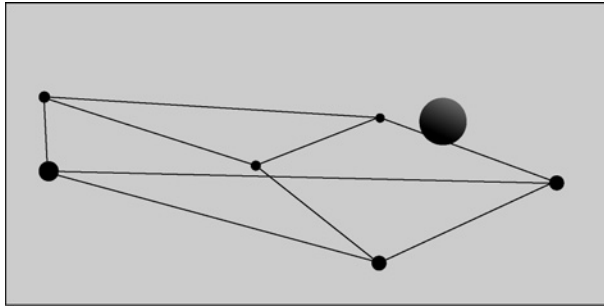
FIGURE 8.2   Screenshot of the **platform** demo.

the engine in part V. If you create mass aggregates, they will appear to slide over the ground as if skating on ice. Try replacing the infinite masses of the **platform** demo and see the platform slide about.

If you are intending to implement only a mass-aggregate physics system, then it is worth skipping forward to chapter 15. The discussion of friction there can be easily adapted for particle contacts. In fact the mathematics is a little simpler: we can ignore all the rotational components of the contact.

For anything but the simplest assemblies of particle masses, it may be worth implementing the full physics engine in any case. You can create any object with a mass-aggregate system, but as the number of constraints increases, so does the burden on the collision response system and the tendency for stiff constraints to flex slightly as groups of hard constraints compete to be resolved. A full rigid-body solution is the most practical for general physics simulation. It's time to bite the bullet and move from particles to complete rotating, extended objects.

8.2.3   B LOB G AMES

Recently there have been a couple of games with soft-bodied characters simulated in a way that is easy to replicate with our engine.

The independent Gish and the hit PSP game Loco Roco use 2D characters made up of a set of particles (in the case of Loco Roco you get more particles as you play; in the case of Gish there appear to be three or four at all times). These particles are connected together using soft springs, so they can move a reasonable distance apart. To avoid moving too far apart the springs have a limit of elasticity, beyond which they act as rods and cannot be further extended (you could use this limit to split the blob into smaller blobs).

The difficult part of using this setup is to then render the whole character as the agglomeration of blobs. In 2D this can be done by superimposing a circle on each particle, and making sure the springs don't allow the circles to separate from one another, giving the impression of a soft blob. In both 2D and 3D you could also use a

metaball implementation, seen in many 3D modeling packages. This is quite a complex algorithm, but for a small number of masses it's easily tractable. The metaball algorithm isn't simple, and I won't cover it here; you can see any good textbook on modeling for an explanation of how metaballs work.

The **blob** demo on the CD gives a simple implementation of a Loco Roco-style blob game.

8.3 Summary

While slightly cumbersome, a mass-aggregate physics engine is capable of simulating some interesting and complex effects. Sets of relatively simple objects, joined by a mixture of hard and elastic constraints, are particularly suited to this approach.

The first example we saw, rope-bridges, have been simulated with a mass-aggregate approach for many years. The second example showed how to build large objects out of a set of particles. While this can work successfully, it is prone to many problems. Objects made up of lots of particles and lots of hard constraints can be slightly unstable; they can appear to flex and bend when simulated, and in the worst case there can be noticeable vibration in the particles as their constraints pull them in different ways.

There is a better way to simulate a single large object. Rather than build it out of particles, we can treat it as a whole. To do this we'll need to change our physics engine dramatically, however. As well as just simulating the position, velocity, and acceleration of an object, we'll need to take into account how it rotates as it moves. This will introduce a large amount of complexity into our physics engine and will take us the rest of this book to implement properly. Chapter 9 takes the first step, introducing the mathematics of rotation.

Rigid-Body Physics

This page intentionally left blank

9

The Mathematics of Rotations

So far we have covered almost all there is to know when creating a physics engine. We have built a sophisticated system capable of simulating particles, either individually or connected into aggregates.

We are missing two things:

- A robust general-purpose collision detection system (currently we're using quite an ad hoc system of hard constraints).
- The ability of objects to rotate as well as move around.

The first of these problems is fairly easy to resolve and is the subject of part IV of this book.

The second is more complex: it is the difference between a complete rigid-body physics system and the mass-aggregate systems we've seen so far. To add rotations we'll need to go backward in the capability of our engine. We'll need to remove a good deal of functionality and rebuild it based on full rotating rigid bodies. This will take this part and part V—almost the rest of the book.

This chapter looks at the properties of rotating bodies and the mathematical structures needed to represent and manipulate them.

9.1 Rotating Objects in Two Dimensions

Before we look at rotations in three dimensions, it is worth understanding them in two. I will not implement any code from this section, but thinking about the two-dimensional case is a good analogy for understanding three dimensions.
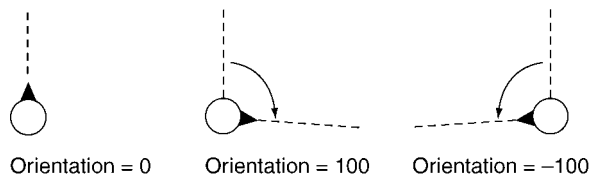
145

Orientation = 0   Orientation = 100   Orientation = −100

FIGURE 9.1   The angle that an object is facing.

In two dimensions we can represent an object by its two-dimensional position and an angle that shows how it is oriented. Just as the position is specified relative to some fixed origin point, the angle is also given relative to a predetermined direction. Figure 9.1 illustrates this.

If the object is rotating, its orientation will change over time. Just as velocity is the first derivative of position (see chapter 2), angular velocity is the first derivative of orientation.

I will use the word *orientation* throughout this book to refer to the direction in which an object is facing. The word *rotation* has many meanings in different contexts, and while most people feel they know what it means, it is one of those terms that can be a chameleon, causing subtle confusion.

To be specific, I'll try to use *rotation* only to mean a change in orientation (the exception being that when everybody and their dog calls something "rotation," I'll avoid the temptation to make up a new name). If something is rotated, it is natural to mean that its orientation has changed.

If an object is spinning, however, I'll use the term *angular velocity* to mean the rate of change of orientation.

9.1.1   The Mathematics of Angles

If we do any mathematics with orientations, we need to be careful: many different orientation values can represent the same orientation. If we measure orientation in radians (there are 2π radians in the 360° of a circle), then the orientation of 2π is the same as 0. Developers normally set a fixed range of orientation values, say $(-\pi, \pi]$ (the square bracket indicates that π is included in the range, and the curved bracket that $-\pi$ is not). If an orientation falls outside this range, it is brought back into the range. The mathematical routines that deal with this kind of orientation scalar can look messy, with lots of adjustments and checks.

An alternative approach is to use vectors to represent orientation. We take a two-element vector representing the direction in which the object is pointing. The vector is related to the scalar value according to the equation

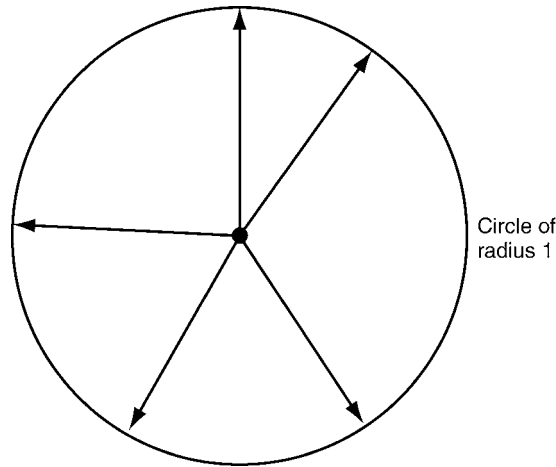$$\boldsymbol{\theta} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \tag{9.1}$$

Circle of
radius 1

FIGURE 9.2   The circle of orientation vectors.

where θ is the angular representation of orientation and $\boldsymbol{\theta}$ is the vector representation. I have assumed that zero orientation would see the object facing along the positive X axis, and that orientation increases in the counterclockwise direction. This is simply a matter of convention.

The vector form of orientation makes many (but not all) mathematical operations easier to perform, with less special-case code and bounds-checking.

In moving to a two-dimensional representation we have doubled the number of values representing our orientation. We have only one degree of freedom when deciding which direction an object should face, but the representation of a vector has two degrees of freedom. A *degree of freedom* is some quantity that we could change independent of others. A 3D position has three degrees of freedom, for example, because we can move it in any of three directions without altering its position in the other two. Calculating the number of degrees of freedom is an important tool for understanding rotations in 3D.

Having this extra degree of freedom means that we could end up with a vector that doesn't represent an orientation. In fact most vectors will not match equation 9.1. To guarantee that our vector represents an orientation, we need to remove some of its freedom. We do this by forcing the vector to have a magnitude of 1. Any vector with a magnitude of 1 will match equation 9.1, and we'll be able to find its corresponding angle.

There's a geometric way of looking at this constraint. If we draw a point at the end of all possible vectors with a magnitude of 1, we get a circle, as shown in figure 9.2. We could say that a vector orientation correctly represents an orientation if it lies on this circle. If we find a vector that is supposed to represent an orientation but is slightly off (because of numerical errors in some calculation), we can fix it by bringing it onto

the circle. Mathematically we do this by forcing its magnitude to be 1, by normalizing the vector.

If we built a 2D game using vectors to represent orientations, we'd need to occasionally make sure that the orientations still lie on the circle by normalizing them.

Let's summarize these steps (not surprisingly we'll see them again later). We started with problems of bounds-checking, which led us to use a representation with one more degree of freedom that needed an extra constraint; in turn this led us to add in an extra step to enforce the constraint.

9.1.2 Angular Speed

When we look at the angular speed of an object (sometimes called its "rotation"), we don't have any of the problems we saw for orientation. An angular speed of 4π radians per second is different from 2π radians per second. Every angular speed, expressed as a single scalar value, is unique. The mathematics for angular speed is simple, so we don't need bounds-checking and special-case code. This in turn means we don't need to use a vector representation and we can stick with our scalar value.

9.1.3 The Origin and the Center of Mass

Before we leave two dimensions, it is worth considering what our position and orientation represent. When we were dealing with particles, the position represented the location of the particle. Particles by definition exist only at a single point in space, even though in this book we've stretched the definition slightly and treated them like small spheres.

The Origin of an Object

If we have a larger object, what does the position represent? The object is at many locations at the same time: it covers some extended area.

The position represents some preagreed location on the object that doesn't change. It is sometimes called the "origin" of the object. In a game we might choose the root of the spine of a character or the center of the chassis of a car. The position doesn't need to be inside the object at all. Many developers represent the position of a character as a location between the character's heels resting on the ground.

As long as the location doesn't move around the object, we can always determine where every bit of the object will be from just its position and orientation. Locations on the object are given *relative* to the origin of the object. If the origin of a car is in the center of its chassis, as shown in figure 9.3, then its right headlight might be at a position of

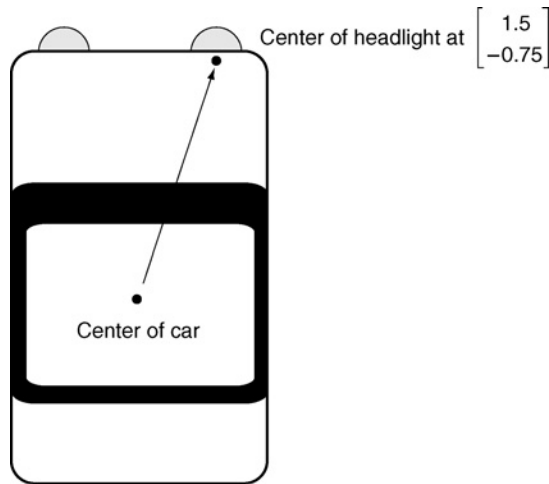$$\begin{bmatrix} 1.5 \\ -0.75 \end{bmatrix}$$

Center of headlight at $\begin{bmatrix} 1.5 \\ -0.75 \end{bmatrix}$

Center of car

FIGURE 9.3   The relative position of a car component.

relative to the origin. If the car is moved so that its origin is at

$$\begin{bmatrix} 4 \\ 3.85 \end{bmatrix}$$

then its headlight will be at

$$\begin{bmatrix} 1.5 \\ -0.75 \end{bmatrix} + \begin{bmatrix} 4 \\ 3.85 \end{bmatrix} = \begin{bmatrix} 5.5 \\ 3.1 \end{bmatrix}$$

This movement is called a "translation": we are translating the car from one position to another.

Rotations

The same thing occurs if the object is facing in a different direction. In figure 9.4 the car has had its position and orientation altered.

So how do we calculate the location of the headlamp now? First we need to turn the headlamp around to represent the direction in which the car is facing. We do this by using a third version of our orientation value.

This time the orientation is expressed in matrix form. If you are unsure about matrices, I'll return to their mathematics when we come to implementing matrix classes for 3D in Section 9.2.3. You can skip the mathematics here unless you need a refresher.
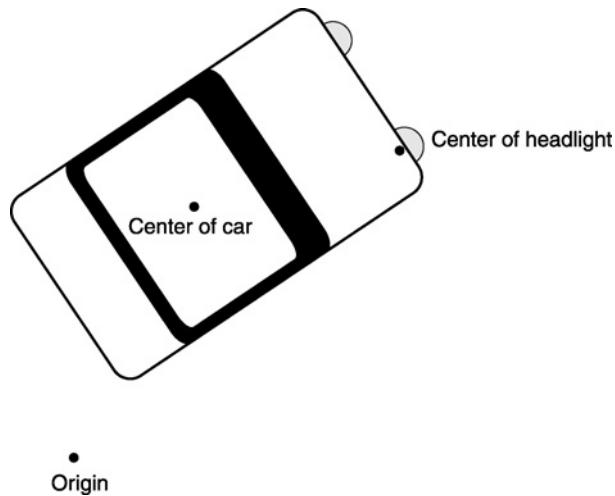
FIGURE 9.4   The car is rotated.

The matrix form of orientation looks like this:

$$\Theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

where θ is the orientation angle, as before. This matrix is usually called the "rotation matrix": it can be used to rotate a location. We can work out the new position of the headlamp by multiplying the *relative* position of the headlamp by the rotation matrix

$$q' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} q_b$$

where q_b is the relative location of the headlamp. In our case, where $\theta = 3\pi/8$, we get

$$q' = \begin{bmatrix} 0.38 & -0.92 \\ 0.92 & 0.38 \end{bmatrix} \begin{bmatrix} 1.5 \\ -0.75 \end{bmatrix} = \begin{bmatrix} 0.57 + 0.69 \\ 1.39 - 0.29 \end{bmatrix} = \begin{bmatrix} 1.27 \\ 1.10 \end{bmatrix}$$

where all values are given to two decimal places.

After applying the orientation in this way we can then apply the change in position as before. The total process looks like this:

$$q = \Theta q_b + p \tag{9.2}$$

where p is the position of the object. This equation works in both 2D and 3D, although the definition of Θ is different, as we'll see later in the chapter.

For our car example we get

$$q = \begin{bmatrix} 0.38 & -0.92 \\ 0.92 & 0.38 \end{bmatrix} \begin{bmatrix} 1.5 \\ -0.75 \end{bmatrix} + \begin{bmatrix} 4 \\ 3.85 \end{bmatrix} = \begin{bmatrix} 1.27 \\ 1.10 \end{bmatrix} + \begin{bmatrix} 4 \\ 3.85 \end{bmatrix} = \begin{bmatrix} 5.27 \\ 4.95 \end{bmatrix}$$

This calculation of the location of part of an object, based on the object's position and orientation and the relative position of the component, is called a "transformation from local space" (also called "body space" and "object space") to world space. We'll return to world space and local space in section 9.4.5.

The Composition of Rotations and Translations

One vital result to notice is that any sequence of translations and rotations can be represented with a single position and orientation. In other words, no matter how many times I move and turn the car, we can always give a single set of values for its current position and orientation. This is equivalent to saying that any combination of rotations and translations is equivalent to a single rotation followed by a single translation.

Rigid Bodies

The fact that all the components of an object are fixed relative to its origin is the reason why we talk about rigid bodies when it comes to physics engines. If our car is a infant's toy made of squashable rubber, then knowing the position and orientation isn't enough to tell us where the headlamp is: the headlamp might have been stretched out into a very different position.

Some physics engines can deal with simple soft bodies, but usually they work by assuming the body is rigid and then applying some after-effects to make it look soft. In our engine, as well as in the vast majority of game physics engines, we will support only rigid bodies.

Theoretically we could choose any point on the object to be its origin. For objects that aren't being physically simulated, this is often the approach developers take: they choose a point that is convenient for the artist or artificial intelligence (AI) programmer to work with. It is possible to create physics code that works with an arbitrary origin, but the code rapidly becomes fiendishly complicated. There is one position on every object where the origin can be set that dramatically simplifies the mathematics: the center of mass.

Center of Mass

The center of mass (often called the "center of gravity") is the balance point of an object. If you divide the object in two by cutting any straight line through this point, you will end up with two objects that have exactly the same weight. If the object is a

two-dimensional shape, you can balance it on your finger by placing your finger at the center of mass.

If you think of an object as being made up of millions of tiny particles (atoms, for example), you can think of the center of mass as being the average position of all these little particles, where each particle contributes to the average depending on its mass. In fact this is how we can calculate the center of mass. We split the object into tiny particles and take the average position of all of them:

$$\boldsymbol{p}_{\text{cofm}} = \frac{1}{m} \sum_n m_i \boldsymbol{p}_i$$

where $\boldsymbol{p}_{\text{cofm}}$ is the position of the center of mass, m is the total mass of the object, m_i is the mass, and $\boldsymbol{p}_i$ is the position of particle i.

The center of mass of a sphere of uniform density will be located at the center point of the sphere. Similarly with a cuboid, the center of mass will be at its geometric center. The center of mass isn't always contained within the object. A donut has its center of mass in the hole, for example. Appendix A gives a breakdown of a range of different geometries and where their center of mass is located.

The center of mass is important because if we watch the center of mass of a rigid body, *it will always behave like a particle*. In other words, we can use exactly the same formulae we have used so far in this book to perform the force calculations and update the position and velocity for the center of mass. By selecting the center of mass as our origin position we can completely separate the calculations for the linear motion of the object (which is the same as for particles) and its angular motion (for which we'll need extra mathematics).

Any physical behavior of the object can be decomposed into the linear motion of the center of mass and the angular motion around the same point. This is a profound and crucial result, but one that takes some time to prove; if you want the background, any good undergraduate textbook on mechanics will give details.

If we choose any other point as the origin, we can no longer separate the two kinds of motion, so we'd need to take into account how the object was rotating in order to work out where the origin is. Obviously this would make all our calculations considerably more complicated.

Some authors and instructors work through code either way (although typically only for a few results; when the mathematics gets really hard, they give up). Personally I think it is a very bad idea to even consider having your origin anywhere else but at the center of mass. I'll assume this will always be the case for the rest of the book; if you want your origin somewhere else, you're on your own!

9.2   Orientation in Three Dimensions

In two dimensions we started out with a single angle for orientation. Problems with keeping this value in bounds led us to look at alternative representations. In many

two-dimensional games a vector representation is useful, but the mathematics for angles alone isn't *so* difficult that you couldn't stick with the angle and adjust the surrounding code to cope.

Not surprisingly there are similar problems in three dimensions, and we will end up with a representation for orientation that is not a common bit of mathematics you might learn in high school. In three dimensions, however, the obvious representation is so fundamentally flawed that it is almost impossible to imagine providing the right workarounds to get them running.

I don't want to get bogged down in representations that *don't* work, but it is worth taking a brief look at the problems before we look at a range of improving solutions.

9.2.1 EULER ANGLES

In three dimensions an object has three degrees of freedom for rotation. By analogy with the movement of aircraft we can call these yaw, pitch, and roll. Any rotation of the aircraft can be made up of a combination of these three maneuvers. Figure 9.5 illustrates them.

For an aircraft these rotations are about the three axes: *pitch* is a rotation about the X axis, *yaw* is about the Y axis, and *roll* is about the Z axis (assuming an aircraft is looking down the Z axis, with the Y axis up).

Recall that a position is represented as a vector, where each component represents the distance from the origin in one direction. We could use a vector to represent rotation, where each component represents the amount of rotation about the corresponding axis. We have a similar situation to our two-dimensional rotation, but here we have three angles, one for each axis. These three angles are called "Euler angles."

This is the most obvious representation of orientation. It has been used in many graphics applications. Several of the leading graphics modeling packages use Euler
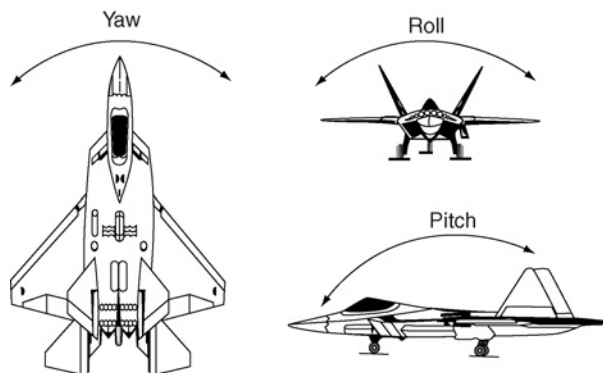

FIGURE 9.5   Aircraft rotation axes.

angles internally, and those that don't still often represent orientations to the user as Euler angles.

Unfortunately Euler angles are almost useless for our needs. We can see this by looking at some of the implications of working with them. You can follow this through by making a set of axes with your hand (as described in section 2.1.1), remembering that your imaginary object is facing in the same direction as your palm—along the Z axis.

Imagine we first perform a pitch, by 30° or so. The object now has its nose up in the air. Now perform a yaw by about the same amount. Notice that the yaw axis is no longer pointing up: when we pitched the object, the yaw axis also moved. Remember where the object is pointing. Now start again, but perform the yaw first, then the pitch. The object will be in a slightly different position. What does this mean? If we have a rotation vector like

$$\begin{bmatrix} 0.3 \\ 0.4 \\ 0.1 \end{bmatrix}$$

in what order do we perform the rotations? The result may be different for each order. What is more, because the order is crucial, we can't simply use regular vector mathematics to combine rotations. In particular,

$$\boldsymbol{r}_1 \cdot \boldsymbol{r}_2 \neq \boldsymbol{r}_2 \cdot \boldsymbol{r}_1$$

where $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are two rotations.

In case you think that the problem is caused by moving the rotation axes around (i.e., keeping them welded to the object rather than fixed in the world), try it the other way. Not only does the same problem still occur, but now we have another issue—gimbal lock.

Gimbal lock occurs when we rotate an object such that what started as one axis now aligns with another. For example, assume we're applying the rotations in the order X, then Y, then Z. If we yaw around by 90° (i.e., no X rotation, 90° Y rotation), the front of the object is now pointing in the negative X direction. Say we wanted to have the object roll slightly now (roll from its own point of view). We can't do that: the axis we need (the local Z axis) is now pointing in the X direction, and we've already passed the point of applying X rotations.

So maybe we should have applied a little bit of X rotation first before rotating in the Y direction. Try it: you can't do it. For this particular problem we could perform the rotations in a different order—ZYX, for example. This would solve the problem for the previous example, but there'd be new orientations that this ordering couldn't represent. Once rotations of around 90° come into play, we can't achieve all desired orientations with a combination of Euler angles. This is called "gimbal lock."

There are some ways to mitigate the problem, by using combinations of axes, some of which move with the object and some of which are fixed in world space. Alternatively we can repeat rotations around some axes. There are a lot of different

schemes, and some of them are more useful than others. All are characterized by very arbitrary mathematics, horrendous boundary conditions, and a tendency to find difficult situations to crash on long after you think they've been debugged.

Gimbal lock was so significant that it featured in NASA's Apollo moon program. The mathematics we'll end up with for orientations was not available, and Euler angles were used in the flight-monitoring computers. To prevent the computers from reaching gimbal lock and finding it impossible to represent the orientation of the spacecraft, restrictions were placed on the way astronauts could exert control. If the craft got too near gimbal lock, a warning would sound. There was no physical reason why the craft couldn't orient in that way; it was purely a precaution to avoid having NASA's computers fall over. Personally I take this as a salutary lesson. If the best minds of NASA can't write software to cope with gimbal lock, I am certainly not going to try.

Fortunately there are much better ways of representing orientation. They may not be as intuitive to visualize, but their mathematics is a lot more reliable.

9.2.2 Axis–Angle

Any rotation, or combination of rotations, in three dimensions can be represented as a single rotation about a fixed axis. In other words, no matter what combination of rotations takes place, we can always specify the orientation of an object as an axis and an angle.

Although this isn't immediately obvious, you can easily verify it for yourself with a small ball. Regardless of how you orient the ball, you can get it into any other orientation by one rotation about a single axis.

We could use this as a representation for orientation (called, not surprisingly, an "axis–angle representation"). It is roughly equivalent to the angle representation we used for two dimensions, and suffers some of the same problems: we need to perform lots of bounds-checking to make sure that the angle is always in the correct range $(-\pi, \pi]$.

Having a vector (for the axis) and an angle gives us four degrees of freedom. The rotation is only three degrees of freedom. The extra degree of freedom is removed by requiring that the vector representing the axis is normalized. It represents only a direction.

Another possible representation using axis and angle is the scaled axis representation. If the axis is normalized, then we can combine the axis and angle into one vector. The direction of the vector gives the axis, and the magnitude of the vector gives the angle. The angle is therefore in the range $[0, \pi)$. We don't need to represent negative angles because they are equivalent to a positive rotation in the opposite direction.

The scaled axis representation is the most compact representation we have. It has three values for three degrees of freedom, and it can represent any orientation. Although it will be useful to us later in this chapter when we come to look at angular velocity, it is almost never used to represent orientations.

This is for the same reasons we avoided a single angle representation for two-dimensional rotations. The mathematics involved in manipulating a scaled axis rep-

resentation of orientation isn't simple. Unlike for the two-dimensional case, we have more than just the bounds to worry about: it isn't clear how to combine rotations easily because the axis as well as the angle need to change.

Until a few years ago the most common way to represent orientations went to the opposite extreme. Rather than use three values, a 3×3 matrix was used.

9.2.3   ROTATION MATRICES

If we were interested in the mathematics of combining rotations, then we could borrow from 3D geometry and represent orientations with a rotation matrix. In games we regularly use matrices to represent rotations. In fact the chances are that whatever representation we use, we'll have to turn it into a rotation matrix and send it to the rendering engine in order to draw the object. Why not save the effort and use the rotation matrix from the start?

Using rotation matrices is a good solution; and we can represent any rotation with a rotation matrix. The elements of the matrix are

$$\Theta = \begin{bmatrix} tx^2 + c & txy + sz & txz - sy \\ txy - sz & ty^2 + c & tyz + sx \\ txz + sy & tyz - sx & tz^2 + x \end{bmatrix} \qquad [9.3]$$

where

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

is the axis; $c = \cos\theta$, $s = \sin\theta$, and $t = 1 - \cos\theta$; and θ is the angle.

Because the elements are related to the sine and cosine of the angle rather than to the angles themselves, we don't have to do any bounds-checking. Combining two rotations is simply a matter of multiplying the two matrices together.

The downside with using rotation matrices is their excess degrees of freedom. We are representing a three-degree-of-freedom system with nine numbers. Floating-point arithmetic in the computer isn't totally accurate. So to make sure the matrix represents a rotation (as opposed to some other kind of transformation such as a skew or even a mirror image) after it has been manipulated in some way, we need to adjust its values periodically. With so many degrees of freedom this adjustment process needs to take place more often than we'd like, and it isn't a trivial process as normalizing a vector is.

Ideally we'd like a representation that has the advantages of matrices: a straightforward combination of rotations, no bounds-checking, and fewer degrees of freedom.

The solution, now almost ubiquitous, is to use a mathematical structure called a "quaternion."

9.2.4 QUATERNIONS

The best and most widely used representation for orientations is the quaternion. A *quaternion* represents an orientation with four values, related to the axis and angle in the following way:

$$\begin{bmatrix} \cos\frac{\theta}{2} \\ x\sin\frac{\theta}{2} \\ y\sin\frac{\theta}{2} \\ z\sin\frac{\theta}{2} \end{bmatrix}$$

[9.4]

where

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

is the axis and θ is the angle, as before.

Quaternions are not merely a four-element vector, however; the mathematics is more exotic. If you are allergic to mathematics, then feel free to skip this explanation and head for the next section.

You may remember in high school mathematics learning about the square root of -1, the so-called imaginary number (in contrast to real numbers), often written as i or j. So $i^2 = -1$. A complex number is then made up of both a real number and some multiple of i, in the form $a + bi$. If your mathematical memory is very good, you might recall drawing complex numbers as coordinates in two dimensions and deriving lots of their properties geometrically. Complex numbers have a very strong connection with geometry and in particular with rotations in two dimensions. If you don't remember, not to worry—quaternions are a little more complex still.

A quaternion is a number of the form $w + xi + yj + yk$, where i, j, and k are all imaginary numbers:

$$i^2 = j^2 = k^2 = -1.$$

When all are multiplied together, we also get -1:

$$ijk = -1$$

Together these are the fundamental formulae of quaternion algebra.[1] The second part of this result means that any two of the three imaginary numbers, when multiplied

---

1. The formulae are reputed to have been scratched in the stone of the Bougham Bridge near Dublin by the discoverer of quaternions, William Rowan Hamilton (the site is now marked by a plaque and the original carving, if it existed, cannot be seen).

together, give us the third. But beware: quaternion mathematics isn't commutative. In other words, $ab \neq ba$, and in particular,

$$ij = -ji = k$$
$$jk = -kj = i$$
$$ki = -ik = j$$

by definition.

With these laws we can combine quaternions by multiplication:

$$(w_1 + x_1 i + y_1 j + z_1 k) \times (w_2 + x_2 i + y_2 j + z_2 k)$$

$$= (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + (w_1 x_2 + x_1 w_2 - y_1 z_2 - z_1 y_2)i$$

$$+ (w_1 y_2 - x_1 z_2 + y_1 w_2 - z_1 x_2)j + (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2)k$$

and if the original two quaternions represent rotations according to equation 9.4, then the resulting quaternion is equivalent to the two rotations combined. I will write quaternions using the format $\hat{\theta}$ and in a four-element vector format to show their components:

$$\hat{\theta} = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

Quaternions have four degrees of freedom to represent the three degrees of freedom of rotation. Clearly we have an extra degree of freedom that we need to constrain away.

In fact, for all rotations equation 9.4 implies that the magnitude of the quaternion is exactly 1. We calculate the magnitude of the quaternion in exactly the same way as we did for a three-element vector, by using a four-component version of Pythagoras's theorem:

$$\sqrt{w^2 + x^2 + y^2 + z^2}$$

To make sure that a quaternion always represents a rotation, we therefore need to make sure that it has unit length:

$$\sqrt{w^2 + x^2 + y^2 + z^2} = 1$$

We do this using a procedure identical to normalizing a vector, but operating on all four components of the quaternion rather than on the three values in a vector.

Just as for two-dimensional rotation, we have fixed the problem of messy bounds-checking by adding an extra value to our representation, adding a constraint to remove the extra degree of freedom, and making sure we only get rotations.

In the same way that normalizing our two-dimensional vector representation gave us a point on a circle, normalizing a quaternion can be thought of as giving a point on the surface of a four-dimensional sphere. In fact, lots of the mathematics of quaternions can be derived based on the surface geometry of a four-dimensional sphere. While some developers like to think in these terms (or at least claim they do), personally I find four-dimensional geometry even more difficult to visualize than three-dimensional rotations, so I tend to stick with the algebraic formulation I've given here.

9.3 Angular Velocity and Acceleration

Representing the current orientation of rigid bodies is only one part of the problem. We also need to be able to keep track of how fast and in what direction they are rotating.

Recall that for two dimensions we could use a single value for the angular velocity without the need to perform bounds-checking. The same is true of angular velocity in three dimensions. We abandoned the scaled axis representation for orientations because of the boundary problems. Once again, when we are concerned with the speed at which an object is rotating, we have no bounds: the object can be rotating as fast as it likes.

Our solution is to stick with the scaled axis representation for angular velocity. It has exactly the right number of degrees of freedom, and without the problem of keeping its angle in bounds, the mathematics is simple enough for efficient implementation.

The angular velocity is a three-element vector that can be decomposed into an axis and rate of angular change:

$$\dot{\boldsymbol{\theta}} = r\widehat{\boldsymbol{a}}$$

where $\widehat{\boldsymbol{a}}$ is the axis around which the object is turning and r is the rate at which it is spinning, which (by convention) is measured in radians per second.

The mathematics of vectors matches well with the mathematics of angular velocity. In particular, if we have an object spinning at a certain rate, $\dot{\boldsymbol{\theta}}$, and we add to its rotation a spin at some rate in a new direction, $\boldsymbol{\omega}$, then the new total angular velocity will be given by

$$\dot{\boldsymbol{\theta}}' = \dot{\boldsymbol{\theta}} + \boldsymbol{\omega}$$

In other words, we can add two angular velocities together using vector arithmetic and get a new, and correct, angular velocity.

It's all very well combining angular velocities, but we'll also need to update the orientation by the angular velocity. For linear updates we use the formula

$$p' = p + \ddot{p}t$$

We need some way to do the same for orientation and angular velocity: to update a quaternion by a vector and a time. The equivalent formula is not much more complex:

$$\hat{\theta}' = \hat{\theta} + \frac{\Delta t}{2} \hat{\omega}\hat{\theta}$$

where

$$\hat{\omega} = \begin{bmatrix} 0 \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}$$

which is a quaternion constructed with a zero w component and the remaining components are taken directly from the three components of the angular velocity vector. The constructed quaternion $\hat{\omega}$ doesn't represent an orientation, so it shouldn't be normalized.

I hope you agree that this is really quite painless. We can benefit from the best of both worlds: the ease of vectors for angular velocity and the mathematical properties of quaternions for orientations.

9.3.1 THE VELOCITY OF A POINT

In section 9.1.3 we calculated the position of part of an object even when it had been moved and rotated. To process collisions between objects, in chapter 14 we'll also have to calculate the velocity of any point of an object.

The velocity of a point on an object depends on both its linear and angular velocity:

$$\dot{q} = \dot{\theta} \times (q - p) + \dot{p} \qquad [9.5]$$

where $\dot{q}$ is the velocity of the point, q is the position of the point in world coordinates, p is the position of the origin of the object, and $\dot{\theta}$ is the angular velocity of the object.

If we want to calculate the velocity of a known point on the object (the mirror on the side of the car, for example), we can calculate q from equation 9.2.

9.3.2 ANGULAR ACCELERATION

Because angular acceleration is simply the first derivative of angular velocity, we can use the same vector representation in both acceleration and velocity. What is more,

the relationships between them remain the same as for linear velocity and acceleration. In particular we can update the angular velocity using the equation

$$\dot{\boldsymbol{\theta}}' = \dot{\boldsymbol{\theta}} + \ddot{\boldsymbol{\theta}}t$$

where $\ddot{\boldsymbol{\theta}}$ is the angular acceleration and $\dot{\boldsymbol{\theta}}$ is the angular velocity, as before.

9.4 Implementing the Mathematics

We've covered the theory. Now it's time to implement functions and data structures that are capable of performing the right mathematics. In chapter 2 we created a `Vector3` class that encapsulated vector mathematics; we'll now do the same thing for matrices and quaternions. As part of this process I'll introduce the mathematics of many operations for each type.

If you are working with an existing rendering library, you may already have matrix, vector, and quaternion classes implemented. There is nothing physics-specific in the implementations I give here. You should be able to use your own implementations without alteration. I've personally worked with the DirectX utility library implementations on many projects without having to make any changes to the rest of the physics code.

9.4.1 The Matrix Classes

A matrix is a rectangular array of scalar values. They don't have the same obvious geometric interpretation as vectors do. We will use them in several different contexts, but in each case they will be used to change (or "transform") vectors.

Although matrices can be any size, with any number of rows and columns, we are primarily interested in two kinds: 3×3 matrices and 3×4 matrices. To implement matrices we could create a general matrix data structure capable of supporting any number of rows and columns. We could implement matrix mathematics in the most general way, and use the same code for both of our matrix types (and other types of matrix we might need later). While this would be an acceptable strategy, having the extra flexibility is difficult to optimize. It would be better to create specific data structures for the types of matrix we need. This will be our approach.

We will create a data structure called `Matrix3` for 3×3 matrices and one called `Matrix4` for 3×4 matrices.

The basic data structure for `Matrix3` looks like this:

```
                  ────── Excerpt from include/cyclone/core.h ──────
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
```

```
 * of a rigid body.
 */
class Matrix3
{
public:
    /**
     * Holds the tensor matrix data in array form.
     */
    real data[9];
};
```

and for `Matrix4` it looks like this:

```
─────────── Excerpt from include/cyclone/core.h ───────────
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
public:
    /**
     * Holds the transform matrix data in array form.
     */
    real data[12];
};
```

Clearly there is nothing taxing so far; we just have two arrays of numbers.

Just as we did for the `Vector3` class in chapter 2, we can add methods to these classes to implement their mathematics.

9.4.2 Matrix Multiplication

Since I've said that matrices exist mainly to transform vectors, let's look at this first. We transform a vector by multiplying it by the matrix

$$v' = Mv$$

which is often called "post-multiplication" because the vector occurs after the matrix in the multiplication.

Matrix multiplication works in the same way whether we are multiplying two matrices together or multiplying a matrix and a vector. In fact we can think of a vector as simply a matrix with a single column—a 3×1 matrix.

It is important to realize that matrix multiplication of all kinds is not commutative; in general $ab \neq ba$. In particular, to multiply two matrices the number of columns in the first matrix needs to be the same as the number of rows in the second. So if we wanted to do

$$\boldsymbol{v}M$$

where M is a 3×3 matrix and $\boldsymbol{v}$ is a three-element vector, we would have a mismatch. The vector has one column, and the matrix has three rows. We cannot perform this multiplication: it is undefined. Some game engines do use a pre-multiplication scheme, but they do so by treating vectors as having one row and three columns,

$$[x\, y\, z]$$

rather than the column form we have used. With a row vector we can perform pre-multiplication, but not post-multiplication. Confusingly, I have also seen pre-multiplication mathematics written with the vector after the matrix (i.e., a matrix and then a row vector), so it's worth taking care if you are working with existing code. I will use post-multiplication and column vectors exclusively in this book. If you are working with an engine that uses pre-multiplication, you will have to adapt the order of your code accordingly.

The result of matrix multiplication is a new matrix with the same number of rows as the first matrix in the multiplication, and the same number of columns as the second. So, if we multiply a 3×3 matrix by a 3×1 vector, we get a matrix with three rows and one column (i.e., another vector). If we multiply a 3×3 matrix by another 3×3 matrix, we end up with a 3×3 matrix.

If we are multiplying matrices A and B to give matrix C, each element in C is found by the formula

$$C_{(i,j)} = \sum_k A_{(i,k)} B_{(k,j)}$$

where $C_{(i,j)}$ is the entry in matrix C at the ith row and the jth column, and where k ranges up to the number of columns in the first matrix (i.e., the number of rows in the second—this is why they need to be the same).

For a 3×3 matrix multiplied by a vector we get

$$\begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

With this result we can implement multiplication of a vector by a matrix. I have overloaded the `*` operator for the `matrix` class to perform this.

──────── **Excerpt from include/cyclone/core.h** ────────

```
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
```

```
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

};
```

Matrices as Transformations

Earlier in this chapter I talked about using matrices to represent orientations. In fact matrices can represent rotations, scaling, sheering, and any number of other transformations.

The elements of the matrix control the transformation being performed, and it is worth getting to know how they do it. We can think of the matrix

$$
\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}
$$

as being made up of three vectors:

$$
\begin{bmatrix} a \\ d \\ g \end{bmatrix}, \quad \begin{bmatrix} b \\ e \\ h \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} c \\ f \\ i \end{bmatrix}
$$

These three vectors represent where each of the three main axes—X, Y, and Z—will end up pointing after the transformation. For example, if we have a vector pointing along the positive X axis,

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

it will be transformed into the vector

$$
\begin{bmatrix} a \\ d \\ g \end{bmatrix}
$$

which we can verify with the matrix multiplication

$$\begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a \times 1 + b \times 0 + c \times 0 \\ d \times 1 + e \times 0 + f \times 0 \\ g \times 1 + h \times 0 + i \times 0 \end{bmatrix} = \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

and so on for the other two axes. When I introduced vectors, I mentioned that their three components could be thought of as a position along three axes. The x component is the distance along the X axis and so on. We could write the vector as

$$\boldsymbol{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + z \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

In other words, a vector is made up of some proportion of each basic axis.

If the three axes move under a transformation, then the new location of the vector will be determined in the same way as before. The axes will have moved, but the new vector will still combine them in the same proportions:

$$\boldsymbol{v}' = x \begin{bmatrix} a \\ d \\ g \end{bmatrix} + y \begin{bmatrix} b \\ e \\ h \end{bmatrix} + z \begin{bmatrix} c \\ f \\ i \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Thinking about matrix transformations as a change of axis is an important visualization tool.

The set of axes is called a "basis." We looked at orthonormal bases in chapter 2, where the axes all have a length of 1 and are at right angles to one another. A 3×3 matrix will transform a vector from one basis to another. This is sometimes, not surprisingly, called a "change of basis."

Thinking back to the rotation matrices in section 9.1.3, we saw how the position of a headlamp on a car could be converted into a position in the game level. This is a change of basis. We start with the local coordinates of the headlamp relative to the origin of the car, and end up with the world coordinates of the headlamp in the game.

In the headlamp example we had two stages: first we rotated the object (using a matrix multiplication—a change of basis), and then we translated it (by adding an offset vector). If we extend our matrices a little, we can perform both steps in one go. This is the purpose of the 3×4 matrix.

The 3×4 **Matrices**

If you have been thinking ahead, you may have noticed that, by the matrix multiplication rules, we can't multiply a 3×4 matrix by a 3×1 vector. In fact we will end up

doing just this, but to understand why we need to look more closely at what the 3×4 matrix will be used for.

In the previous section we looked at transformation matrices. The transformations that can be represented as a 3×3 matrix all keep the origin at the same place. To handle general combinations of movement and rotation in our game we need to be able to move the origin around: there is no use modeling a car if it is stuck with its origin at the origin of the game level. We could do this as a two-stage process: perform a rotation matrix multiplication and then add an offset vector. A better alternative is to extend our matrices and do it in one step.

First we extend our vector by one element, so we have four elements, where the last element is always 1:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The four values in the vector are called "homogeneous" coordinates, and they are used in some graphics packages. You can think of them as a four-dimensional coordinate if you like, although thinking in four dimensions may not help you visualize what we're doing with them (it doesn't help me).

If we now take a 3×4 matrix,

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$$

and multiply it in the normal way, by our four-element vector,

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \end{bmatrix} \qquad [9.6]$$

we get a combination of two effects. It is as if we had first multiplied by the 3×3 matrix,

$$\begin{bmatrix} a & b & c \\ e & f & g \\ i & j & k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ ex + fy + gz \\ ix + jy + kz \end{bmatrix}$$

and then added the vector

$$\begin{bmatrix} ax+by+cz \\ ex+fy+gz \\ ix+jy+kz \end{bmatrix} + \begin{bmatrix} d \\ h \\ i \end{bmatrix} = \begin{bmatrix} ax+by+cz+d \\ ex+fy+gz+h \\ ix+jy+kz+l \end{bmatrix}$$

which is exactly the two-step, transform-then-move process we had before, but all in one step. If the first three columns give the directions of the three axes in the new basis, the fourth column gives us the new position of the origin.

We could also view this as multiplying a 4×4 matrix by the 1×4 vector:

$$\begin{bmatrix} a\ b\ c\ d \\ e\ f\ g\ h \\ i\ j\ k\ l \\ 0\ 0\ 0\ 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by+cz+d \\ ex+fy+gz+h \\ ix+jy+kz+l \\ 1 \end{bmatrix}$$

In other words, we start and end with a homogeneous coordinate. Because we are not interested in four-dimensional coordinates, the bottom row of the matrix is always [0 0 0 1] and the last value in the vector is always 1. We can therefore use just the version of the equation given in equation 9.6 and make the fourth value in the multiplied vector (the 1) magically appear as it is needed. We don't store the fourth value in the Vector3 class.

The matrix-vector multiplication gets implemented in the Matrix4 class as

```
                        ── Excerpt from include/cyclone/core.h ──
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Transform the given vector by this matrix.
     *
     * @param vector The vector to transform.
     */
    Vector3 operator*(const Vector3 &vector) const
    {
        return Vector3(
            vector.x * data[0] +
```

```
                vector.y * data[1] +
                vector.z * data[2] + data[3],

                vector.x * data[4] +
                vector.y * data[5] +
                vector.z * data[6] + data[7],

                vector.x * data[8] +
                vector.y * data[9] +
                vector.z * data[10] + data[11]
            );
        }
    };
```

Multiplying Two Matrices

We can use exactly the same process to multiply two matrices together. If we multiply two 3×3 matrices together, we get another 3×3 matrix. This can be easily done with this code:

```
                      ─── Excerpt from include/cyclone/core.h ───
    /**
     * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
     * This matrix is not padding to produce an aligned structure, since
     * it is most commonly used with a mass (single real) and two
     * damping coefficients to make the 12-element characteristics array
     * of a rigid body.
     */
    class Matrix3
        // ... Other Matrix3 code as before ...

        /**
         * Returns a matrix that is this matrix multiplied by the given
         * other matrix.
         */
        Matrix3 operator*(const Matrix3 &o) const
        {
            return Matrix3(
                data[0]*o.data[0] + data[1]*o.data[3] + data[2]*o.data[6],
                data[0]*o.data[1] + data[1]*o.data[4] + data[2]*o.data[7],
                data[0]*o.data[2] + data[1]*o.data[5] + data[2]*o.data[8],

                data[3]*o.data[0] + data[4]*o.data[3] + data[5]*o.data[6],
                data[3]*o.data[1] + data[4]*o.data[4] + data[5]*o.data[7],
                data[3]*o.data[2] + data[4]*o.data[5] + data[5]*o.data[8],
```

```
                data[6]*o.data[0] + data[7]*o.data[3] + data[8]*o.data[6],
                data[6]*o.data[1] + data[7]*o.data[4] + data[8]*o.data[7],
                data[6]*o.data[2] + data[7]*o.data[5] + data[8]*o.data[8]
                );
    }
};
```

Multiplying two matrices together in this way combines their effects. If matrices *A* and *B* are two transformations, then the matrix *AB* will represent the combined transformation. Order is crucial for both transformations and matrix multiplication: the matrix *AB* is a transformation that would result from *first* doing *B* and *then* doing *A*. In other words, the order of the transformations is the opposite of the order of the matrices in the multiplication. This is a "gotcha" that catches even experienced developers from time to time.

So much for 3×3 matrices; how about 3×4 matrices? From the rules of matrix multiplication we can't multiply two 3×4 matrices together: the columns of the first matrix don't match the rows of the second. To make progress we need to return to the full form of our 4×4 matrix. Remember that the matrix we are storing as

$$\begin{bmatrix} a\ b\ c\ d \\ e\ f\ g\ h \\ i\ j\ k\ l \end{bmatrix}$$

can be thought of as shorthand for

$$\begin{bmatrix} a\ b\ c\ d \\ e\ f\ g\ h \\ i\ j\ k\ l \\ 0\ 0\ 0\ 1 \end{bmatrix}$$

We can certainly multiply two 4×4 matrices together. If we multiply two 4×4 matrices with [0 0 0 1] as their bottom line, we end up with another matrix whose bottom line is [0 0 0 1] (try it to convince yourself).

So in our code, when we come to multiply two 3×4 matrices (to combine their transformations), we magically make the extra values appear, without storing them, exactly as we did for transforming vectors. The code looks like this:

──────────── **Excerpt from include/cyclone/core.h** ────────────

```
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
```

```
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Returns a matrix that is this matrix multiplied by the given
     * other matrix.
     */
    Matrix4 operator*(const Matrix4 &o) const
    {
        Matrix4 result;
        result.data[0] = (o.data[0]*data[0]) + (o.data[4]*data[1]) +
                         (o.data[8]*data[2]);
        result.data[4] = (o.data[0]*data[4]) + (o.data[4]*data[5]) +
                         (o.data[8]*data[6]);
        result.data[8] = (o.data[0]*data[8]) + (o.data[4]*data[9]) +
                         (o.data[8]*data[10]);

        result.data[1] = (o.data[1]*data[0]) + (o.data[5]*data[1]) +
                         (o.data[9]*data[2]);
        result.data[5] = (o.data[1]*data[4]) + (o.data[5]*data[5]) +
                         (o.data[9]*data[6]);
        result.data[9] = (o.data[1]*data[8]) + (o.data[5]*data[9]) +
                         (o.data[9]*data[10]);

        result.data[2] = (o.data[2]*data[0]) + (o.data[6]*data[1]) +
                         (o.data[10]*data[2]);
        result.data[6] = (o.data[2]*data[4]) + (o.data[6]*data[5]) +
                         (o.data[10]*data[6]);
        result.data[10] = (o.data[2]*data[8]) + (o.data[6]*data[9]) +
                          (o.data[10]*data[10]);

        result.data[3] = (o.data[3]*data[0]) + (o.data[7]*data[1]) +
                         (o.data[11]*data[2]) + data[3];
        result.data[7] = (o.data[3]*data[4]) + (o.data[7]*data[5]) +
                         (o.data[11]*data[6]) + data[7];
        result.data[11] = (o.data[3]*data[8]) + (o.data[7]*data[9]) +
                          (o.data[11]*data[10]) + data[11];

        return result;
    }
};
```

Some graphics libraries use a full 16-element matrix for transforms; most of those (but not all) will also use four-element vectors for position. They allow the programmer to work in four dimensions. There are some interesting graphical effects that are made possible this way, including the perspective transformations needed to model a camera. If you are relying on the mathematics libraries that these APIs provide, you will not need to worry about the number of entries in the matrix: chances are you'll only be using the first 12 for your physics development. If you are implementing the mathematics classes as I have been, then you have the choice of whether to use the full or the 3 × 4 matrix.

Whereas we added an extra padding element to our vector class so that it sits nicely on machines with 128-bit math processors and 16-byte alignment, we don't need to do the same for matrices because each row of the matrix is 128 bits long (assuming we're using 32-bit floating-point numbers, although running this at double precision will be much slower in any case).

The code will take less memory if you use 3 × 4 matrices and rely on the last, unstored line of every matrix being [0 0 0 1]. But check to see whether the machine you are developing has built-in hardware-level support for matrix transformation. Implementing your own routines and ignoring these will result in worse performance (and take more effort) in the long run.

9.4.3 The Matrix Inverse and Transpose

A matrix represents a transformation, and we often need to find out how to reverse the transformation. If we have a matrix that changes from an object's local coordinates to world coordinates, it will be useful to be able to create a matrix that gets us back again: converting world coordinates to local coordinates.

For example, if we determine that our car has collided with a barrier, we know the position of the collision in world coordinates. We'd like to be able to turn this position into local coordinates to see which bit of the car got hit.

If a matrix transforms vectors from one basis to another, then the inverse of the matrix can convert them back. If we combine a matrix with its inverse, we get the "identity matrix": a matrix representing a transformation that has no effect. In other words, if we transform a vector by a matrix and then by its inverse, we get back to where we started:

$$M^{-1}M = I$$

For a 3 × 3 matrix, the identity matrix is

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

Inverting large matrices is a challenging computer science problem (in fact, it is the fundamental problem that the most complex game physics engines try to solve,

as we'll see in chapter 18). Techniques involve walking through the matrix and rearranging its elements using a range of mathematical manipulations. Fortunately, for 3×3 and 4×4 matrices we can write the solutions directly. For a 3×3 matrix,

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

the inverse is

$$M^{-1} = \frac{1}{\det M} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \qquad [9.7]$$

where $\det M$ is the determinant of the matrix, which for a 3×3 matrix is

$$\det M = aei + dhc + gbf - ahf - gec - dbi$$

Because we take 1 over the determinant in equation 9.7, the inverse only exists if the determinant is non-zero.

You can verify for yourself that the inverse matrix, when multiplied by the original matrix, does give the identity matrix. The reason the inverse has the form it does, and what the meaning of the determinant is, are beyond the scope of this book.[2] To understand why the preceding equations work, we'd have to cover various bits of matrix mathematics that we otherwise wouldn't need. If you are interested in the features and mathematics of matrices, any undergraduate textbook on matrix analysis will provide more details. For an even more exhaustive (if considerably tougher) treatment, I'd recommend Horn and Johnson [1990] and [1994], two highly respected references on the topic.

We can implement our 3×3 matrix inverse as follows:

Excerpt from `include/cyclone/core.h`

```
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
```

---

2. A good rule of thumb I use (which may offend mathematical purists) is to think of the determinant as the "size" of the matrix. In fact, for a 2×2 dimensional matrix the determinant is the area of the parallelogram formed from its column vectors, and for a 3×3 matrix it is the area of the parallelepiped formed from its three columns.

The inverse formula of equation 9.7 can then be thought of as adjusting the elements and dividing by the size of the matrix. Thinking this way can cause problems with more advanced matrix math, so remember that it's only a mnemonic.

```
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

    /**
     * Sets the matrix to be the inverse of the given matrix.
     *
     * @param m The matrix to invert and use to set this.
     */
    void setInverse(const Matrix3 &m)
    {
        real t4 = m.data[0]*m.data[4];
        real t6 = m.data[0]*m.data[5];
        real t8 = m.data[1]*m.data[3];
        real t10 = m.data[2]*m.data[3];
        real t12 = m.data[1]*m.data[6];
        real t14 = m.data[2]*m.data[6];

        // Calculate the determinant.
        real t16 = (t4*m.data[8] - t6*m.data[7] - t8*m.data[8] +
                    t10*m.data[7] + t12*m.data[5] - t14*m.data[4]);

        // Make sure the determinant is non-zero.
        if (t16 == (real)0.0f) return;
        real t17 = 1/t16;

        data[0] = (m.data[4]*m.data[8]-m.data[5]*m.data[7])*t17;
        data[1] = -(m.data[1]*m.data[8]-m.data[2]*m.data[7])*t17;
        data[2] = (m.data[1]*m.data[5]-m.data[2]*m.data[4])*t17;
        data[3] = -(m.data[3]*m.data[8]-m.data[5]*m.data[6])*t17;
        data[4] = (m.data[0]*m.data[8]-t14)*t17;
        data[5] = -(t6-t10)*t17;
        data[6] = (m.data[3]*m.data[7]-m.data[4]*m.data[6])*t17;
        data[7] = -(m.data[0]*m.data[7]-t12)*t17;
        data[8] = (t4-t8)*t17;
    }

    /** Returns a new matrix containing the inverse of this matrix. */
    Matrix3 inverse() const
    {
        Matrix3 result;
        result.setInverse(*this);
        return result;
```

```
    }

    /**
     * Inverts the matrix.
     */
    void invert()
    {
        setInverse(*this);
    }
};
```

Only square matrices have an inverse. For a 3×4 matrix, we need to again remember that our matrix is shorthand for a 4×4 matrix. The 4×4 matrix has an inverse that can be written in much the same way as the 3×3 matrix. And fortunately for us the resulting matrix will have a bottom row of [0 0 0 1], so we can represent the inverse as a 3×4 matrix.

Unfortunately the algebra is more complex still, and it would run to about a page of equations. Assuming your aim is to implement the code, I'll skip the long equations and give the implementation:

```
                    ___ Excerpt from include/cyclone/core.h ___
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Returns the determinant of the matrix.
     */
    real getDeterminant() const;

    /**
     * Sets the matrix to be the inverse of the given matrix.
     *
     * @param m The matrix to invert and use to set this.
     */
    void setInverse(const Matrix4 &m);

    /** Returns a new matrix containing the inverse of this matrix. */
```

```
    Matrix4 inverse() const
    {
        Matrix4 result;
        result.setInverse(*this);
        return result;
    }


    /**
     * Inverts the matrix.
     */
    void invert()
    {
        setInverse(*this);
    }
};
```

———————— **Excerpt from src/core.cpp** ————————

```
real Matrix4::getDeterminant() const
{
    return data[8]*data[5]*data[2]+
        data[4]*data[9]*data[2]+
        data[8]*data[1]*data[6]-
        data[0]*data[9]*data[6]-
        data[4]*data[1]*data[10]+
        data[0]*data[5]*data[10];
}

void Matrix4::setInverse(const Matrix4 &m)
{
    // Make sure the determinant is non-zero.
    real det = getDeterminant();
    if (det == 0) return;
    det = ((real)1.0)/det;

    data[0] = (-m.data[9]*m.data[6]+m.data[5]*m.data[10])*det;
    data[4] = (m.data[8]*m.data[6]-m.data[4]*m.data[10])*det;
    data[8] = (-m.data[8]*m.data[5]+m.data[4]*m.data[9]* m.data[15])*det;

    data[1] = (m.data[9]*m.data[2]-m.data[1]*m.data[10])*det;
    data[5] = (-m.data[8]*m.data[2]+m.data[0]*m.data[10])*det;
    data[9] = (m.data[8]*m.data[1]-m.data[0]*m.data[9]* m.data[15])*det;

    data[2] = (-m.data[5]*m.data[2]+m.data[1]*m.data[6]* m.data[15])*det;
```

```
    data[6] = (+m.data[4]*m.data[2]-m.data[0]*m.data[6]* m.data[15])*det;
    data[10] = (-m.data[4]*m.data[1]+m.data[0]*m.data[5]* m.data[15])*det;

    data[3] = (m.data[9]*m.data[6]*m.data[3]
              -m.data[5]*m.data[10]*m.data[3]
              -m.data[9]*m.data[2]*m.data[7]
              +m.data[1]*m.data[10]*m.data[7]
              +m.data[5]*m.data[2]*m.data[11]
              -m.data[1]*m.data[6]*m.data[11])*det;
    data[7] = (-m.data[8]*m.data[6]*m.data[3]
              +m.data[4]*m.data[10]*m.data[3]
              +m.data[8]*m.data[2]*m.data[7]
              -m.data[0]*m.data[10]*m.data[7]
              -m.data[4]*m.data[2]*m.data[11]
              +m.data[0]*m.data[6]*m.data[11])*det;
    data[11] =(m.data[8]*m.data[5]*m.data[3]
              -m.data[4]*m.data[9]*m.data[3]
              -m.data[8]*m.data[1]*m.data[7]
              +m.data[0]*m.data[9]*m.data[7]
              +m.data[4]*m.data[1]*m.data[11]
              -m.data[0]*m.data[5]*m.data[11])*det;
}
```

You'll notice from this code that the inverse again exists only when the determinant of the matrix is non-zero.

The Matrix Transpose

Whenever the determinant is non-zero, we can always use the preceding equations to find the inverse of a matrix. It is not the simplest process, however, and in some cases we can do much better.

If we have a matrix that represents a rotation only, we can make use of the fact that the inverse of the transformation is another rotation, about the same axis but at the opposite angle. This is equivalent to inverting the axis and using the same angle. We can create a matrix that rotates the same degree in the opposite direction by transposing the original matrix.

The transpose of a matrix

$$M = \begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix}$$

is made by swapping its rows and columns:

$$M^\top = \begin{bmatrix} a\ d\ g \\ b\ e\ h \\ c\ f\ i \end{bmatrix}$$

If M is a rotation matrix, then

$$M^\top = M^{-1}$$

We can implement this for our 3×3 matrix in this way:

```
────── Excerpt from include/cyclone/core.h ──────
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

    /**
     * Sets the matrix to be the transpose of the given matrix.
     *
     * @param m The matrix to transpose and use to set this.
     */
    void setTranspose(const Matrix3 &m)
    {
        data[0] = m.data[0];
        data[1] = m.data[3];
        data[2] = m.data[6];
        data[3] = m.data[1];
        data[4] = m.data[4];
        data[5] = m.data[7];
        data[6] = m.data[2];
        data[7] = m.data[5];
        data[8] = m.data[8];
    }

    /** Returns a new matrix containing the transpose of this matrix. */
    Matrix3 transpose() const
    {
        Matrix3 result;
```

```
            result.setTranspose(*this);
            return result;
        }
    };
```

It will be useful at several points in the engine to transpose rather than request a full inverse when we know the matrix is a rotation matrix only.

There is no point implementing a transpose function for the 3×4 matrix. It doesn't have a geometric correlate: transposing a homogeneous matrix doesn't make sense geometrically. If there is any non-zero element in the fourth column, then it will be transposed into the fourth row, which we don't have in our matrix.

This makes sense: we will only use transposition to do cheap inverses on rotation matrices. If the 3×4 were a pure rotation matrix with no translation, then it would have zeros in its fourth column. If this were the case, we could represent it as a 3×3 matrix.

There are other reasons to transpose a matrix, outside of our needs. If you are working with an existing matrix library with a full 4×4 matrix implementation, it is likely to have a transpose function.

9.4.4   CONVERTING A QUATERNION TO A MATRIX

In addition to matrix manipulation, we'll need an operation to convert a quaternion to a matrix. Your graphics engine is likely to need transformations expressed as a matrix. In order to draw an object we'll need to convert from its position vector and orientation quaternion into a transform matrix for rendering.

Sometimes we'll want just the rotation matrix in its 3×3 form, and other times we'll want the full transformation matrix. In each case the conversion from a quaternion to a matrix uses the results we saw in sections 9.2.3 and 9.2.4, where both the quaternion and the rotation matrix were expressed in terms of an axis and an angle.

We could reconstruct the axis and angle from the quaternion and then feed it into equation 9.3. If we do this, we can simplify out the axes and angles and find an expression for the matrix purely in terms of the coefficients of the quaternion

$$\Theta = \begin{bmatrix} 1 - (2y^2 + 2z^2) & 2xy + 2zw & 2xz - 2yw \\ 2xy - 2zw & 1 - (2x^2 + 2z^2) & 2yz + 2xw \\ 2xz + 2yw & 2yz - 2xw & 1 - (2x^2 + 2y^2) \end{bmatrix}$$

where w, x, y, and z are the components of the quaternion

$$\hat{\theta} = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

When implemented, the 3×3 version, including rotation, looks like this:

```
_____ Excerpt from include/cyclone/core.h _____
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

    /**
     * Sets this matrix to be the rotation matrix corresponding to
     * the given quaternion.
     */
    void setOrientation(const Quaternion &q)
    {
        data[0] = 1 - (2*q.j*q.j + 2*q.k*q.k);
        data[1] = 2*q.i*q.j + 2*q.k*q.r;
        data[2] = 2*q.i*q.k - 2*q.j*q.r;
        data[3] = 2*q.i*q.j - 2*q.k*q.r;
        data[4] = 1 - (2*q.i*q.i  + 2*q.k*q.k);
        data[5] = 2*q.j*q.k + 2*q.i*q.r;
        data[6] = 2*q.i*q.k + 2*q.j*q.r;
        data[7] = 2*q.j*q.k - 2*q.i*q.r;
        data[8] = 1 - (2*q.i*q.i  + 2*q.j*q.j);
    }
};
```

The 3×4 version, adding position to the rotation, looks like this:

```
_____ Excerpt from include/cyclone/core.h _____
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Sets this matrix to be the rotation matrix corresponding to
```

```
      * the given quaternion.
      */
     void setOrientationAndPos(const Quaternion &q, const Vector3 &pos)
     {
         data[0] = 1 - (2*q.j*q.j + 2*q.k*q.k);
         data[1] = 2*q.i*q.j + 2*q.k*q.r;
         data[2] = 2*q.i*q.k - 2*q.j*q.r;
         data[3] = pos.x;

         data[4] = 2*q.i*q.j - 2*q.k*q.r;
         data[5] = 1 - (2*q.i*q.i  + 2*q.k*q.k);
         data[6] = 2*q.j*q.k + 2*q.i*q.r;
         data[7] = pos.y;

         data[8] = 2*q.i*q.k + 2*q.j*q.r;
         data[9] = 2*q.j*q.k - 2*q.i*q.r;
         data[10] = 1 - (2*q.i*q.i  + 2*q.j*q.j);
         data[11] = pos.z;
     }
};
```

9.4.5   Transforming Vectors

In section 9.1.3 we looked at finding the position of part of an object, even when the object had been moved and rotated. This is a conversion between object coordinates (i.e., the position of the part relative to the origin of the object and its axes) and world coordinates (its position relative to the global origin and direction of axes).

This conversion can be performed by multiplying the local coordinates by the object's transform matrix. The transform matrix in turn can be generated from the quaternion and position as we saw earlier. We end up with a 3×4 transform matrix. Working out the world coordinates, given local coordinates and a transform matrix, is a matter of simply multiplying the vector by the matrix:

```
Vector3 localToWorld(const Vector3 &local, const Matrix4 &transform)
{
    return transform.transform(local);
}
```

The opposite transform, from world coordinates to local coordinates, involves the same process but using the inverse of the transform matrix. The inverse does the opposite of the original matrix: it converts world coordinates into local coordinates.

```
Vector3 worldToLocal(const Vector3 &world, const Matrix4 &transform)
{
    Matrix4 inverseTransform;
    inverseTransform.setInverse(transform);

    return inverseTransform.transform(world);
}
```

If the transform matrix is made up of only a rotation and a translation (as it should be for our needs), we can do this in one step.

We split the 3×4 matrix into two components: the translation vector (i.e., the fourth column of the matrix) and the 3×3 rotation matrix. First we perform the inverse translation by simply subtracting the translation vector. Then we make use of the fact that the inverse of a 3×3 rotation matrix is simply its transpose, and multiply by the transpose.

This can be done in one method that looks like this:

Excerpt from include/cyclone/core.h

```
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Transform the given vector by the transformational inverse
     * of this matrix.
     */
    Vector3 transformInverse(const Vector3 &vector) const
    {
        Vector3 tmp = vector;
        tmp.x -= data[3];
        tmp.y -= data[7];
        tmp.z -= data[11];
        return Vector3(
            tmp.x * data[0] +
            tmp.y * data[4] +
            tmp.z * data[8],

            tmp.x * data[1] +
```

```
            tmp.y * data[5] +
            tmp.z * data[9],

            tmp.x * data[2] +
            tmp.y * data[6] +
            tmp.z * data[10]
        );
    }
};
```

which is called like this:

```
Vector3 worldToLocal(const Vector3 &world, const Matrix4 &transform)
{
    return transform.transformInverse(world);
}
```

Recall from chapter 2 that vectors can represent positions as well as directions. This is a significant distinction when it comes to transforming vectors. So far we have looked at vectors representing positions. In this case converting between local and object coordinates is a matter of multiplying by the transform matrix, as we have seen.

For direction vectors, however, the same is not true. If we start with a direction vector in object space, for example, the Z-axis direction vector

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

and multiply it by a transformation matrix—for example, the translation only

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

we end up with a direction vector, that of

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

Clearly converting the local Z-axis direction vector into world coordinates, for an object that has no rotation, should give us the Z-axis direction vector. Directions should not change magnitude, and if there is no rotation, they should not change at all.

In other words, direction vectors should be immune to any translational component of the transformation matrix. We can do this only by multiplying the vector by a 3×3 matrix, which ensures that there is no translational component. Unfortunately this will be inconvenient at several points, because we will have gone to the trouble of building a 3×4 transform matrix, and it would be a waste to create another matrix just for transforming directions. To solve this we can add two specialized methods to the `Matrix4` class to deal specifically with transforming vectors. One performs the normal transformation (from local to world coordinates), and the other performs the inverse (from world to local coordinates).

Excerpt from include/cyclone/core.h

```
/**
 * Holds a transform matrix, consisting of a rotation matrix and
 * a position. The matrix has 12 elements; it is assumed that the
 * remaining four are (0,0,0,1), producing a homogenous matrix.
 */
class Matrix4
{
    // ... Other Matrix4 code as before ...

    /**
     * Transform the given direction vector by this matrix.
     */
    Vector3 transformDirection(const Vector3 &vector) const
    {
        return Vector3(
            vector.x * data[0] +
            vector.y * data[1] +
            vector.z * data[2],

            vector.x * data[4] +
            vector.y * data[5] +
            vector.z * data[6],

            vector.x * data[8] +
            vector.y * data[9] +
            vector.z * data[10]
        );
    }
```

```
    /**
     * Transform the given direction vector by the
     * transformational inverse of this matrix.
     */
    Vector3 transformInverseDirection(const Vector3 &vector) const
    {
        return Vector3(
            vector.x * data[0] +
            vector.y * data[4] +
            vector.z * data[8],

            vector.x * data[1] +
            vector.y * data[5] +
            vector.z * data[9],

            vector.x * data[2] +
            vector.y * data[6] +
            vector.z * data[10]
        );
    }
};
```

These can be called in the same way as before:

```
Vector3 localToWorldDirn(const Vector3 &local, const Matrix4 &transform)
{
    return transform.transformDirection(local);
}
```

and

```
Vector3 worldToLocalDirn(const Vector3 &world, const Matrix4 &transform)
{
    return transform.transformInverseDirection(world);
}
```

9.4.6 Changing the Basis of a Matrix

There is one final thing we'll need to do with matrices that hasn't been covered yet. Recall that we can think of a transformation matrix as converting between one basis and another, one set of axes and another. If the transformation is a 3×4 matrix,
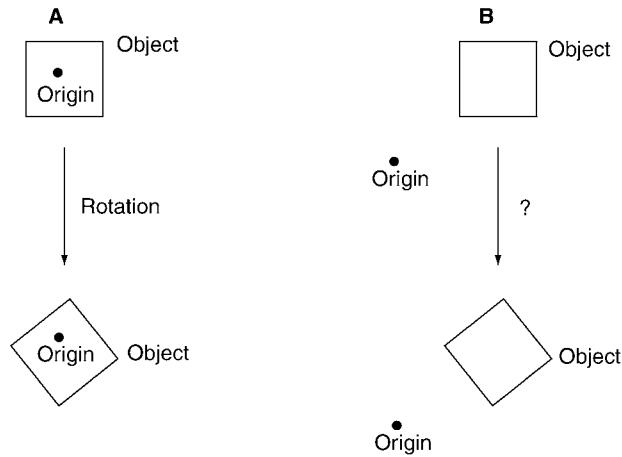
FIGURE 9.6   A matrix has its basis changed.

then the change can also involve a shift in the origin. We used this transformation to convert a vector from one basis to another.

We will also meet a situation in which we need to transform a whole matrix from one basis to another. This can be a little more difficult to visualize.

Let's say we have a matrix M_t that performs some transformation, as shown in the first part of figure 9.6. (The figure is in 2D for ease of illustration, but the same principles apply in 3D.) It performs a small rotation around the origin; part A of the figure shows an object being rotated.

Now let's say we have a different basis, but we want exactly the same transformation. In our new basis we'd like to find a transformation that has the same effect (i.e., it leaves the object at the same final position), but works with the new coordinate system. This is shown in part B of figure 9.6; now the origin has moved (we're in a different basis), but we'd like the effect of the transformation to be the same. Clearly, if we applied M_t in the new basis, it would give a different end result.

Let's assume we have a transformation M_b between our original basis $\mathcal{B}_1$ and our new basis $\mathcal{B}_2$. Is there some way we can create a new transformation from M_t and M_b that would replicate the behavior that M_t gave us in $\mathcal{B}_1$ but in the new $\mathcal{B}_2$?

The solution is to use M_b and M_b^{-1} in a three-stage process:

1. We perform the transformation M_b^{-1}, which takes us from $\mathcal{B}_2$ back into $\mathcal{B}_1$.

2. We then perform the original transform M_t, since we are now in the basis $\mathcal{B}_1$, where it was originally correct.

3. We then need to get back into basis $\mathcal{B}_2$, so we apply transformation M_b.

So we end up with

$$M'_t = M_b M_t M_b^{-1}$$

bearing in mind that multiplied matrices are equivalent to transformations carried out in reverse order.

We will need to use this function whenever we have a matrix expressed in one basis and we need it in another. We can do this using the multiplication and inverse functions we have already implemented: there is no need for a specialized function.

In particular the technique will be indispensable in the next chapter when we come to work with the inertia tensor of a rigid body. At that stage I will provide a dedicated implementation that takes advantage of some other properties of the inertia tensor that simplifies the mathematics.

9.4.7 THE QUATERNION CLASS

We've covered the basic mathematical operations for matrices and have a solid `Matrix` and `Vector` class implemented. Before we can move on, we also need to create a data structure to manipulate quaternions.

In this section we will build a Quaternion class. The basic data structure looks like this:

───── **Excerpt from include/cyclone/core.h** ─────

```
/**
 * Holds a three degree of freedom orientation.
 */
class Quaternion
{
public:
    union {
        struct {
            /**
             * Holds the real component of the quaternion.
             */
            real r;

            /**
             * Holds the first complex component of the quaternion.
             */
            real i;

            /**
             * Holds the second complex component of the quaternion.
             */
            real j;
```

```
                  /**
                   * Holds the third complex component of the quaternion.
                   */
                  real k;
              };

              /**
               * Holds the quaternion data in array form.
               */
              real data[4];
          };
      };
```

9.4.8 NORMALIZING QUATERNIONS

As we saw in the earlier discussion, quaternions only represent a rotation if they have a magnitude of 1. All the operations we will be performing keep the magnitude at 1, but numerical inaccuracies and rounding errors can cause this constraint to be violated over time. Once in a while it is a good idea to renormalize the quaternion. We can perform this with the following method:

Excerpt from `include/cyclone/core.h`

```
/**
 * Holds a three degree of freedom orientation.
 */
class Quaternion
{
    // ... other Quaternion code as before ...

    /**
     * Normalizes the quaternion to unit length, making it a valid
     * orientation quaternion.
     */
    void normalize()
    {
        real d = r*r+i*i+j*j+k*k;

        // Check for zero length quaternion, and use the no-rotation
        // quaternion in that case.
        if (d == 0) {
            r = 1;
            return;
        }
```

```
                d = ((real)1.0)/real_sqrt(d);
                r *= d;
                i *= d;
                j *= d;
                k *= d;
            }
    };
```

9.4.9   COMBINING QUATERNIONS

We combine two quaternions by multiplying them together. This is exactly the same as for rotation (or any other transformation) matrix: the result of $\hat{q}\hat{p}$ is a rotation that is equivalent to performing rotation $(\hat{p})$ first, then $(\hat{q})$.

As we saw in section 9.2.4, the multiplication of two quaternions has the following form:

$$\begin{bmatrix} w_1 \\ x_1 \\ y_1 \\ z_1 \end{bmatrix} \begin{bmatrix} w_2 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \\ w_1 x_2 + x_1 w_2 - y_1 z_2 - z_1 y_2 \\ w_1 y_2 - x_1 z_2 + y_1 w_2 - z_1 x_2 \\ w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2 \end{bmatrix}$$

which is implemented as

```
                    ────── Excerpt from include/cyclone/core.h ──────
    /**
     * Holds a three degree of freedom orientation.
     */
    class Quaternion
    {
        // ... other Quaternion code as before ...

        /**
         * Multiplies the quaternion by the given quaternion.
         *
         * @param multiplier The quaternion by which to multiply.
         */
        void operator *=(const Quaternion &multiplier)
        {
            Quaternion q = *this;
            r = q.r*multiplier.r - q.i*multiplier.i -
                q.j*multiplier.j - q.k*multiplier.k;
            i = q.r*multiplier.i + q.i*multiplier.r +
                q.j*multiplier.k - q.k*multiplier.j;
```

```
            j = q.r*multiplier.j + q.j*multiplier.r +
                q.k*multiplier.i - q.i*multiplier.k;
            k = q.r*multiplier.k + q.k*multiplier.r +
                q.i*multiplier.j - q.j*multiplier.i;
        }
    };
```

9.4.10 ROTATING

Occasionally we need to rotate a quaternion by some given amount. If a quaternion represents the orientation of an object, and we need to alter that orientation by rotating it, we can convert the orientation and the desired rotation into matrices and multiply them. But there is a more direct way to do this.

The amount of rotation is most simply represented as a vector (since the rotation amount isn't bounded, just as we saw for angular velocity). We can then alter the quaternion using the equation

$$\overset{'}{\underset{\wedge}{\theta}} = \underset{\wedge}{\theta} + \frac{1}{2}\underset{\wedge}{\Delta\theta}\underset{\wedge}{\theta} \tag{9.8}$$

which is similar to the equation we saw in section 9.3, but replaces the velocity $\times$ time with a single absolute angular change (θ).

Here, as in the case of angular velocity, the rotation is provided as a vector converted into a non-normalized quaternion:

$$[\Delta\theta_x \Delta\theta_y \Delta\theta_z] \rightarrow [0 \Delta\theta_x \Delta\theta_y \Delta\theta_z]$$

This can be implemented as

--- **Excerpt from include/cyclone/core.h** ---

```
    /**
     * Holds a three degree of freedom orientation.
     */
    class Quaternion
    {
        // ... other Quaternion code as before ...

        void rotateByVector(constVector& vector)
        {
            Quaternion q(0, vector.x * scale, vector.y * scale,
                        vector.z * scale);
            (*this) *= q;
        }
    };
```

9.4.11 UPDATING BY THE ANGULAR VELOCITY

The final operation we need to do is to update the orientation quaternion by applying the angular velocity for a specified duration of time. In section 9.3 we saw that this is handled by the equation

$$\hat{\theta}' = \hat{\theta} + \frac{\delta t}{2} \hat{\omega} \hat{\theta}$$

where $\hat{\omega}$ is the quaternion form of the angular velocity and t is the duration to update by. This can be implemented as

---
Excerpt from `include/cyclone/core.h`

```
/**
 * Holds a three degree of freedom orientation.
 */
class Quaternion
{
    // ... other Quaternion code as before ...

    /**
     * Adds the given vector to this, scaled by the given amount.
     * This is used to update the orientation quaternion by a rotation
     * and time.
     *
     * @param vector The vector to add.
     *
     * @param scale The amount of the vector to add.
     */
    void addScaledVector(const Vector3& vector, real scale)
    {
        Quaternion q(0,
            vector.x * scale,
            vector.y * scale,
            vector.z * scale);
        q *= *this;
        r += q.r * ((real)0.5);
        i += q.i * ((real)0.5);
        j += q.j * ((real)0.5);
        k += q.k * ((real)0.5);
    }
};
```

We now have a `Quaternion` class that contains all the functionality we need for the rest of the engine. As with vectors and matrices, there are a lot of other operations we could add—more conversions, other mathematical operators. If you are using an existing quaternion library, it might have other functions defined, but since we will not need them, I will avoid giving implementations that we won't use.

9.5 SUMMARY

We have come a long way in this chapter, and if you weren't familiar with matrices and quaternions before, then this has been a big step. We've now met all the mathematics we need to see us through to our final physics engine.

In this chapter I've hinted at the way some of this mathematics is used in the engine. Chapter 10 starts to rebuild our engine to support full 3D rigid bodies, with angular as well as linear motion.

This page intentionally left blank
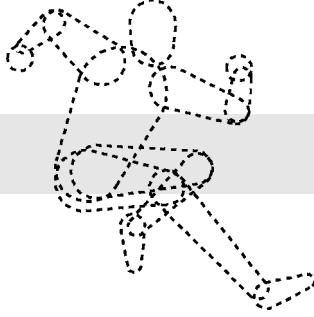
10

Laws of Motion for Rigid Bodies

In this chapter we're going to repeat the work we did in chapters 2, 3, and 5, this time working with rigid bodies rather than particles. We'll do this by creating a new class: `RigidBody`.

In section 9.1.3 I mentioned that if an object's origin is placed at its center of mass, then its linear motion will be just like that of a particle. We'll make use of that fact in this chapter. Almost all the code for the linear motion of our rigid body is lifted directly from our particle class. To this we will add two things:

1. The laws of motion for rotating bodies, equivalent to the way we implemented Newton's second law of motion.

2. The mathematics of forces that have both a linear and a rotational effect. The question is, for a given force applied, how much will the object rotate?

With the rigid body in place, and these two extensions implemented, we will have a rigid-body physics engine equivalent to the particle engine from part I. Adding collisions and hard constraints will then occupy us for the remainder of this book.

10.1 The Rigid Body

We can start by creating a `RigidBody` class, containing the same information we had in the `Particle` class, and adding the extra data structures for rotation that we met in the previous chapter. The code looks like the following.

```
                        ── Excerpt from include/cyclone/body.h ──
/**
 * A rigid body is the basic simulation object in the physics core.
 */
class RigidBody
{
public:
    /**
     * Holds the inverse of the mass of the rigid body. It is more
     * useful to hold the inverse mass because integration is simpler,
     * and because in real time simulation it is more useful to have
     * bodies with infinite mass (immovable) than zero mass (completely
     * unstable in numerical simulation).
     */
    real inverseMass;
    /**
     * Holds the linear position of the rigid body in world space.
     */
    Vector3 position;

    /**
     * Holds the angular orientation of the rigid body in world space.
     */
    Quaternion orientation;

    /**
     * Holds the linear velocity of the rigid body in world space.
     */
    Vector3 velocity;

    /**
     * Holds the angular velocity, or rotation, or the rigid body
     * in world space.
     */
    Vector3 rotation;

    /**
     * Holds a transform matrix for converting body space into world
     * space and vice versa. This can be achieved by calling the
     * getPointIn*Space functions.
    Matrix4 transformMatrix;

};
```

I have added a matrix to the class to hold the current transform matrix for the object. This matrix is useful for rendering the object and will be useful at various points in the physics too, so much so that it is worth the storage space to keep a copy with the rigid body.

It should be derived from the orientation and position of the body once per frame to make sure it is correct. We will not update the matrix within the physics or use it in any way where it might get out of synch with the orientation and position. We're not trying to store the same information twice: the position and orientation are in charge; the transform matrix member just acts as a cache to avoid repeatedly recalculating this important quantity.

I call this "derived data," and it is the first of a handful we'll add to the rigid body. If you are working on a highly memory-starved machine, you may want to remove these data: they are only copies of existing information in a more convenient form. You can simply calculate them as they are needed. The same is true for all the derived data I will add to the rigid body.

Let's add a function to the class to calculate the transform matrix and a function to calculate all derived data. Initially `calculateDerivedData` will only calculate the transform matrix:

```
                    ─── Excerpt from include/cyclone/body.h ───
    class RigidBody
    {
        // ... Other RigidBody code as before ...

        /**
         * Calculates internal data from state data. This should be called
         * after the body's state is altered directly (it is called
         * automatically during integration). If you change the body's
         * state and then intend to integrate before querying any data
         * (such as the transform matrix), then you can omit this step.
         */
        void calculateDerivedData();
    };
```

```
                    ─── Excerpt from src/body.cpp ───
/**
 * Inline function that creates a transform matrix from a position
 * and orientation.
 */
static inline void _calculateTransformMatrix(Matrix4 &transformMatrix,
                                             const Vector3 &position,
                                             const Quaternion &orientation)
{
    transformMatrix.data[0] = 1-2*orientation.j*orientation.j-
        2*orientation.k*orientation.k;
```

```
        transformMatrix.data[1] = 2*orientation.i*orientation.j -
            2*orientation.r*orientation.k;
        transformMatrix.data[2] = 2*orientation.i*orientation.k +
            2*orientation.r*orientation.j;
        transformMatrix.data[3] = position.x;

        transformMatrix.data[4] = 2*orientation.i*orientation.j +
            2*orientation.r*orientation.k;
        transformMatrix.data[5] = 1-2*orientation.i*orientation.i-
            2*orientation.k*orientation.k;
        transformMatrix.data[6] = 2*orientation.j*orientation.k -
            2*orientation.r*orientation.i;
        transformMatrix.data[7] = position.y;

        transformMatrix.data[8] = 2*orientation.i*orientation.k -
            2*orientation.r*orientation.j;
        transformMatrix.data[9] = 2*orientation.j*orientation.k +
            2*orientation.r*orientation.i;
        transformMatrix.data[10] = 1-2*orientation.i*orientation.i-
            2*orientation.j*orientation.j;
        transformMatrix.data[11] = position.z;
}
void RigidBody::calculateDerivedData()
{
    // Calculate the transform matrix for the body.
    _calculateTransformMatrix(transformMatrix, position, orientation);
}
```

Later we will add calls to additional calculations to this method.

10.2 Newton 2 for Rotation

In Newton's second law of motion, we saw that the change in velocity depends on a force acting on the object and the object's mass:

$$\ddot{p} = m^{-1}f$$

For rotation we have a very similar law. The change in angular velocity depends on two things: we have torque τ rather than force, and the moment of inertia I rather than mass:

$$\ddot{\theta} = I^{-1}\tau$$

Let's look at these two in more depth.

10.2.1 Torque

Torque (also sometimes called "moments") can be thought of as a twisting force. You may be familiar with a car that has a lot of torque: it can apply a great deal of turning force to the wheels. An engine that can generate a lot of torque will be better at accelerating the spin of the wheels. If the car has poor tires, this will leave a big black mark on the road and a lot of smoke in the air; with appropriate grip, this rotation will be converted into forward acceleration. In either case the torque is spinning the wheels, and the forward motion is a secondary effect caused by the tires gripping the road.

In fact torque is slightly different from force. We can turn a force into a torque—that is, a straight push or pull into a turning motion. Imagine turning a stiff nut with a wrench: you turn the nut by pushing or pulling on the handle of the wrench. When you turn up the volume knob on a stereo, you grip it by both sides and push up with your thumb and down with your finger (if you're right-handed). In either case you are applying a force and getting angular motion as a result.

The angular acceleration depends on the size of the force you exert and how far from the turning point you apply it. Take the wrench and nut example: you can undo the nut if you exert more force onto the wrench or if you push farther along the handle (or use a longer-handled wrench). When turning a force into a torque, the size of the force is important, as is the distance from the axis of rotation.

The equation that links force and torque is

$$\boldsymbol{\tau} = \boldsymbol{p}_f \times \boldsymbol{f} \qquad\qquad [10.1]$$

where $\boldsymbol{f}$ is the force being applied, and $\boldsymbol{p}_f$ is the point at which the force is being applied, relative to the origin of the object (i.e., its center of mass, for our purposes).

Every force that applies to an object will generate a corresponding torque. Whenever we apply a force to a rigid body, we need to use it in the way we have so far: to perform a linear acceleration. We will additionally need to use it to generate a torque. If you look at equation 10.1, you may notice that any force applied so that $\boldsymbol{f}$ and $\boldsymbol{p}_f$ are in the same direction will have zero torque. Geometrically this is equivalent to saying that if the extended line of the force passes through the center of mass, then no torque is generated. Figure 10.1 illustrates this. We'll return to this property in section 10.3 when we combine all the forces and torques.

In three dimensions it is important to notice that a torque needs to have an axis. We can apply a turning force about any axis we choose. So far we've considered cases such as the volume knob or nut where the axis is fixed. For a freely rotating object, however, the torque can act to turn the object about any axis. We give torques in a scaled axis representation:

$$\boldsymbol{\tau} = a\widehat{\boldsymbol{d}}$$

where a is the magnitude of the torque and $\widehat{\boldsymbol{d}}$ is a unit-length vector in the axis around which the torque applies. We always consider that torques act clockwise when looking
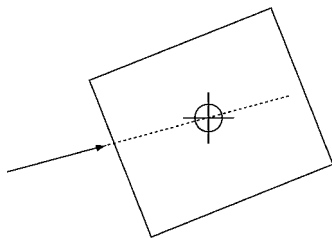
FIGURE 10.1   A force generating zero torque.

in the direction of their axis. To get a counterclockwise torque, we simply flip the sign of the axis.

Equation 10.1 provides our torque in the correct format: the torque is the vector product of the force (which includes its direction and magnitude) and the position of its application.

10.2.2   THE MOMENT OF INERTIA

So we have torque—the rotational equivalent to force. Now we come to the moment of inertia: roughly the rotational equivalent of mass.

The *moment of inertia* of an object is a measure of how difficult it is to change that object's rotation speed. Unlike mass, however, it depends on *how* you spin the object.

Take a long stick like a broom handle and twirl it. You have to put a reasonable amount of effort into getting it twirling. Once it is twirling, you likewise have to apply a noticeable braking force to stop it again. Now stand it on end on the ground and you can get it spinning lengthwise quite easily with two fingers. And you can very easily stop its motion.

For any axis on which you spin an object, it may have a different moment of inertia. The moment of inertia depends on the mass of the object and the distance of that mass from the axis of rotation. Imagine the stick being made up of lots of particles; twirling the stick in the manner of a majorette involves accelerating particles that lie a long way from the axis of rotation. In comparison to twirling the stick lengthwise, the particles of the stick are a long way from the axis. The inertia will therefore be greater, and the stick will be more difficult to rotate.

We can calculate the moment of inertia about an axis in terms of a set of particles in the object:

$$I_a = \sum_{i=1}^{n} m_i d_{p_i \to a}^2$$

where n is the number of particles, $d_{p_i \to a}$ is the distance of particle i from the axis of rotation a, and I_a is the moment of inertia about that axis. You may also see this

equation in terms of an infinite number of particles, using an integral. For almost all applications, however, you can get away with splitting an object into particles and using the sum. This is particularly useful when trying to calculate the moment of inertia of an unusual-shape object. We'll return to the moments of inertia of different objects later in the section.

Clearly we can't use a single value for the moment of inertia as we did for mass. It depends completely on the axis we choose. About any particular axis, we have only one moment of inertia, but there are any number of axes we could choose. Fortunately the physics of rigid bodies means we don't need to have an unlimited number of different values either. We can compactly represent all the different values in a matrix called the "inertia tensor."

Before I describe the inertia tensor in more detail, it is worth getting some terminology clear. The moments of inertia for an object are normally *represented* as an inertia tensor. However, the two terms are somewhat synonymous in physics engine development. The "tensor" bit also causes confusion. A *tensor* is simply a generalized version of a matrix. Whereas vectors can be thought of as a one-dimensional array of values and matrices as a two-dimensional array, tensors can have any number of dimensions. Thus both a vector and a matrix are tensors.

Although the inertia tensor is called a tensor, for our purposes it is always two-dimensional. In other words, it is always just a matrix. It is sometimes called the "mass matrix," and we could call it the "inertia matrix," I suppose, but it's not a term that I've heard used. For most of this book I'll just talk about the *inertia tensor*, meaning the matrix representing all the moments of inertia of an object; this follows the normal idiom of game development.

The inertia tensor in three dimensions is a 3×3 matrix that is characteristic of a rigid body (in other words, we keep an inertia tensor for each body, just as each body has its own mass). Along the leading diagonals the tensor has the moment of inertia about each of its axes—X, Y, and Z:

$$\begin{bmatrix} I_x & & \\ & I_y & \\ & & I_z \end{bmatrix}$$

where I_x is the moment of inertia of the object about its X axis *through its center of mass*; similarly for I_y and I_z.

The remaining entries don't hold moments of inertia. They are called "products of inertia" and are defined in this way:

$$I_{ab} = \sum_{i=1}^{n} m_i a_{p_i} b_{p_i}$$

where a_{p_i} is the distance of particle i from the center of mass of the object, in the direction of a. We use this to calculate I_{xy}, I_{xz}, and I_{yz}. In the case of I_{xy}, we get

$$I_{xy} = \sum_{i=1}^{n} m_i x_{p_i} y_{p_i}$$

where x_{p_i} is the distance of the particle from the center of mass in the X axis direction; similarly for y_{p_i} in the Y axis direction. Using the scalar products of vectors we get

$$I_{xy} = \sum_{i=1}^{n} m_i (\boldsymbol{p}_i \cdot \boldsymbol{x})(\boldsymbol{p}_i \cdot \boldsymbol{y})$$

Note that, unlike for the moment of inertia, each particle can contribute a negative value to this sum. In the moment of inertia calculation, the distance was squared, so its contribution is always positive. It is entirely possible to have a non-positive total product of inertia. Zero values are particularly common for many different shaped objects.

It is difficult to visualize what the product of inertia *means* either in geometrical or mathematical terms. It represents the tendency of an object to rotate in a direction different from the direction in which the torque is being applied. You may have seen this in the behavior of a child's top. You start by spinning it in one direction, but it jumps upside down and spins on its head almost immediately.

For a freely rotating object, if you apply a torque, you will not always get rotation about the same axis to which you applied the torque. This is the effect that gyroscopes are based on: they resist falling over because they transfer any gravity-induced falling rotation back into the opposite direction to stand up straight once more. The products of inertia control this process: the transfer of rotation from one axis to another.

We place the products of inertia into our inertia tensor to give the final structure:

$$I = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix} \qquad [10.2]$$

The mathematician Euler gave the rotational version of Newton's second law of motion in terms of this structure:

$$\boldsymbol{\tau} = I\ddot{\boldsymbol{\theta}}$$

which gives us the angular acceleration in terms of the torque applied:

$$\ddot{\boldsymbol{\theta}} = I^{-1}\boldsymbol{\tau} \qquad [10.3]$$

where I^{-1} is the inverse of the inertia tensor, performed using a regular matrix inversion.

Note that because of the presence of the products of inertia, the direction of the torque vector $\boldsymbol{\tau}$ is not necessarily the same as the angular acceleration vector $\ddot{\boldsymbol{\theta}}$. If the products of inertia are all zero,

$$I = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

and the torque vector is in one of the principal axis directions—X, Y, or Z—then the acceleration *will* be in the direction of the torque.

Many shapes have easy formulae for calculating their inertia tensor. A rectangular block, for example, of mass m and dimensions d_x, d_y, and d_z aligned along the X, Y, and Z axes, respectively, has an inertia tensor of

$$I = \begin{bmatrix} \frac{1}{12}m(d_y^2 + d_z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d_x^2 + d_z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d_x^2 + d_y^2) \end{bmatrix}$$

A list of some other inertia tensors for common shapes is provided in appendix A.

The Inverse Inertia Tensor

For exactly the same reasons as we saw for mass, we will store the inverse inertia tensor rather than the raw inertia tensor. The rigid body has an additional member added, a `Matrix3` instance:

```
────────── Excerpt from include/cyclone/body.h ──────────
class RigidBody
    {
        // ... Other RigidBody code as before ...

        /**
         * Holds the inverse of the body's inertia tensor. The inertia
         * tensor provided must not be degenerate (that would mean
         * the body had zero inertia for spinning along one axis).
         * As long as the tensor is finite, it will be invertible.
         * The inverse tensor is used for similar reasons as those
         * for the use of inverse mass.
         *
         * The inertia tensor, unlike the other variables that define
         * a rigid body, is given in body space.
         *
         * @see inverseMass
         */
```

```
        Matrix3 inverseInertiaTensor;
    };
```

Having the inverse at hand allows us to calculate the angular acceleration directly from equation 10.3 without performing the inverse operation each time. When setting up a rigid body, we can start with a regular inertia tensor, then call the `inverse` function of the matrix, and set the rigid body's inverse inertia tensor to get the result:

```
────────── Excerpt from src/body.cpp ──────────
void RigidBody::setInertiaTensor(const Matrix3 &inertiaTensor)
{
    inverseInertiaTensor.setInverse(inertiaTensor);
    _checkInverseInertiaTensor(inverseInertiaTensor);
}
```

10.2.3   The Inertia Tensor in World Coordinates

There is still one subtle complication to address before we can leave the inertia tensor. Throughout the discussion of moments of inertia I have deliberately not distinguished between the object's local coordinates and the game's world coordinates. Consider the example in figure 10.2. In the first example the object's local axes are in the same direction as the world's axes. If we apply a torque about the X axis, then we will get the same moment of inertia whether we work in local or world coordinates.

In the second part of the figure, the object has rotated. Now whose X axis do we need to use? In fact the torque is expressed in world coordinates, so the rotation will depend on the moment of inertia of the object about the world's X axis. The inertia tensor is defined in terms of the object's axis, however. It is constant: we don't change the inertia tensor each time the object moves.
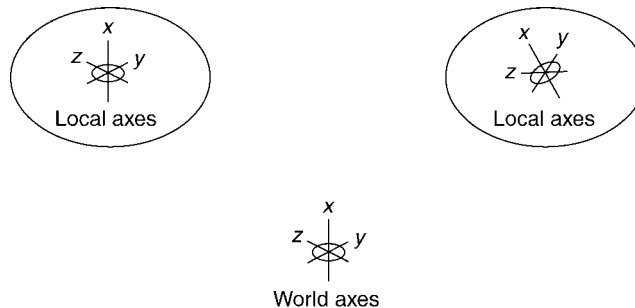
FIGURE 10.2   The moment of inertia is local to an object.

In fact, in the acceleration equation

$$\ddot{\boldsymbol{\theta}} = I^{-1}\boldsymbol{\tau}$$

the torque $\boldsymbol{\tau}$ and the resulting angular acceleration $\ddot{\boldsymbol{\theta}}$ are both given relative to the world axes. So the inertia tensor we need should also be given in world coordinates.

We don't want to have to recalculate the inertia tensor by summing masses at each frame, so we need a simpler way to get the inertia tensor in world coordinates. We can achieve this by creating a new derived quantity: the inverse inertia tensor in world coordinates. At each frame we can apply a change of basis transformation to convert the constant inertia tensor in object coordinates into the corresponding matrix in world coordinates.

As with the transform matrix we add an update to recalculate the derived quantity at each frame. It gets put together in this way:

─────── **Excerpt from src/body.cpp** ───────

```cpp
/**
 * Internal function to do an inertia tensor transform by a quaternion.
 * Note that the implementation of this function was created by an
 * automated code generator and optimizer.
 */
static inline void _transformInertiaTensor(Matrix3 &iitWorld,
                                           const Quaternion &q,
                                           const Matrix3 &iitBody,
                                           const Matrix4 &rotmat)
{
    real t4 = rotmat.data[0]*iitBody.data[0]+
        rotmat.data[1]*iitBody.data[3]+
        rotmat.data[2]*iitBody.data[6];
    real t9 = rotmat.data[0]*iitBody.data[1]+
        rotmat.data[1]*iitBody.data[4]+
        rotmat.data[2]*iitBody.data[7];
    real t14 = rotmat.data[0]*iitBody.data[2]+
        rotmat.data[1]*iitBody.data[5]+
        rotmat.data[2]*iitBody.data[8];
    real t28 = rotmat.data[4]*iitBody.data[0]+
        rotmat.data[5]*iitBody.data[3]+
        rotmat.data[6]*iitBody.data[6];
    real t33 = rotmat.data[4]*iitBody.data[1]+
        rotmat.data[5]*iitBody.data[4]+
        rotmat.data[6]*iitBody.data[7];
    real t38 = rotmat.data[4]*iitBody.data[2]+
        rotmat.data[5]*iitBody.data[5]+
        rotmat.data[6]*iitBody.data[8];
```

```
        real t52 = rotmat.data[8]*iitBody.data[0]+
            rotmat.data[9]*iitBody.data[3]+
            rotmat.data[10]*iitBody.data[6];
        real t57 = rotmat.data[8]*iitBody.data[1]+
            rotmat.data[9]*iitBody.data[4]+
            rotmat.data[10]*iitBody.data[7];
        real t62 = rotmat.data[8]*iitBody.data[2]+
            rotmat.data[9]*iitBody.data[5]+
            rotmat.data[10]*iitBody.data[8];

        iitWorld.data[0] = t4*rotmat.data[0]+
            t9*rotmat.data[1]+
            t14*rotmat.data[2];
        iitWorld.data[1] = t4*rotmat.data[4]+
            t9*rotmat.data[5]+
            t14*rotmat.data[6];
        iitWorld.data[2] = t4*rotmat.data[8]+
            t9*rotmat.data[9]+
            t14*rotmat.data[10];
        iitWorld.data[3] = t28*rotmat.data[0]+
            t33*rotmat.data[1]+
            t38*rotmat.data[2];
        iitWorld.data[4] = t28*rotmat.data[4]+
            t33*rotmat.data[5]+
            t38*rotmat.data[6];
        iitWorld.data[5] = t28*rotmat.data[8]+
            t33*rotmat.data[9]+
            t38*rotmat.data[10];
        iitWorld.data[6] = t52*rotmat.data[0]+
            t57*rotmat.data[1]+
            t62*rotmat.data[2];
        iitWorld.data[7] = t52*rotmat.data[4]+
            t57*rotmat.data[5]+
            t62*rotmat.data[6];
        iitWorld.data[8] = t52*rotmat.data[8]+
            t57*rotmat.data[9]+
            t62*rotmat.data[10];
}
void RigidBody::calculateDerivedData()
{
    // Calculate the inertiaTensor in world space.
    _transformInertiaTensor(inverseInertiaTensorWorld,
        orientation,
        inverseInertiaTensor,
```

```
        transformMatrix);
}
```

Note particularly that the change of basis transform from section 9.4.6 is optimized into one operation.

When we transform the inertia tensor, we are only interested in the rotational component of the object's transform. It doesn't matter where the object is in space, but only which direction it is oriented in. The code therefore treats the 4×3 transform matrix as if it were a 3×3 matrix (i.e., a rotation matrix only). Together these two optimizations make for considerably faster code.

So at each frame we calculate the transform matrix, transform the inverse inertia tensor into world coordinates, and then perform the rigid-body integration with this transformed version. Before we look at the code to perform the final integration, we need to examine how a body reacts to a whole series of torques and forces (with their corresponding torque components).

10.3 D'Alembert for Rotation

Just as we have an equivalent of Newton's second law of motion, we can also find a rotational version of D'Alembert's principle. Recall that D'Alembert's principle allows us to accumulate a whole series of forces into one single force, and then apply just this one force. The effect of the one accumulated force is identical to the effect of all its component forces. We take advantage of this by simply adding together all the forces applied to an object, and then only calculating its acceleration once, based on the resulting total.

The same principle applies to torques: the effect of a whole series of torques is equal to the effect of a single torque that combines them all. We have

$$\tau = \sum_i \tau_i$$

where τ_i is the ith torque.

There is a complication, however. We saw earlier in the chapter that an off-center force can be converted into torques. To get the correct set of forces and torques we need to take into account this calculation.

Another consequence of D'Alembert's principle is that we can accumulate the torques caused by forces in exactly the same way as we accumulate any other torques. Note that we *cannot* merely accumulate the forces and then take the torque equivalent of the resulting force. We could have two forces (like the finger and thumb on a volume dial) that cancel each other out as linear forces but combine to generate a large torque.

So we have two accumulators: one for forces and another for torques. Each force applied is added to both the force and torque accumulator, where its torque is calculated by

$$\boldsymbol{\tau} = \boldsymbol{p}_f \times \boldsymbol{f}$$

(i.e., equation 10.1, which we saw earlier). For each torque applied we accumulate just the torque (torques have no corresponding force component).

Some forces, such as gravity, will always apply to the center of mass of an object. In this case there is no point trying to work out their torque component because they can never induce rotation. We allow this in our engine by providing a third route: adding a force with no position of application. In this case we merely add the force to the force accumulator and bypass torque accumulation. In code this looks like this:

Excerpt from `include/cyclone/body.h`

```cpp
class RigidBody
{
    // ... Other RigidBody code as before ...

    /**
     * Adds the given force to the center of mass of the rigid body.
     * The force is expressed in world coordinates.
     *
     * @param force The force to apply.
     */
    void addForce(const Vector3 &force);
};
```

Excerpt from `src/body.cpp`

```cpp
void RigidBody::addForce(const Vector3 &force)
{
    forceAccum += force;
}
```

In addition, when we perform our per-frame setup of the body, we zero the torque:

Excerpt from `include/cyclone/body.h`

```cpp
class RigidBody
{
    // ... Other RigidBody code as before ...

};
```

Excerpt from `src/body.cpp`

```cpp
void RigidBody::integrate(real duration)
{
    // Clear accumulators.
    clearAccumulators();
}
```

```
void RigidBody::clearAccumulators()
{
    forceAccum.clear();
    torqueAccum.clear();
}
```

An important caution here concerns the location of the application of a force. It should be expressed in world coordinates. If you have a spring attached at a fixed point on an object, you need to recalculate the position of the attachment point at each frame. You can do this simply by transforming the object coordinates' position by the transform matrix, to get a position in world coordinates. Because this is such a useful thing to be able to do, I provide an additional force-accumulation method to support it:

—————— **Excerpt from include/cyclone/body.h** ——————

```
class RigidBody
{
    // ... Other RigidBody code as before ...

    /**
     * Adds the given force to the given point on the rigid body.
     * The direction of the force is given in world coordinates,
     * but the application point is given in body space. This is
     * useful for spring forces, or other forces fixed to the
     * body.
     *
     * @param force The force to apply.
     *
     * @param point The location at which to apply the force, in
     * body coordinates.
     */
    void addForceAtBodyPoint(const Vector3 &force,
                             const Vector3 &point);
};
```

—————— **Excerpt from src/body.cpp** ——————

```
void RigidBody::addForceAtBodyPoint(const Vector3 &force,
                                    const Vector3 &point)
{
    // Convert to coordinates relative to the center of mass.
    Vector3 pt = getPointInWorldSpace(point);
    addForceAtPoint(force, pt);
}
```

Be careful: The direction of the force is expected in world coordinates, whereas the application point is expected in object coordinates! This matches the way that these calculations are normally performed, but you could create yet another version that transforms both the force and the position into world coordinates. In this case, be careful with the transformation of the force: it should be rotated only; it shouldn't be transformed by the full 4×3 matrix (which adds the offset position to the vector).

10.3.1 Force Generators

We need to update the force generators we created for particles to work with rigid bodies. In particular they may have to be able to apply a force at a specific point on the rigid body. If this isn't at the center of mass, we'll be generating both a force and a torque for our rigid body, as we saw in the previous section.

This is the logic of not having a force generator return just a single force vector: we won't know where the force is applied. Instead we allow the force generator to apply a force in whatever way it wants. We can create force generators that call the method to apply a force at a point other than the body's center of mass, or they may just apply a force to the center of mass.

This means that the gravity force generator is almost the same. It is changed only to accept the rigid-body type rather than a particle:

<div align="center">Excerpt from include/cyclone/fgen.h</div>

```
/**
 * A force generator that applies a gravitational force. One instance
 * can be used for multiple rigid bodies.
 */
class Gravity : public ForceGenerator
{
    /** Holds the acceleration due to gravity. */
    Vector3 gravity;

public:

    /** Creates the generator with the given acceleration. */
    Gravity(const Vector3 &gravity);

    /** Applies the gravitational force to the given rigid body. */
    virtual void updateForce(RigidBody *body, real duration);
};
```

<div align="center">Excerpt from src/fgen.cpp</div>

```
void Gravity::updateForce(RigidBody* body, real duration)
{
    // Check that we do not have infinite mass
```

```
        if (!body->hasFiniteMass()) return;

        // Apply the mass-scaled force to the body
        body->addForce(gravity * body->getMass());
}
```

The spring force generator now needs to know where the spring is attached on each object, and it should generate an appropriate force with its application point.

```
                    ──── Excerpt from include/cyclone/fgen.h ────
/**
 * A force generator that applies a Spring force.
 */
class Spring : public ForceGenerator
{
    /** The point of connection of the spring, in local coordinates. */
    Vector3 connectionPoint;

    /**
     * The point of connection of the spring to the other object,
     * in that object's local coordinates.
     */
    Vector3 otherConnectionPoint;

    /** The particle at the other end of the spring. */
    RigidBody *other;

    /** Holds the spring constant. */
    real springConstant;

    /** Holds the rest length of the spring. */
    real restLength;

public:

    /** Creates a new spring with the given parameters. */
    Spring(const Vector3 &localConnectionPt,
           RigidBody *other,
           const Vector3 &otherConnectionPt,
           real springConstant,
           real restLength);

    /** Applies the spring force to the given particle. */
```

```
    virtual void updateForce(RigidBody *body, real duration);
};
```

```cpp
void Spring::updateForce(RigidBody* body, real duration)
{
    // Calculate the two ends in world space.
        Vector3 lws = body->getPointInWorldSpace(connectionPoint);
        Vector3 ows = other->getPointInWorldSpace(otherConnectionPoint);

    // Calculate the vector of the spring.
    Vector3 force = lws - ows;

    // Calculate the magnitude of the force.
    real magnitude = force.magnitude();
    magnitude = real_abs(magnitude - restLength);
    magnitude *= springConstant;

    // Calculate the final force and apply it.
    force.normalize();
    force *= -magnitude;
    body->addForceAtPoint(force, lws);
}
```

Torque Generators

We could follow the lead of the force generators and create a set of torque generators. They fit into the same force generator structure we've used so far: calling the rigid body's addTorque method.

You can use this to constantly drive a rotating object, such as a set of fan blades or the wheels of a car.

10.4   THE RIGID-BODY INTEGRATION

So, we're finally in the position to write the integration routine that will update the position and orientation of a rigid body based on its forces and torques. It will have the same format as the integration for a particle, with the rotation components added. To correspond with the linear case, we add an additional data member to the rigid body to control angular velocity damping—the amount of angular velocity the body loses each second:

```cpp
class RigidBody
{
```

```
        // ... Other RigidBody code as before ...

        /**
         * Holds the amount of damping applied to angular
         * motion. Damping is required to remove energy added
         * through numerical instability in the integrator.
         */
        real angularDamping;
    };
```

Just as we saw with linear velocity, the angular velocity is updated with the equation

$$\dot{\theta}' = \dot{\theta}(d_a)^t + \ddot{\theta}t$$

where d_a is the angular damping coefficient. The complete integration routine now looks like this:

———— **Excerpt from src/body.cpp** ————

```
void RigidBody::integrate(real duration)
{
    // Calculate linear acceleration from force inputs.
    lastFrameAcceleration = acceleration;
    lastFrameAcceleration.addScaledVector(forceAccum, inverseMass);

    // Calculate angular acceleration from torque inputs.
    Vector3 angularAcceleration =
        inverseInertiaTensorWorld.transform(torqueAccum);

    // Adjust velocities
    // Update linear velocity from both acceleration and impulse.
    velocity.addScaledVector(lastFrameAcceleration, duration);

    // Update angular velocity from both acceleration and impulse.
    rotation.addScaledVector(angularAcceleration, duration);

    // Impose drag.
    velocity *= real_pow(linearDamping, duration);
    rotation *= real_pow(angularDamping, duration);

    // Adjust positions
    // Update linear position.
    position.addScaledVector(velocity, duration);

    // Update angular position.
```

```
    orientation.addScaledVector(rotation, duration);

    // Impose drag.
    velocity *= real_pow(linearDamping, duration);
    rotation *= real_pow(angularDamping, duration);

    // Normalize the orientation, and update the matrices with the new
    // position and orientation.
    calculateDerivedData();

    // Clear accumulators.
    clearAccumulators();
}
```

10.5 Summary

The physics of angular motion is very similar to the physics of linear motion we met in chapter 3. In the same way that force is related to acceleration via mass, we've seen that torque is related to angular acceleration via moment of inertia. The physics is similar, but in each case the mathematics is more complex and the implementation longer. The vector position found its angular correspondence in the quaternion for orientation, and the scalar-valued mass became an inertia tensor.

The last two chapters have therefore been considerably more difficult than those at the start of the book. If you have followed through to get a rigid-body physics engine, then you can be proud of yourself. There are significant limits to what we've built so far (notably we haven't brought collisions into the new engine) that we'll spend the rest of the book resolving, but there are also a lot of great things that you can do with what we have. Chapter 11 introduces some applications for our current engine.
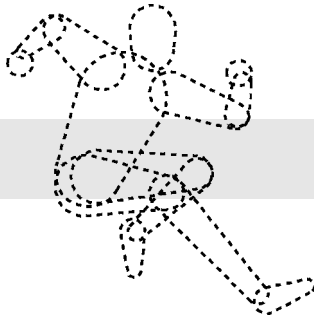
11

THE RIGID-BODY PHYSICS ENGINE

O ur physics engine is now capable of simulating full rigid bodies in full 3D. The spring forces and other force generators will work with this approach, but the hard constraints we met in chapter 7 will not. We will look at collision detection in the next part (part IV) of the book, and then return to full 3D constraints in part V.

Even without hard constraints, there is still a lot we can do. This chapter looks at two applications of physics that don't rely on hard constraints for their effects: boats and aircraft. We'll build a flight simulator and a boat model. Adding the aerodynamics from the flight simulator allows us to build a sailing simulation.

11.1 OVERVIEW OF THE ENGINE

The rigid-body physics engine has two components:

1. The rigid bodies themselves keep track of their position, movement, and mass characteristics. To set up a simulation we need to work out what rigid bodies are needed and then set their initial position, orientation, and velocities (both linear and angular). We also need to set their inverse mass and inverse inertia tensor. The acceleration of an object due to gravity is also held in the rigid body. (This could be removed and replaced by a force, if you so desire.)

2. The force generators are used to keep track of forces that apply over several frames of the game.

We have removed the contact resolution system from the mass-aggregate system (it will be reintroduced in parts IV and V).

We can use the system we introduced in chapter 8 to manage the objects to be simulated. In this case, however, they are rigid bodies rather than particles. The World structure is modified accordingly:

```
—————— Excerpt from include/cyclone/world.h ——————
/**
 * The world represents an independent simulation of physics. It
 * keeps track of a set of rigid bodies, and provides the means to
 * update them all.
 */
class World
{
    /**
     * Holds a single rigid body in a linked list of bodies.
     */
    struct BodyRegistration
    {
        RigidBody *body;
        BodyRegistration * next;
    };

    /**
     * Holds the head of the list of registered bodies.
     */
    BodyRegistration *firstBody;

};
```

As before, at each frame the startFrame method is first called, which sets up each object by zeroing its force and torque accumulators and calculating its derived quantities:

```
—————— Excerpt from include/cyclone/world.h ——————
/**
 * The world represents an independent simulation of physics. It
 * keeps track of a set of rigid bodies, and provides the means to
 * update them all.
 */
class World
{
    // ... other World data as before ...
    /**
     * Initializes the world for a simulation frame. This clears
     * the force and torque accumulators for bodies in the
     * world. After calling this, the bodies can have their forces
```

```
 * and torques for this frame added.
 */
void startFrame();
};
```

```
                 ──── Excerpt from src/world.cpp ────
void World::startFrame()
{
    BodyRegistration *reg = firstBody;
    while (reg)
    {
        // Remove all forces from the accumulator.
        reg->body->clearAccumulators();
        reg->body->calculateDerivedData();

        // Get the next registration.
        reg = reg->next;
    }
}
```

Again, additional forces can be applied after calling this method.

To execute the physics, the runPhysics method is called. This calls all the force generators to apply their forces and then performs the integration of all objects:

```
              ──── Excerpt from include/cyclone/world.h ────
/**
 * The world represents an independent simulation of physics.  It
 * keeps track of a set of rigid bodies, and provides the means to
 * update them all.
 */
class World
{
    // ... other World data as before ...
    /**
     * Processes all the physics for the world.
     */
    void runPhysics(real duration);
};
```

```
                 ──── Excerpt from src/world.cpp ────
void World::runPhysics(real duration)
{
    // First apply the force generators
    //registry.updateForces(duration);
```

```
    // Then integrate the objects
    BodyRegistration *reg = firstBody;
    while (reg)
    {
        // Remove all forces from the accumulator
        reg->body->integrate(duration);

        // Get the next registration
        reg = reg->next;
    }
}
```

It no longer calls the collision detection system.

The calls to startFrame and runPhysics can occur in the same place in the game loop.

Notice that I've made an additional call to the updateTransform method of each object. It may have moved during the update (and in later sections during collision resolution), so its transform matrix needs updating before it is rendered. Each object is then rendered in turn using the rigid body's transform.

11.2   Using the Physics Engine

Both sample programs for this physics engine use aerodynamics. We will create a new force generator that can fake some important features of flight aerodynamics, enough to produce a basic flight simulator suitable for use in a flight action game. We will use the same generator to drive a sail model for a sailing simulator.

11.2.1   A Flight Simulator

There is no need for contact physics in a flight simulator, except with the ground, of course. Many flight simulators assume that if you hit something in an airplane, then it's all over: a crash animation plays and the player starts again. This makes it a perfect exercise ground for our current engine.

The dynamics of an aircraft are generated by the way air flows over its surfaces. (This includes the surfaces that don't move relative to the center of mass, such as the fuselage, and control surfaces that can be made to move or change shape, such as the wings and rudder.) The flow of air causes forces to be generated. Some, like drag, act in the same direction that the aircraft is moving in. The most important force, lift, acts at right angles to the flow of air. As the aircraft's surfaces move at different angles through the air, the proportion of each kind of force can change dramatically. If the wing is slicing through the air, it generates lift; but if it is moving vertically through

the air, then it generates no lift. We'd like to be able to capture this kind of behavior in a force generator that can produce sensible aerodynamic forces.

The Aerodynamic Tensor

To model the aerodynamic forces properly is very complex. The behavior of a real aircraft depends on the fluid dynamics of air movement. This is a horrendously complex discipline involving mathematics well beyond the scope of this book. To create a truly realistic flight simulator involves some specialized physics that I don't want to venture into.

To make our life easier I will use a simplification: the "aerodynamic tensor." The aerodynamic tensor is a way of calculating the total force that a surface of the airplane is generating based only on the speed that the air is moving over it.

The tensor is a matrix: a 3×3 matrix, exactly as we used for the inertia tensor. We start with a wind speed vector and transform it using the tensor to give a force vector:

$$\boldsymbol{f}_a = A\boldsymbol{v}_w$$

where $\boldsymbol{f}_a$ is the resulting force, A is the aerodynamic tensor, and $\boldsymbol{v}_w$ is the velocity of the air. Just as with the inertia tensor, we have to be careful of coordinates here. The velocity of the air and the resulting force are both expressed in world coordinates, but the aerodynamic tensor is in object coordinates. Again we need to change the basis of the tensor at each frame before applying this function.

To fly the plane we can implement control surfaces in one of two ways. The first, and most accurate, is to have two tensors representing the aerodynamic characteristics when the surface is at its two extremes. At each frame the current position of the control surface is used to blend the two tensors to create a tensor for the current surface position.

In practice three tensors are sometimes needed, to represent the two extremes plus the "normal" position of the control surface (which often has a quite different, and not intermediate, behavior). For example, a wing with its aileron (the control surface on the back of each wing) in line with the wing produces lots of lift and only a modest amount of drag. With the aileron out of this position, either up or down, the drag increases dramatically, but the lift can be boosted or cut (depending on whether it is up or down).

The second approach is to actually tilt the whole surface slightly. We can do this by storing an orientation for the aerodynamic surface and allowing the player to directly control some of this orientation. To simulate the aileron on the wing the player might be effectively tilting the whole wing. As the wing changes orientation, the air flow over it will change and its one aerodynamic tensor will generate correspondingly different forces.

The Aerodynamic Surface

We can implement an aerodynamic force generator using this technique. The force generator is created with an aerodynamic tensor, and it is attached to the rigid body

at a given point. This is the point at which all its force will be felt. We can attach as many surfaces as we need. The force generator looks like this:

```
                    ─── Excerpt from include/cyclone/fgen.h ───
/**
 * A force generator that applies an aerodynamic force.
 */
class Aero : public ForceGenerator
{
    /**
     * Holds the aerodynamic tensor for the surface in body space.
     */
    Matrix3 tensor;

    /**
     * Holds the relative position of the aerodynamic surface
     * in body coordinates.
     */
    Vector3 position;

    /**
     * Holds a pointer to a vector containing the wind speed of
     * the environment. This is easier than managing a separate
     * wind speed vector per generator and having to update
     * it manually as the wind changes.
     */
    const Vector3* windspeed;

public:
    /**
     * Creates a new aerodynamic force generator with the
     * given properties.
     */
    Aero(const Matrix3 &tensor, const Vector3 &position,
         const Vector3 *windspeed);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

The air velocity is calculated based on two values—the prevailing wind and the velocity of the rigid body. The prevailing wind is a vector containing both the direc-

tion and magnitude of the wind. If the rigid body were not moving, it would still feel this wind. We could omit this value for a flight game that doesn't need to complicate the players' task by adding wind. It will become very useful when we come to model our sailing simulator in the next section, however.

This implementation uses a single tensor only. To implement control surfaces we need to extend this in one of the ways we looked at earlier. I will choose the more accurate approach, with three tensors to represent the characteristics of the surface at the extremes of its operation:

─────── **Excerpt from include/cyclone/fgen.h** ───────

```
/**
 * A force generator with a control aerodynamic surface. This
 * requires three inertia tensors, for the two extremes and the 'resting'
 * position of the control surface.
 * The latter tensor is the one inherited from the base class;
 * the two extremes are defined in this class.
 */
class AeroControl : public Aero
{
    /**
     * The aerodynamic tensor for the surface, when the control is at
     * its maximum value.
     */
    Matrix3 maxTensor;

    /**
     * The aerodynamic tensor for the surface, when the control is at
     * its minimum value.
     */
    Matrix3 minTensor;

    /**
     * The current position of the control for this surface. This
     * should range between -1 (in which case the minTensor value is
     * used) through 0 (where the base-class tensor value is used)
     * to +1 (where the maxTensor value is used).
     */
    real controlSetting;

private:
    /**
     * Calculates the final aerodynamic tensor for the current
     * control setting.
     */
```

```
    Matrix3 getTensor();

public:
    /**
     * Creates a new aerodynamic control surface with the given
     * properties.
     */
    AeroControl(const Matrix3 &base, const Matrix3 &min,
                const Matrix3 &max, const Vector3 &position,
                const Vector3 *windspeed);
    /**
     * Sets the control position of this control. This
     * should range between -1 (in which case the minTensor value is
     * used) through 0 (where the base-class tensor value is used)
     * to +1 (where the maxTensor value is used). Values outside that
     * range give undefined results.
     */
    void setControl(real value);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

Each control surface has an input, wired to the player's (or AI's) control. It ranges from -1 to $+1$, where 0 is considered the "normal" position. The three tensors match these three positions. Two of the three tensors are blended together to form a current aerodynamic tensor for the setting of the surface. This tensor is then converted into world coordinates and used as before.

Putting It Together

On the CD the **flightsim** demo shows this force generator in operation. You control a model aircraft (seen from the ground for a bit of added challenge). The only forces applied to the aircraft are gravity (represented as an acceleration value) and the aerodynamic forces from surface and control surface force generators. Figure 11.1 shows the aircraft in action.

I have used four control surfaces: two wings, a tailplane, and a rudder. The tailplane is a regular surface force generator, with no control inputs (in a real plane the tailplane usually does have control surfaces, but we don't need them). It has the aero-
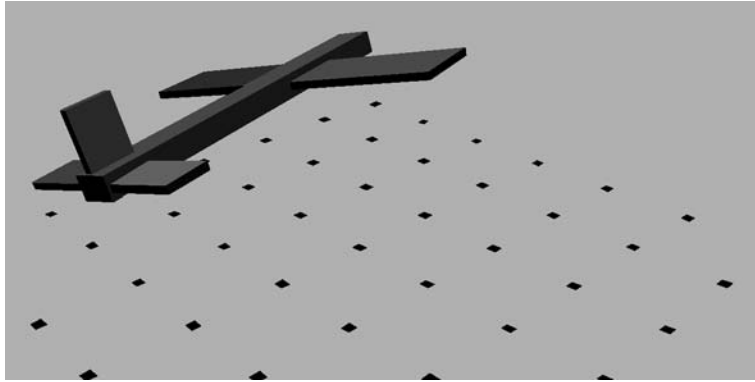
FIGURE 11.1   Screenshot of the **flightsim** demo.

dynamic tensor

$$A = \begin{bmatrix} -0.1 & 0 & 0 \\ 1 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

Each wing has an identical control surface force generator. I have used two so that their control surfaces can be operated independently. They use the following aerodynamic tensors:

$$A_{-1} = \begin{bmatrix} -0.2 & 0 & 0 \\ -0.2 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

$$A_0 = \begin{bmatrix} -0.1 & 0 & 0 \\ 1 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -0.2 & 0 & 0 \\ 1.4 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

for each extreme of the control input. When the player banks the aircraft, both wing controls work in the same direction. When the player rolls, the controls work in opposition.

Finally I have added a rudder—a vertical control surface to regulate the yaw of the aircraft. It has the following tensors:

$$A_{-1} = \begin{bmatrix} -0.1 & 0 & -0.4 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

$$A_0 = \begin{bmatrix} -0.1 & 0 & 0 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -0.1 & 0 & 0.4 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

The surfaces are added to the aircraft in a simple setup function, and the game loop is exactly as we've seen it before. The user input notifies the software as it happens. (This is a function of the OpenGL system we are using to run the demos; in some engines you may have to call a function to explicitly ask for input.) The input directly controls the current values for each control surface.

The full code for the demo can be found on the CD.

11.2.2 A Sailing Simulator

Boat racing is another genre that doesn't require hard constraints, at least in its simplest form. If we want close racing with bumping boats, then we may need to add more complex collision support. For our purpose we'll implement a simple sailing simulator for a single player.

The aerodynamics of the sail is very similar to the aerodynamics we used for flight simulation. We'll come back to the sail-specific setup in a moment, after looking at the floating behavior of the boat.

Buoyancy

What needs revisiting at this point is our buoyancy model. In section 6.2.4 we created a buoyancy force generator to act on a particle. We need to extend this to cope with rigid bodies.

Recall that a submerged shape has a buoyancy that depends on the mass of the water it displaces. If that mass of water is greater than the mass of the object, then the net force will be upward and the object will float. The buoyancy force depends only on the volume of the object that is submerged. We approximated this by treating buoyancy like a spring: as the object is gradually submerged more, the force increases until the object is considered to be completely under water, whereupon the force is
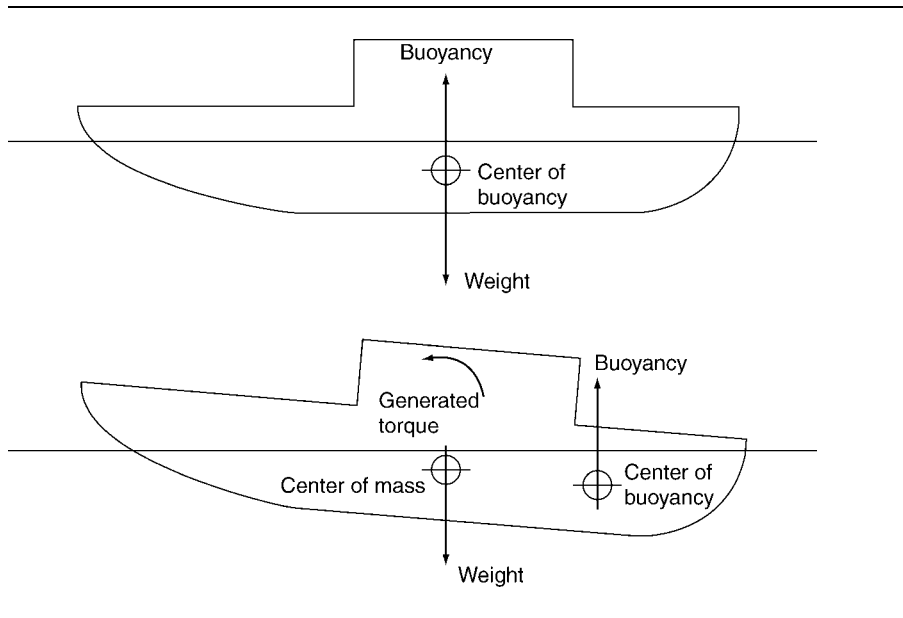
FIGURE 11.2   Different centers of buoyancy.

at its maximum. It doesn't increase with further depth. This is an approximation because it doesn't take into account the shape of the object being submerged.

Originally the force directly acted on the particle. This is fine for representing balls or other regular objects. On a real boat, however, the buoyancy does two jobs: it keeps the boat afloat, and it keeps the boat upright. In other words, if the boat begins to lean over (say a gust of wind catches it), the buoyancy will act to right it.

This tendency to stay upright is a result of the torque component of the buoyancy force. Its linear component keeps the boat afloat, and its torque keeps it vertical. It does this because, unlike in our particle force generator, the buoyancy force doesn't act at the center of gravity.

A submerged part of the boat will have a center of buoyancy, as shown in figure 11.2. The *center of buoyancy* is the point at which the buoyancy force can be thought to be acting. Like the buoyancy force itself, the center of buoyancy is related to the displaced water. The center of mass of the displaced water is the same as the center of buoyancy that it generates.

So, just as the volume of water displaced depends on the shape of the submerged object, so does the center of buoyancy. The farther the center of buoyancy is from the center of mass, the more torque will be generated and the better the boat will be at righting itself. If the center of mass is above the center of buoyancy, then the torque will apply in the opposite direction and the buoyancy will act to topple the boat.

So how do we simulate this in a game? We don't want to get into the messy details of the shape of the water being displaced and finding its center of mass. Instead we

can simply fix the center of buoyancy to the rigid body. In a real boat the center of buoyancy will move around as the boat pitches and rolls and a different volume of water is displaced. Most boats are designed so that this variation is minimized, however. Fixing the center of buoyancy doesn't look odd for most games. It shows itself mostly with big waves, but can be easily remedied, as we'll see later.

Our buoyancy force generator can be updated to take an attachment point; otherwise it is as before:

Excerpt from `include/cyclone/fgen.h`

```
/**
 * A force generator to apply a buoyant force to a rigid body.
 */
class Buoyancy : public ForceGenerator
{
    /**
     * The maximum submersion depth of the object before
     * it generates its maximum buoyancy force.
     */
    real maxDepth;

    /**
     * The volume of the object.
     */
    real volume;

    /**
     * The height of the water plane above y=0. The plane will be
     * parallel to the XZ plane.
     */
    real waterHeight;

    /**
     * The density of the liquid. Pure water has a density of
     * 1000 kg per cubic meter.
     */
    real liquidDensity;

    /**
     * The center of buoyancy of the rigid body, in body coordinates.
     */
    Vector3 centerOfBuoyancy;

public:

    /** Creates a new buoyancy force with the given parameters. */
```

```
    Buoyancy(const Vector3 &cOfB,
        real maxDepth, real volume, real waterHeight,
        real liquidDensity = 1000.0f);


    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

There is nothing to stop us from attaching multiple buoyancy force generators to a boat, to represent different parts of the hull. This allows us to simulate some of the shift in the center of buoyancy. If we have two buoyancy force generators, one at the front (fore) and one at the rear (aft) of a boat, then as it pitches forward and back (through waves, for example), the fore and aft generators will be at different depths in the water and will therefore generate different forces. The highly submerged front of the boat will pitch up rapidly and believably. Without multiple attachments, this wouldn't look nearly as believable and may be obviously inaccurate.

For our sailing simulator we will use a catamaran with two hulls and four buoyancy force generators: one fore and one aft on each hull.

The Sail, Rudder, and Hydrofoils

We will use aerodynamics to provide both the sail and the rudder for our boat. The rudder is like the rudder on the aircraft: it acts to keep the boat going straight (or to turn under the command of the player). On many sailing boats there is both a rudder and a dagger board. The dagger board is a large vertical fin that keeps the boat moving in a straight line and keeps it from easily tipping over when the wind catches the sail. The rudder is a smaller vertical fin that can be tilted for turning. For our needs we can combine the two into one. In fact, in many high-performance sailing boats the two are combined into one structure.

The sail is the main driving force of the boat, converting wind into forward motion. It acts very much like an aircraft wing, turning air flow into lift. In the case of a sailing boat the lift is used to propel the boat forward. There is a misconception that the sail simply "catches" the air and the air drags the boat forward. This can be achieved, certainly, and downwind an extra sail (the spinnaker) is often deployed to increase the aerodynamic drag of the boat and cause it to be pulled along relative to the water. In most cases, however, the sail acts more like a wing than a parachute. In fact, the fastest boats can achieve incredible lift from their sails and travel considerably faster than the wind speed.

Both the rudder and the sail are control surfaces: they can be adjusted to get the best performance. They are both rotated rather than having pop-up control surfaces to modify their behavior (although the sail can have its tension adjusted on some boats). We will therefore implement a force generator for control surfaces using the

second possible adjustment approach from section 11.2.1: rotating the control surface. The force generator looks like this:

```
                    ─── Excerpt from include/cyclone/fgen.h ───
/**
 * A force generator with an aerodynamic surface that can be re-oriented
 * relative to its rigid body. This derives the
 */
class AngledAero : public Aero
{
    /**
     * Holds the orientation of the aerodynamic surface relative
     * to the rigid body to which it is attached.
     */
    Quaternion orientation;

public:
    /**
     * Creates a new aerodynamic surface with the given properties.
     */
    Aero(const Matrix3 &tensor, const Vector3 &position,
        const Vector3 *windspeed);

    /**
     * Sets the relative orientation of the aerodynamic surface relative
     * to the rigid body it is attached to. Note that this doesn't affect
     * the point of connection of the surface to the body.
     */
    void setOrientation(const Quaternion &quat);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

Notice that the generator keeps an orientation for the surface and uses this, in combination with the orientation of the rigid body, to create a final transformation for the aerodynamic surface. There is only one tensor, but the matrix by which it is transformed is now the combination of the rigid body's orientation and the adjustable orientation of the control surface.

Although I won't add them to our example, we could also add wings to the boat: hydrofoils to lift it out of the water. These act just like wings on an aircraft, producing vertical lift. Typically on a hydrofoil boat they are positioned lower than any part of
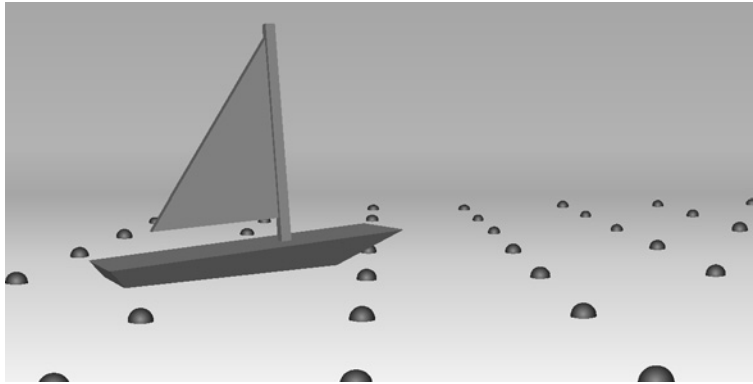
FIGURE 11.3   Screenshot of the **sailboat** demo.

the hull. The lift raises the boat out of the water (whereupon there is no buoyancy force, of course, but no drag from the hull either), and only the hydrofoils remain submerged.

The hydrofoils can be easily implemented as modified surface force generators. The modification needs to make sure that the boat doesn't start flying: it generates no lift once the foil has left the water. In practice a hydrofoil is often designed to produce less lift the higher the boat is out of the water so that the boat rapidly reaches its optimum cruising height. This behavior also wouldn't be difficult to implement; it requires only scaling back the tensor-generated force based on how near the hydrofoil is to the surface of the water.

The Sailing Example

The **sailboat** demo on the CD puts all these bits together. You can control a catamaran on a calm ocean. The orientations of the sail and rudder are the only adjustments you can make. The prevailing wind direction and strength are indicated, as you can see from the screenshot in figure 11.3.

The boat is set up with four buoyancy force generators, a sail, and a rudder. The wind direction changes slowly but randomly over time. It is updated at each frame with a simple recency-weighted random function.

The update of the boat is exactly the same as for the aircraft demo, and user input is also handled as before (see the code on the CD for a complete listing).

11.3 SUMMARY

In this chapter we've met a set of real game examples where our physics engine combines with real-world physics knowledge to produce a believable simulation. In the

case of both sailing and flight we use a simplification of fluid dynamics to quickly and simply generate believable behavior.

The aerodynamic tensor isn't sufficiently accurate for games that intend to simulate flight or sailing accuracy, but is perfectly sufficient for games that are not intended to be realistic.

The situations I chose for this chapter were selected carefully, however, not to embarrass the physics engine. As it stands, our engine is less capable than the mass-aggregate engine we built in part II of this book. To make it truly useful we need to add collisions back in. Unfortunately, with rotations in place this becomes a significantly more complex process than we saw in chapter 7. It is worth taking the time to get it right.

In that spirit, before we consider the physics of collisions again, we'll build the code to detect and report collisions in our game. Part IV of the book does that.

PART IV

Collision Detection

This page intentionally left blank
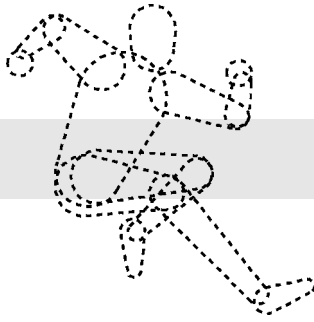
12

COLLISION
DETECTION

I n this chapter and the next we'll take a break from building the physics simulation and look at collision detection.

In chapter 7 we added contacts and collisions to a particle engine, but we removed them to introduce rotating rigid bodies. We're now on the road to having them back in the engine. We could simply introduce contacts and collisions into the engine magically, without worrying about where they came from, just as in chapter 7. If you are working with an existing collision detection library, then you can take this approach and skip to chapter 14. This chapter and the next give an overview of where the contact and collision information comes from and how it is generated.

I will not attempt to produce a comprehensive collision detection system. There are books longer than this one on this one subject alone, and there are many pitfalls and complications that would need discussing.

If you are working with an existing game engine, it is likely to have a collision detection system that you can use. It is still worth reading through the following two chapters, however. Not all rendering engines provide collision detection routines that are inefficient (many use the same geometry as will be drawn, which is a waste of processing time) or don't provide the kind of detailed contact data that the physics system needs.

I will step through one particular approach to collision detection that is useful for smaller games. It is also useful as a jumping-off point into a more complete system and as a way to raise the kinds of issue that are common to all collision detectors.

If you need to build a complete and comprehensive collision detection system, then I'd recommend Ericson [2005] in the same series as this book. It contains many

details on tradeoffs, architecture, and optimization that are invaluable for a robust end product.

12.1 Collision Detection Pipeline

Collision detection can be a very time-consuming process. Each object, in the game may be colliding with any other object, and each such pair has to be checked. If there are hundreds of objects in the game, hundreds of thousands of checks may be needed. And to make things worse, each check has to understand the geometry of the two objects involved, which might consist of thousands of polygons. So to perform a complete collision detection, we may need a huge number of time-consuming checks. This is not possible in the fraction of time we have between frames.

Fortunately there is plenty of room for improvement. The two key problems—having too many possible collisions and having expensive checks—have independent solutions. To reduce the number of checks needed, we can use a two-step process.

1. First we try to find sets of objects that are likely to be in contact with one another, but without being too concerned about whether they actually are. This is typically quite a fast process that uses rules of thumb and specialized data structures to eliminate the vast majority of possible collision checks. It is called "coarse collision detection."

2. Then a second chunk of code looks at the candidate collisions and does the check to determine whether they actually are in contact. This is "fine collision detection." Objects that are in contact are examined to determine exactly where the contact is on each object (needed for rigid bodies), and the normal of collision (which we saw in chapter 7). This is sometimes called "contact generation," and the results can form the input to the physics engine.

The first element is covered in this chapter; the second element will be covered in chapter 13.

To reduce the time taken for each check the geometry is typically simplified. A special geometry just for collision detection is often created, making contact tests far simpler. Collision geometry is discussed in section 13.1.

12.2 Coarse Collision Detection

The first phase of collision detection is tasked with generating a list of full-detail checks that need to be performed.[1] It needs to have the following key features.

---

[1] In fact, we will not create an explicit list as a data structure to pass between the two phases. It is more convenient to have the coarse collision detector simply call the fine collision detector each time it comes across a likely collision. The fine collision detector can then accumulate a real list of actual collisions to pass through to the physics engine.

- It should be conservative. In other words, all the collisions in the game should be contained on the list of checks. The coarse collision detector is allowed to generate checks that end up not being collisions (called "false positives"), but it should not fail to generate checks that would be collisions (called "false negatives").

- It should generate as small a list as possible. In combination with the previous feature, this means that the smallest list it can return is the list of checks that will lead to contacts. In that case the coarse collision detection will be performing a fully accurate collision detection, and there will be no need for further checks. In practice, however, the coarse collision detection usually returns a set of possible checks that contains many false positives.

- It should be as fast as possible. It may be possible to generate close to the optimum number of required checks, but it defeats the object of the coarse collision detector if it takes a long time to do so.

Two broad approaches are used for coarse collision detection: bounding volume hierarchies and spatial data structures. We'll look at these in turn.

12.3 Bounding Volumes

A *bounding volume* is an area of space that is known to contain an object. To represent the volume for coarse collision detection, a simple shape is used, typically a sphere or a box. The shape is made large enough so that the whole object is guaranteed to be inside the shape.

The shape can then be used to perform some simple collision tests. If two objects have bounding volumes that don't touch, then there is no way in which the objects within them can be in contact. Figure 12.1 shows two objects with spherical bounding volumes.

Ideally bounding volumes should be as close-fitting to their object as possible. If two close-fitting bounding volumes touch, then their objects are likely to touch. If most of the space inside the bounding volumes isn't occupied, then touching bounding volumes is unlikely to mean the objects are in contact.

Spheres are convenient because they are easy to represent. Storing the center of the sphere and its radius is enough:

```
struct BoundingSphere
{
    Vector3 center;
    real radius;
};
```
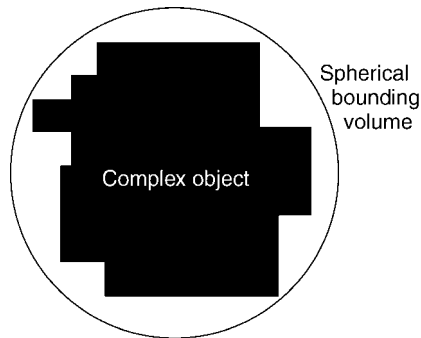
FIGURE 12.1   A spherical bounding volume.

It is also very easy to check whether two spheres overlap (see chapter 13 for the code). They overlap if the distance between their centers is less than the sum of their radii. Spheres are a good shape for bounding volumes for most objects.

Cubic boxes are also often used. They can be represented as a central point and a set of dimensions, one for each direction. These dimensions are often called "half-sizes" or "half-widths" because they represent the distance from the central point to the edge of the box, which is half the overall size of the box in the corresponding direction.

```
struct BoundingBox
{
    Vector3 center;
    Vector3 halfSize;
};
```

There are two common ways to use boxes as bounding volumes: either aligned to the world coordinates (called "axis-aligned bounding boxes," or AABBs) or aligned to the object's coordinates (called "object bounding boxes," or OBBs).[2] Spheres have no such distinction because they don't change under rotation. For tall and thin ob-

---

2. OBBs commonly can be oriented in a different way to the object they are enclosing. They still rotate with the object and are expressed in object space, but they have a constant offset orientation. This allows an even tighter fit in some cases, but adds an extra orientation to their representation and some overhead when working with them. The `BoundingBox` data structure would work for either axis-aligned bounding boxes or object bounding boxes with the same orientation as the rigid body they contained. For a general object bounding box we'd need to have a separate orientation quaternion in the bounding box structure.

jects, a bounding box will fit much more tightly than a bounding sphere. But detecting touching boxes is much more complex than detecting touching spheres, and so spheres are often a good place to start.

There are other possible bounding volume representations, with their own strengths and weaknesses. None are very widespread, however, so I will ignore them for the purpose of this chapter. They are discussed at length in Ericson [2005].

In the rest of this chapter I will use only bounding spheres. Anything that can be done with bounding spheres can also be done with bounding boxes. Typically the box version has exactly the same algorithm but will take longer and will use more tricky mathematics. In learning to use the algorithms, bounding spheres are simpler to work with.

As a tradeoff, however, it's important to remember that we're using these volumes as a first check to see whether objects are touching. If we had more accurate bounding volumes, then the first check would be more accurate, so we'd have less follow-up work to do. In many cases (particularly with lots of boxlike things in the game, such as crates), bounding spheres will generate many more potential collisions than bounding boxes. Then the time we save in doing the simpler sphere-collision tests will be lost by having additional potential collisions to reject using the more complex collision detection routines in this chapter.

12.3.1 Hierarchies

With each object enclosed in a bounding volume we can perform a cheap test to see whether objects are likely to be in contact. If the bounding volumes are touching, then the check can be returned from the coarse collision detector for a more detailed examination by the fine collision detector. This speeds up collision detection dramatically, but it still involves checking every pair of objects. We can avoid doing most of these checks by arranging bounding volumes in hierarchies.

A bounding volume hierarchy (BVH) has each object in its bounding volume at the leaves of a tree data structure. The lowest level bounding volumes are connected to parent nodes in the data structure, each of which has its own bounding volume. The bounding volume for a parent node is big enough to enclose all the objects descended from it.

We could calculate the bounding box at each level in the hierarchy so it best fits the object contained within it. This would give us the best possible set of hierarchical volumes. Many times, however, we can take the simpler route of choosing a bounding volume for a parent node that encompasses the bounding volumes of all its descendents. This leads to larger high-level bounding volumes, but recalculation of bounding volumes can be much faster. There is a tradeoff, therefore, between query performance (determining potential collisions) and the speed of building the data structure.

Figure 12.2 illustrates a hierarchy containing four objects and three layers. Note that there are no objects attached to parent nodes in the figure. This isn't an absolute
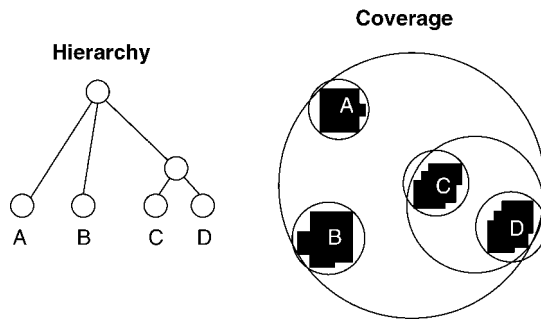
FIGURE 12.2 A spherical bounding volume hierarchy.

requirement: we could have objects higher in the tree, providing their bounding volume completely encompasses their descendents. In most implementations, however, objects are only found at the bottom. It is also common practice to have only two children for each node in the tree (i.e., a binary tree data structure). There are mathematical reasons for doing this (in terms of the speed of execution of collision queries), but the best reason to use a binary tree is ease of implementation: it makes the data structure compact and simplifies several of the algorithms we will meet later.

We can use the hierarchy to speed up collision detection: if the bounding volumes of two nodes in the tree do not touch, then *none* of the objects that descend from those nodes can possibly be in contact. By testing two bounding volumes high in the hierarchy we can exclude all their descendents immediately.

If the two high-level nodes do touch, then the children of each node need to be considered. Only combinations of these children that touch can have descendents that are in contact. The hierarchy is descended recursively; at each stage only those combinations of volumes that are touching are considered further. The algorithm finally generates a list of potential contacts between objects. This list is exactly the same as would have been produced by considering each possible pair of bounding volumes, but it is typically found many times faster.[3]

Assuming that the hierarchy encompasses all the objects in the game, the code to get a list of potential collisions looks like the following:

---

3. I say typically because it is possible for the bounding hierarchy to be slower than checking all possible combinations. If all the objects in the game are touching or nearly touching one another, then almost every bounding volume check will come up positive. In this case the overhead of descending the hierarchy adds time. Fortunately this situation occurs only rarely and when there are very few objects. With a larger number of objects (more than ten, I would estimate; it depends on the shape of the objects), there are checks that will fail, and the hierarchy becomes faster.

```
                          Excerpt from include/cyclone/collide_coarse.h
/**
 * Stores a potential contact to check later.
 */
struct PotentialContact
{
    /**
     * Holds the bodies that might be in contact.
     */
    RigidBody* body[2];
};

/**
 * A base class for nodes in a bounding volume hierarchy.
 *
 * This class uses a binary tree to store the bounding
 * volumes.
 */
template<class BoundingVolumeClass>
class BVHNode
{

public:
    /**
     * Holds the child nodes of this node.
     */
    BVHNode * children[2];

    /**
     * Holds a single bounding volume encompassing all the
     * descendents of this node.
     */
    BoundingVolumeClass volume;

    /**
     * Holds the rigid body at this node of the hierarchy.
     * Only leaf nodes can have a rigid body defined (see isLeaf).
     * Note that it is possible to rewrite the algorithms in this
     * class to handle objects at all levels of the hierarchy,
     * but the code provided ignores this vector unless firstChild
     * is NULL.
     */
    RigidBody * body;
    /**
```

```
     * Checks if this node is at the bottom of the hierarchy.
     */
    bool isLeaf() const
    {
        return (body != NULL);
    }

    /**
     * Checks the potential contacts from this node downward in
     * the hierarchy, writing them to the given array (up to the
     * given limit). Returns the number of potential contacts it
     * found.
     */
    unsigned getPotentialContacts(PotentialContact* contacts,
                                     unsigned limit) const;
};

template<class BoundingVolumeClass>
bool BVHNode<BoundingVolumeClass>::overlaps(
    const BVHNode<BoundingVolumeClass> * other
    ) const
{
    return volume->overlaps(other->volume);
}
template<class BoundingVolumeClass>
unsigned BVHNode<BoundingVolumeClass>::getPotentialContacts(
    PotentialContact* contacts, unsigned limit
    ) const
{
    // Early out if we don't have the room for contacts, or
    // if we're a leaf node.
    if (isLeaf() || limit == 0) return 0;

    // Get the potential contacts of one of our children with
    // the other.
    return children[0]->getPotentialContactsWith(
        children[1], contacts, limit
        );
}

template<class BoundingVolumeClass>
unsigned BVHNode<BoundingVolumeClass>::getPotentialContactsWith(
    const BVHNode<BoundingVolumeClass> *other,
    PotentialContact* contacts,
```

```
    unsigned limit
    ) const
{
    // Early-out if we don't overlap or if we have no room
    // to report contacts.
    if (!overlaps(other) || limit == 0) return 0;

    // If we're both at leaf nodes, then we have a potential contact.
    if (isLeaf() && other->isLeaf())
    {
        contacts->body[0] = body;
        contacts->body[1] = other->body;
        return 1;
    }

    // Determine which node to descend into. If either is
    // a leaf, then we descend the other. If both are branches,
    // then we use the one with the largest size.
    if (other->isLeaf() ||
        (!isLeaf() && volume->getSize() >= other->volume->getSize()))
    {
        // Recurse into ourself.
        unsigned count = children[0]->getPotentialContactsWith(
            other, contacts, limit
            );

        // Check whether we have enough slots to do the other side too.
        if (limit > count) {
            return count + children[1]->getPotentialContactsWith(
                other, contacts+count, limit-count
                );
        } else {
            return count;
        }
    }
    else
    {
        // Recurse into the other node.
        unsigned count = getPotentialContactsWith(
            other->children[0], contacts, limit
            );

        // Check whether we have enough slots to do the other side too.
        if (limit > count) {
```

```
                return count + getPotentialContactsWith(
                    other->children[1], contacts+count, limit-count
                    );
            } else {
                return count;
            }
        }
    }
}
```

This code can work with any kind of bounding volume hierarchy as long as each node implements the overlaps method to check to see whether two volumes overlap. The bounding sphere hierarchy is implemented as

─────────── **Excerpt from include/cyclone/collide_coarse.h** ───────────
```
/**
 * Represents a bounding sphere that can be tested for overlap.
 */
struct BoundingSphere
{
    Vector3 center;
    real radius;

public:
    /**
     * Creates a new bounding sphere at the given center and radius.
     */
    BoundingSphere(const Vector3 &center, real radius);

    /**
     * Creates a bounding sphere to enclose the two given bounding
     * spheres.
     */
    BoundingSphere(const BoundingSphere &one, const BoundingSphere &two);

    /**
     * Checks if the bounding sphere overlaps with the other given
     * bounding sphere.
     */
    bool overlaps(const BoundingSphere *other) const;
};
```

─────────── **Excerpt from src/collide_coarse.cpp** ───────────
```
bool BoundingSphere::overlaps(const BoundingSphere *other) const
{
```

```
    real distanceSquared = (center - other->center).squareMagnitude();
    return distanceSquared < (radius+other->radius)*
                             (radius+other->radius);
}
```

In a full collision detection system it is common to have a method to query the hierarchy against a known object too. This is simpler still: the object's bounding volume is checked against the root of the hierarchy, and as long as it overlaps, each descendent is checked recursively.

12.3.2 BUILDING THE HIERARCHY

An important question to ask at this stage is how the hierarchy gets constructed. It may be that your graphics engine has a bounding volume hierarchy already in place. Bounding volume hierarchies are used extensively to reduce the number of objects that need to be drawn. The root node of the hierarchy has its volume tested against the current camera. If any part of the bounding volume can be seen by the camera, then its child nodes are checked recursively. If a node can't be seen by the camera, then none of its descendents need to be checked. This is the same algorithm we used for collision detection: in fact it is effectively checking for collisions with the viewable area of the game.

In cases where a graphics engine does not have an existing bounding volume hierarchy to determine what objects can be seen, or if you are creating a game from scratch, you'll have to create your own. Ideally the hierarchy should have some key properties:

- The volume of the bounding volumes should be as small as possible, at each level of the tree. This is true when a parent node groups together two bounding volumes that are close together.
- Child bounding volumes of any parent should overlap as little as possible. Often this clashes with the first requirement, and in general it is better to favor smaller volumes over minimal overlaps. In fact, if you choose a minimal overlap at some low level of the tree, it is likely to cause greater overlaps higher up the tree: so a tree with an overall low overlap is likely to fulfill both requirements.
- The tree should be as balanced as possible. You should avoid having some branches of the tree that are very deep while others are very shallow. The worst-case scenario is a tree with one long branch. In this case the advantage of having a hierarchy is minimal. The biggest speed-up is gained when all branches are roughly at the same length.

There are various ways to construct a hierarchy, and each is a compromise between speed and quality. For relatively static worlds, where objects don't move much, a hierarchy can be created offline (i.e., while the game is not running: either while the level is loading or, more likely, while building the level before it ships). For very dy-
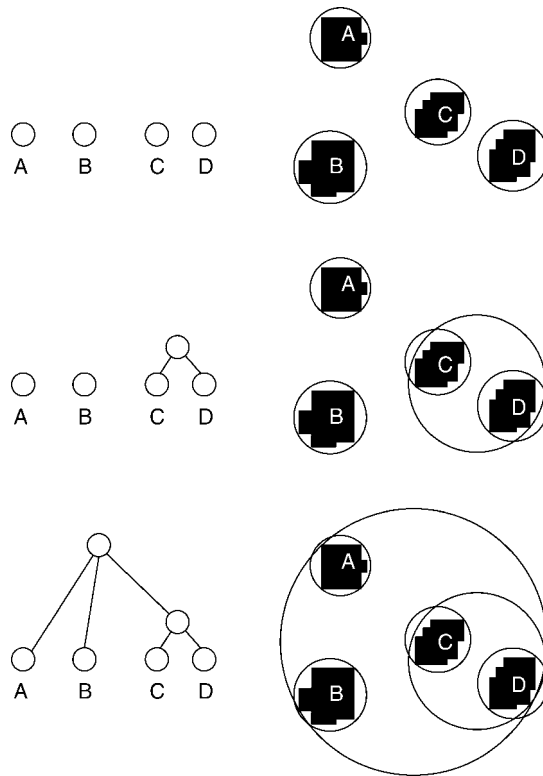
FIGURE 12.3 Bottom-up hierarchy building in action.

namic worlds where objects are constantly in motion (a space shooter, for example), the hierarchy needs to be rebuilt during the game.

I will give an overview and flavor of how hierarchies can be constructed, but it would take many chapters to go into complete detail. You can find more information in Ericson [2005].

The following are the three approaches to building a BVH.

- *Bottom-up* The bottom-up approach (illustrated in figure 12.3) starts with a list of bounding volumes corresponding to individual objects. Pairs of objects are chosen based on the requirements of the hierarchy just discussed, and a parent node is added for the pair. This parent node then replaces the objects in the list. The process continues until there is only one node left in the list.
- *Top-down* The top-down approach (illustrated in figure 12.4) starts with the same list as before. At each iteration of the algorithm the objects in the list are separated into two groups so that members of each group are clustered
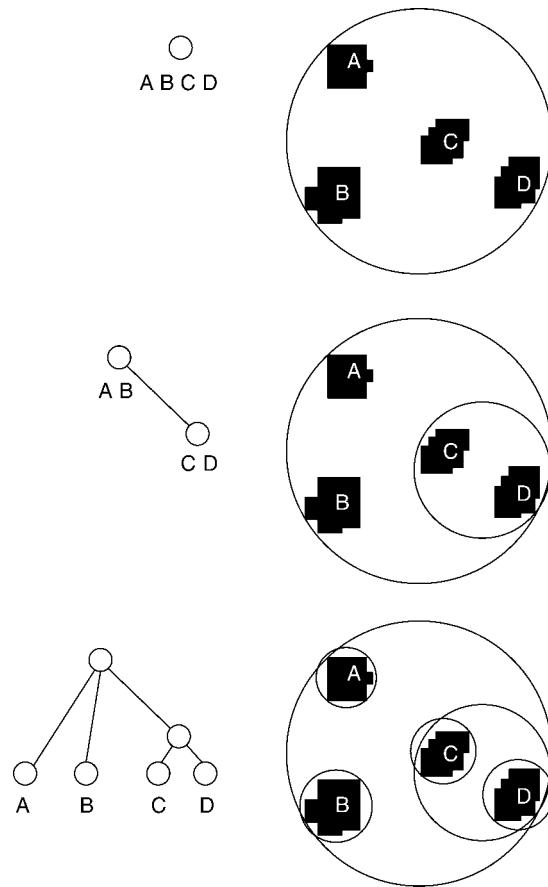
FIGURE 12.4   Top-down hierarchy building in action.

together. The same algorithm then applies to each group, splitting it into two, until there is only one object in each group. Each split represents a node in the tree.

- *Insertion* The insertion approach (illustrated in figure 12.5) is the only one suitable for use during the game. It can adjust the hierarchy without having to rebuild it completely. The algorithm begins with an existing tree (it can be an empty tree if we are starting from scratch). An object is added to the tree by descending the tree recursively: at each node the child is selected that would best accommodate the new object. Eventually an existing leaf is reached, which is then replaced by a new parent for both the existing leaf and the new object.
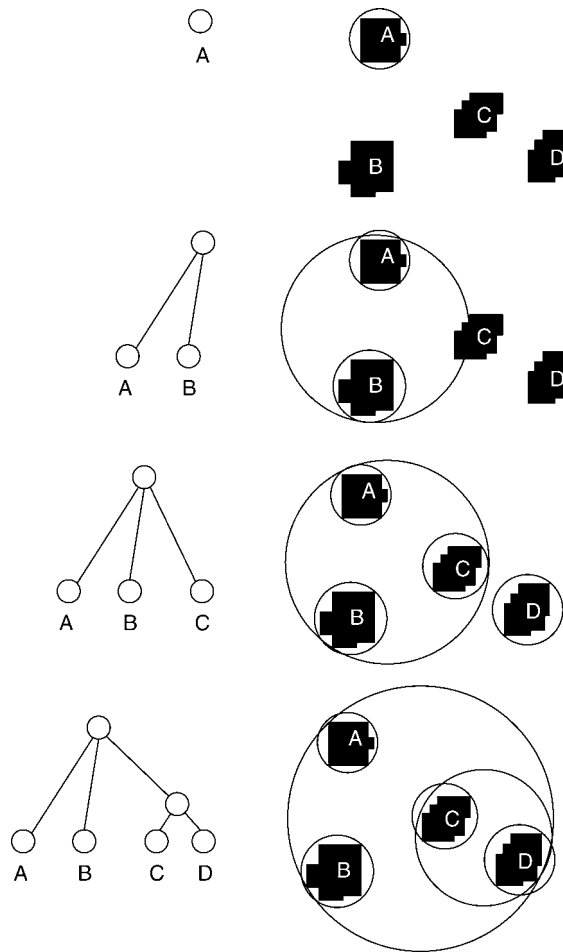
FIGURE 12.5   Insertion hierarchy building in action.

Each algorithm has many variations. In particular, the exact criteria used to group nodes together has a large effect on the quality of the tree. The bottom-up approach generally searches for nearby objects to group; the top-down approach can use any number of clustering techniques to split the set; and the insertion approach needs to select which child would be best to recurse into at each level of the tree. The specifics of the tradeoffs involved are complex, and to get the optimum results they require a good deal of fine-tuning and experimentation.

Fortunately even a simple implementation will give us a reasonable-quality tree and a good speed-up for the coarse collision detector. For our implementation I have

selected an insertion algorithm for the flexibility of being usable during the game. Given the sphere hierarchy we created previously, we can implement the insertion algorithm in this way:

```
                     ____ Excerpt from include/cyclone/collide_coarse.h ____
/**
 * A base class for nodes in a bounding volume hierarchy.
 *
 * This class uses a binary tree to store the bounding volumes.
 */
template<class BoundingVolumeClass>
class BVHNode
{
public:
    // ... other BVHNode code as before ...
    /**
     * Inserts the given rigid body, with the given bounding volume,
     * into the hierarchy. This may involve the creation of
     * further bounding volume nodes.
     */
    void insert(RigidBody* body, const BoundingVolumeClass &volume);
}

template<class BoundingVolumeClass>
void BVHNode<BoundingVolumeClass>::insert(
    RigidBody* newBody, const BoundingVolumeClass &newVolume
    )
{
    // If we are a leaf, then the only option is to spawn two
    // new children and place the new body in one.
    if (isLeaf())
    {
        // Child one is a copy of us.
        children[0] = new BVHNode<BoundingVolumeClass>(
            this, volume, body
            );

        // Child two holds the new body
        children[1] = new BVHNode<BoundingVolumeClass>(
            this, newVolume, newBody
            );

        // And we now loosen the body (we're no longer a leaf).
        this->body = NULL;
```

```
        // We need to recalculate our bounding volume.
        recalculateBoundingVolume();
    }

    // Otherwise we need to work out which child gets to keep
    // the inserted body. We give it to whoever would grow the
    // least to incorporate it.
    else
    {
        if (children[0]->volume.getGrowth(newVolume) <
            children[1]->volume.getGrowth(newVolume))
        {
            children[0]->insert(newBody, newVolume);
        }
        else
        {
            children[1]->insert(newBody, newVolume);
        }
    }
}
```

At each node in the tree we choose the child whose bounding volume would be least expanded by the addition of the new object. The new bounding volume is calculated based on the current bounding volume and the new object. The line between the centers of both spheres is found, as is the distance between the extremes of the two spheres along that line. The center point is then placed on that line between the two extremes, and the radius is half the calculated distance. Figure 12.6 illustrates this process.

Note that the combined bounding sphere encompasses both child bounding spheres: it is not normally the smallest sphere that encloses the child objects. We suffer this extra wasted space for performance reasons. To calculate the bounding sphere around two objects, we need to get down to the nitty-gritty of their geometries. This makes the process too slow for in-game use.

We can perform a similar algorithm to remove an object. In this case it is useful to be able to access the parent node of any node in the tree. Therefore we need to extend the data structure holding the hierarchy, like this:

————— Excerpt from include/cyclone/collide_coarse.h —————

```
/**
 * A base class for nodes in a bounding volume hierarchy.
 *
 * This class uses a binary tree to store the bounding volumes.
 */
```
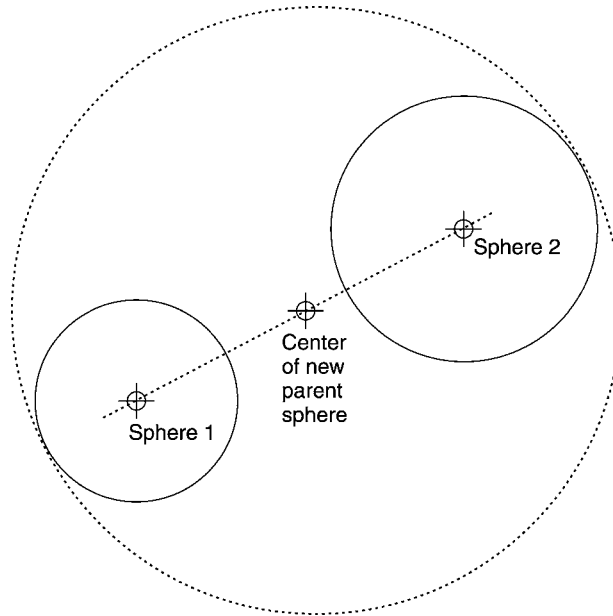
FIGURE 12.6 Working out a parent bounding sphere.

```
template<class BoundingVolumeClass>
class BVHNode
{
public:
    // ... other BVHNode code as before ...
```

Removing an object from the hierarchy involves replacing its parent node with its sibling and recalculating the bounding volumes farther up the hierarchy. Figure 12.7 illustrates this process. It can be implemented as

```
                    Excerpt from include/cyclone/collide_coarse.h
/**
 * A base class for nodes in a bounding volume hierarchy.
 *
 * This class uses a binary tree to store the bounding volumes.
 */
template<class BoundingVolumeClass>
class BVHNode
{
public:
```
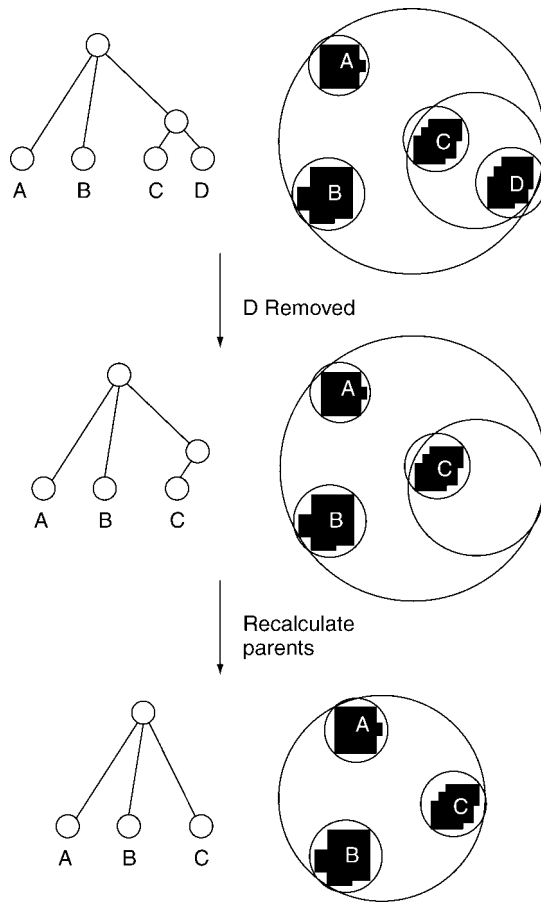
FIGURE 12.7 Removing an object from a hierarchy.

```
// ... other BVHNode code as before ...
/**
 * Deletes this node, removing it first from the hierarchy, along
 * with its associated rigid body and child nodes. This method
 * deletes the node and all its children (but obviously not the
 * rigid bodies). This also has the effect of deleting the sibling
 * of this node, and changing the parent node so that it contains
 * the data currently in that sibling. Finally it forces the
 * hierarchy above the current node to reconsider its bounding
 * volume.
 */
```

```
    ~BVHNode();
}

template<class BoundingVolumeClass>
BVHNode<BoundingVolumeClass>::~BVHNode<BoundingVolumeClass>()
{
    // If we don't have a parent, then we ignore the sibling processing.
    if (parent)
    {
        // Find our sibling.
        BVHNode<BoundingVolumeClass> *sibling;
        if (parent->children[0] == this) sibling = parent->children[1];
        else sibling = parent->children[0];

        // Write its data to our parent.
        parent->volume = sibling->volume;
        parent->body = sibling->body;
        parent->children[0] = sibling->children[0];
        parent->children[1] = sibling->children[1];

        // Delete the sibling (we blank its parent and
        // children to avoid processing/deleting them).
        sibling->parent = NULL;
        sibling->body = NULL;
        sibling->children[0] = NULL;
        sibling->children[1] = NULL;
        delete sibling;

        // Recalculate the parent's bounding volume.
        parent->recalculateBoundingVolume();
    }

    // Delete our children (again we remove their parent data so
    // we don't try to process their siblings as they are deleted).
    if (children[0]) {
        children[0]->parent = NULL;
        delete children[0];
    }
    if (children[1]) {
        children[1]->parent = NULL;
        delete children[0];
    }
}
```

12.3.3 Sub-Object Hierarchies

Some objects you'll need to simulate are large or have awkward shapes. It is difficult to create any simple bounding volume that fits tightly around such objects. For any particular bounding volume shape, there will be additional objects that simply don't suit that format. In each case the bounding volume is too large and the coarse collision detector will return too many false positives.

To solve this problem it is possible to use multiple bounding volumes for one object, arranged in a hierarchy. In figure 12.8 we have a long, thin object with a protrusion. Neither the bounding box nor the sphere fits nicely around it. If we use a hierarchy of bounding objects, we can provide a much closer fit. In this case the bounding boxes provide a better fit, although using a hierarchy of lots of bounding spheres arranged along its length would also work.

The algorithm for detecting collisions is the same as for the single-object hierarchical bounding volume. Rather than stopping at the bounding volume for the whole object, we can perform a finer-grained set of checks while still using the simple bounding volume comparison.

The same process allows us to build a hierarchy for the game level itself. Clearly most game levels are so large that their bounding volume is likely to encompass all other objects (although outdoor levels represented as a box can exclude objects at a high altitude).

To get a better fit we can decompose the level into a hierarchy of bounding volumes. Because of the boxlike structure of most game levels (rectangular walls, flat floors, and so on), a bounding box hierarchy is typically better than bounding spheres.

While this is acceptable, provides good performance, and has been used in some games, a more popular approach is to use a spatial data structure to work out collisions with the game level.
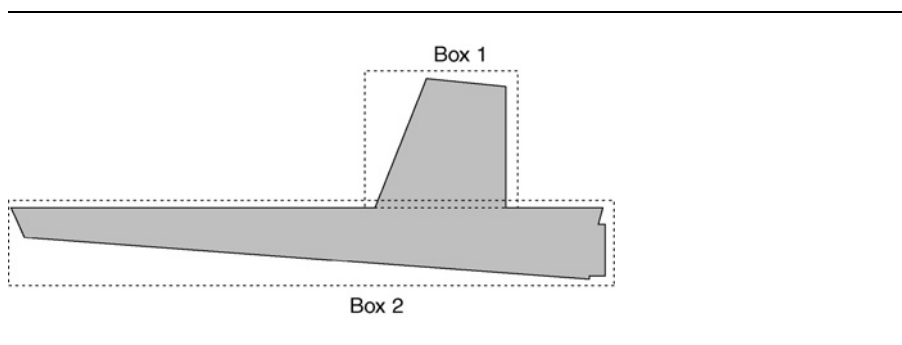
FIGURE 12.8   A sub-object bounding volume hierarchy.

12.4 SPATIAL DATA STRUCTURES

Several different approaches to coarse collision detection fall under the banner of "spatial data structures." The distinction between spatial data structures and bounding volumes is somewhat blurry.

A bounding volume hierarchy groups objects together based on their relative positions and sizes. If the objects move, then the hierarchy will move too. For different sets of objects the hierarchy will have a very different structure.

A spatial data structure is locked to the world. If an object is found at some location in the world, it will be mapped to one position in the data structure. A spatial data structure doesn't change its structure depending on what objects are placed within it. This makes it much easier to build the data structure.

In reality the line between the two is blurred, and a combination of techniques is sometimes used (hierarchies embedded in a spatial data structure, for example, or, less commonly, a spatial data structure at one node in a hierarchy). It is also worth noting that even when no bounding volume hierarchies are used, it is very common to use bounding volumes around each object. In the remainder of this chapter, I will assume objects are wrapped in a bounding volume: it makes many of the algorithms far simpler.

This section looks at three common spatial data structures: binary space partition (BSP) trees, quad- and oct-trees, and grids. In most games only one will be used.

12.4.1 BINARY SPACE PARTITIONING

A binary space partition tree behaves in a similar way to a bounding volume hierarchy. It is a binary tree data structure, and a recursive algorithm starts at the top of the tree and descends into child nodes only if they are capable of taking part in a collision.

Rather than use bounding volumes, however, each node in the BSP uses a plane that divides all space into two. It has two child nodes, one for each side of the plane. Objects lying on one side of the plane or the other will be found as a descendent of the corresponding child. Objects that cross the plane are handled differently: they can be directly attached to that node; placed in the child node that they are nearest to; or, more commonly, placed in both child nodes.

The dividing planes at each level are different, allowing all space to be divided up in any way. The figure later in this section shows a 2D version, but the same structure works for three dimensions.

Each plane in the BSP is represented as a vector position and a vector direction:

```
struct Plane
{
    Vector3 position;
    Vector3 direction;
};
```

This is a very common way to represent a plane: the position is any location on the plane, and the direction points out at right angles to the plane. The same plane can be generated if we reverse the direction: it would still be at right angles to the plane but facing in the opposite direction. The fact that the direction vector points out from one side of the plane means that we can distinguish one side from the other. Any object is either on the side where the direction is pointing or on the other side. This distinction allows us to select the appropriate child node in the BSP.

To determine which side of the plane an object lies on, we make use of the geometric interpretation of the scalar product given in chapter 2. Recall that the scalar product allows us to find the component of one vector in the direction of another:

$$c = (\boldsymbol{p}_o - \boldsymbol{p}_P) \cdot \boldsymbol{d}_P$$

where $\boldsymbol{p}_o$ is the position of the object we are interested in (we normally use the position of the center of its bounding volume), $\boldsymbol{p}_P$ is the position of any point on the plane (i.e., the point we are using to store the plane), and $\boldsymbol{d}_P$ is the direction in which the plane is facing.

If c is positive, then the object lies on the side of the plane to which its direction points. If it is negative, then the object lies on the opposite side. If c is zero, then it lies exactly on the plane. The direction vector, $\boldsymbol{d}_P$, should be a normalized vector, in which case $|c|$ gives the distance of the object from the plane.

Assuming that the object has a spherical bounding volume, we can determine whether it is completely on one side of the plane by checking if

$$|c| \geqslant r_o$$

where r_o is the radius of the bounding volume for the object.

We can build a BSP tree from nodes that contain a plane and two child nodes:

```
struct BSPNode
{
    Plane plane;
    BSPNode front;
    BSPNode back;
};
```

In practice each child node (`front` and `back`) can hold either another node or a set of objects. Unlike for bounding volume hierarchies, it is not normal to have only one object at each leaf of the tree.

In the same way as we saw for the bounding sphere hierarchy, this could be implemented in C++ as

```
typedef vector<Object*> BSPObjectSet;
```

```
enum BSPChildType
{
    NODE,
    OBJECTS
};

struct BSPChild
{
    BSPChildType type;

    union {
        BSPNode *node;
        BSPObjectSet *objects;
    };
};

struct BSPNode
{
    Plane plane;
    BSPChild front;
    BSPChild back;
};
```

Using polymorphism, inheritance, and C++'s runtime time inference (RTTI), the implementation would look like this:

```
struct BSPElement
{
};

struct BSPObjectSet : public BSPElement
{
    vector<Object*> objects;
};

struct BSPNode : public BSPElement
{
    Plane plane;
    BSPElement front;
    BSPElement back;
};
```

For all spatial data structures, leaves will usually be capable of carrying any number of objects. This is where bounding volume hierarchies can be useful: the group of objects at the leaf of the BSP can be represented as a BVH:

```
enum BSPChildType
{
    NODE,
    OBJECTS
};

struct BSPChild
{
    BSPChildType type;

    union {
        BSPNode *node;
        BoundingSphereHierarchy *objects;
    };
};

struct BSPNode
{
    Plane plane;
    BSPChild front;
    BSPChild back;
};
```

Let's assume we have a BSP tree where objects that intersect a plane are placed in both child nodes. In other words, one object can be at several locations in the tree. The only collisions that can possibly occur are between objects at the same leaf in the tree. We can simply consider each leaf of the tree in turn. If it has at least two objects contained within it, then all pair combinations of those objects can be sent to the fine collision detector for detailed checking.

If we place a bounding volume hierarchy at the leaves of the BSP tree, we can then call the coarse collision detection algorithm for each hierarchy. In this case we have two coarse collision detection steps.

If there are many objects, some large objects, or lots of partition planes, then having an object in multiple branches of the tree can lead to huge data structures and poor performance. The preceding algorithm can be modified to detect collisions when overlapping objects are sent to only one child node or are held in a list with the parent node. See Ericson [2005] for more comprehensive details.

BSP trees are common in rendering engines, and as with bounding volume hierarchies, you may be able to use an existing implementation for your physics system.
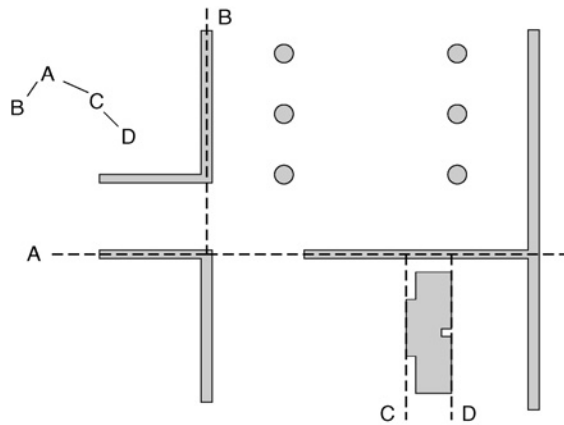
FIGURE 12.9 A binary space partition tree.

They are also commonly used to detect collisions between the level geometry and the game level. Figure 12.9 shows a small BSP for part of a game level.

The BSP doesn't hold objects at its leaves, but rather is a boolean indication of whether the object is colliding or not. An object is tested against each plane, recursively. If it intersects the plane, both children are checked; otherwise, only one is checked, as before. If the object reaches a leaf marked as a collision we know a collision has occurred.

Because most collisions will occur between moving objects and the level geometry (which typically cannot change or move in any way), the BSP approach is very useful. Unfortunately a complex preprocessing stage is required to build the BSP from the level geometry.

12.4.2 OCT-TREES AND QUAD-TREES

Oct-trees and quad-trees are spatial tree data structures with many similarities to both BSPs and BVHs. Quad-trees are used for two dimensions (or three dimensions where most objects will be stuck on the ground), and oct-trees for three dimensions. In many 3D games a quad-tree is as useful as an oct-tree and takes less memory, so I'll focus on that first.

A quad-tree is made up of a set of nodes, each with four descendents. The node splits space into four areas that intersect at a single point. It can be thought of as three nodes of a BSP tree, although the directions that are split are always aligned with the world axes. A node can be represented as a vector position and four children:
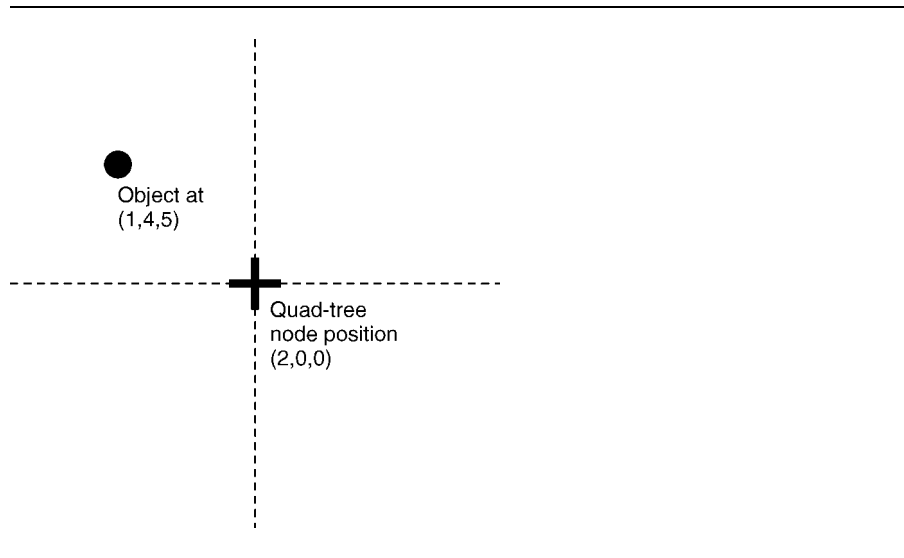
FIGURE 12.10 Identifying an object's location in a quad-tree.

```
struct QuadTreeNode
{
    Vector3 position;
    QuadTreeNode child[4];
};
```

Testing which of the four areas an object lies in is a simple matter of comparing the corresponding components of their position vector. For an object at $(1,4,5)$ and a QuadTreeNode at $(2,0,0)$, we know that it must be in the top left area, as shown in figure 12.10, because the x coordinate of the object is less than the node's coordinate and the z coordinate is greater. We can calculate which child in the array to use with the following simple algorithm:

```
struct QuadTreeNode
{
    // ... Other code as before ...

    unsigned int getChildIndex(const Vector3 &object)
    {
        unsigned int index;
        if (object.x > position.x) index += 1;
        if (object.z > position.z) index += 2;
```

```
            return index;
        }
    }
```

where the indices for each area match those shown in figure 12.10.

An oct-tree works in exactly the same way, but has eight child nodes and performs a comparison on each of the three vector components to determine where an object is located:

```
struct OctTreeNode
{
    Vector3 position;
    OctTreeNode child[8];

    unsigned int getChildIndex(const Vector3 &object)
    {
        unsigned int index;
        if (object.x > position.x) index += 1;
        if (object.y > position.y) index += 2;
        if (object.z > position.z) index += 4;
        return index;
    }
}
```

Although in theory the position vector for each node can be set anywhere, it is common to see quad- and oct-trees with each node dividing its parents in half. Starting with an axis-aligned bounding box that covers all the objects in the game, the top-level node is positioned at the center point of this box. This effectively creates four boxes of the same size (for a quad-tree; eight for an oct-tree). Each of these boxes is represented by a node, whose position is at the center point of that box, creating four (or eight) more boxes of the same size. And so on down the hierarchy.

There are two advantages to using this halving. First, it is possible to get rid of the position vector from the node data structure and calculate the point on the fly during recursion down the tree. This saves memory.

Second, it means we don't need to perform any calculations to find the best location to place each node's split point. This makes it much faster to build the initial hierarchy.

Other than their method of recursion and the number of children at each node, the quad- and oct-trees work in exactly the same way as the BSP tree. The algorithms that work with a BSP tree for determining collisions are the same as for a quad- or oct-tree, but the test is simpler and there are more possible children to recurse into.

Also as with the BSP tree, we have to decide where to put objects that overlap the dividing lines. In the previous code examples I have assumed the object goes into the

child that contains its center. We could instead place the object into all the children that it touches, as we did for the BSP tree, and have the same simple coarse collision detection: only objects in the same leaf can possibly be in collision.

Quad-trees are particularly useful for outdoor scenes, where objects are placed on a landscape. They are less useful than BSP trees for indoor games because they can't be used as easily for collision detection with the walls of the level. And, just like BSP trees, they are often used for optimizing rendering and may be part of any existing rendering engine you are using.

Because they are so similar to BSPs in practice, I will avoid repeating the code to work with them. On the CD, there is a complete implementation for BSPs, quad-trees, and oct-trees.

12.4.3 Grids

Our penultimate spatial data structure takes the idea of a quad-tree further. If we draw the split pattern of a halving quad-tree that is several layers deep, we see that it forms a grid (figure 12.11). Rather than use a tree structure to represent a regular grid, we could simply use a regular grid.

A *grid* is an array of locations in which there may be any number of objects. It is not a tree data structure because the location can be directly determined from the position of the object. This makes it much faster to find where an object is located than recursing down a tree.
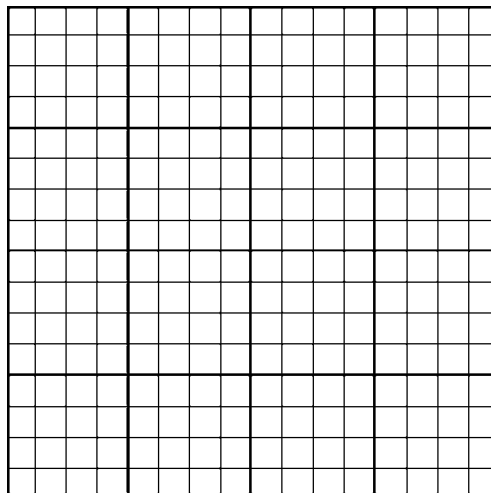
FIGURE 12.11 A quad-tree forms a grid.

The grid has the structure

```
struct Grid
{
    unsigned int xExtent;
    unsigned int zExtent;
    ObjectSet *locations; // An array of size (xExtent * zExtent)

    Vector origin;
    Vector oneOverCellSize;
};
```

where xExtent and zExtent store the number of cells in each direction of the grid; the x and z components of the oneOverCellSize vector contain 1 divided by the size of each cell (we use 1 over the value rather than the actual size to speed up the next algorithm); the y component is normally set to 1; origin is the origin of the grid. The grid should be large enough so that any object in the game is contained within it.

To determine the location in which the center of an object is contained, we use a simple algorithm:

```
struct Grid
{
    // ... Previous code as before ...

    def getLocationIndex(const Vector& object)
    {
        Vector square = object.componentProduct(oneOverCellSize);
    return (unsigned int)(square.x) + xExtent*(unsigned int)(square.z);
    }
};
```

In this code snippet, we first find which square the object is in by dividing each component by the size of the squares (we do the division by multiplying by 1 over the value). This gives us a floating-point value for each component in the vector called square. These floating-point values need to be converted into unsigned integers. The integer values are then used to find the index in the grid array, which is returned.

Just as in the BSP and quad-tree cases we need to decide what to do with objects that overlap the edge of a square. It is most common to simply place them into one cell or the other, although it would be possible to place them into all the cells they overlap. Just as before, the latter makes it faster to determine the set of possible collisions, but can take up considerably more memory. I'll look at this simpler case before returning to the more complex case.

In a grid where each cell contains all the objects that overlap it, the set of collisions can be generated very simply. Two objects can only be in collision if they occupy the same location in the grid. We can simply look at each location containing more than one object and check each pair for possible collisions.

Unlike tree-based representations, the only way we can tell if a location contains two or more objects is to check it. Whereas for a tree we stored the number of objects at each node and could completely miss branches that couldn't generate collisions, there is no such speed-up here.

To avoid searching thousands of locations for possible collisions (which for a small number of objects may take longer than if we had not performed coarse collision detection at all), we can create a set of locations in the grid data structure containing more than one object. When we add an object to a cell, if the cell now contains two objects, it should be added to the occupied list.

Likewise, when we remove an object from a cell, if the cell now contains just one object, it should be removed from the list. Determining a complete set of collisions is then just a matter of walking the occupied list and passing all pair-wise combinations to the fine-grained collision detector.

If objects are larger than the size of a cell, they will need to occupy many cells in the grid. This can lead to very large memory requirements with lots of wasted space. For a reasonable-size game level running on a PC this might not be an issue, but for large levels or memory-constrained consoles, it can be unacceptable.

If we place an object in just one grid cell (the cell in which its center is located, normally), then the coarse collision detection routine needs to check for collisions with objects in neighboring cells. For an object that is the same size as the cell, it needs to check a maximum of three neighbors from a possible set of eight (see figure 12.12). This rapidly increases, however. For an object four times the size of the cell, 15 from a possible 24 neighbors need to be considered. It is possible to write code to check the correct neighbors, but for very large objects it involves lots of wasted effort.

A hybrid data structure can be useful in this situation, using multiple grids of different sizes. It is normally called a "multi-resolution map."

12.4.4   Multi-Resolution Maps

A multi-resolution map is a set of grids with increasing cell sizes. Objects are added into one of the grids only, in the same way as for a single grid. The grid is selected based on the size of the object: it will be the smallest grid whose cells are bigger than the object.

Often the grids are selected so that each one has cells four times the size of the previous one (i.e., twice the width and twice the length for each cell). This allows the multi-resolution map to directly calculate which grid to add an object to.

During coarse collision detection the map uses a modified version of the single-grid algorithm. For each grid it creates a potential collision between each object and objects in the same or neighboring cells (there is a maximum of three neighbors to check now because objects can't be in a grid cell that is smaller than they are). In
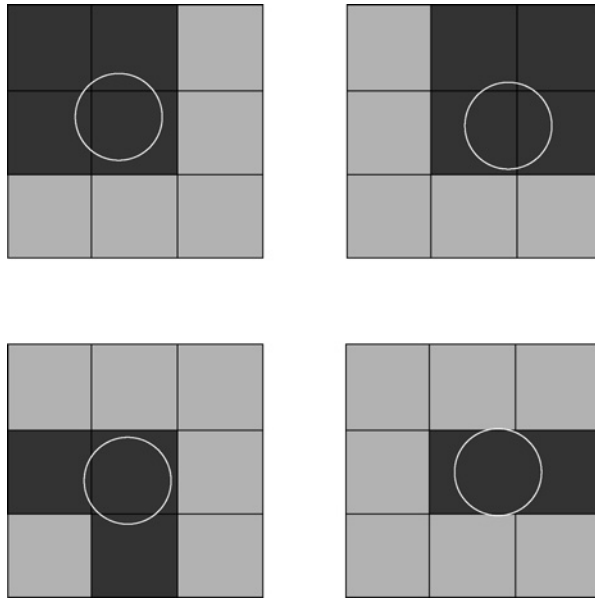
FIGURE 12.12   An object may occupy up to four same-sized grid cells.

addition, the object is checked against all objects in all cells in larger-celled grids that overlap. We don't need to check against objects in smaller-celled grids because the small objects are responsible for checking against larger objects.

For each grid in the map we can use the grid data structure with the same set of occupied cells. However, we need to add a cell to the active list if it contains any objects at all (because they may be in contact with neighbors).

12.5 SUMMARY
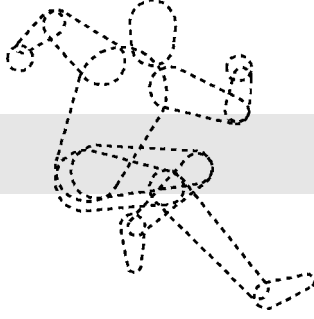
Collision detection is a complex and time-consuming process. To do it exhaustively takes too long for real-time physics, so some optimization is needed.

We can split collision detection into two phases: a coarse phase that finds possible contacts (some may turn out not to be collisions, but it should never *miss* a collision); and a fine-grained phase that checks potential collisions in detail and works out the contact properties.

Coarse collision detection works by wrapping objects in a simple bounding volume, such as a sphere or box, and performing checks on each collision volume. The collision volumes can be arranged in a hierarchy, which allows whole branches to be excluded, or in a spatial data structure, which allows nearby objects to be accessed together.

There are tradeoffs between different methods and the potential to blend several together in many cases. One of the most important factors to consider is how the rendering engine manages objects to decide whether they should be drawn. If your renderer has a system in place, it would be advisable to try to use or adapt it to save memory.

The result of the coarse phase is a set of possible contacts that need to be checked in more detail. We will look at the algorithms needed to perform these checks in the next chapter.

13

GENERATING CONTACTS

Coarse collision detection is just the first stage of collision-generating contacts. The algorithms in the previous chapter produce a list of object pairs that then needs to be checked in more detail to see whether the pairs do in fact collide.

Many collision detection systems perform this check for each pair and return a single point of maximum interpenetration if the objects are in contact. That is not what we need. We need contact generation. Two objects that are colliding can have more than one point of contact between them. Representing the collision with just a single contact works okay for some combinations of objects (such as a sphere and a plane), but not for others (such as a car and a plane: which wheel do we choose?).

Contact generation is more complex than single-intersection collision detection and takes more processor time to complete. Often we will have a two-stage process of contact generation: a fine collision detection step to determine whether there are contacts to generate and then a contact generation step to work out the contacts that are present.

Just because we have performed a coarse filtering step, it doesn't mean we can take as much time as we like to perform fine collision detection and contact generation. We need to make sure that fine collision detection runs as fast as possible. We can dramatically improve the speed by performing collision detection against a simplified geometry rather than the full-resolution rendering geometry.

The bulk of this chapter looks at generating the contacts between geometric primitives that are useful as stand-in collision geometry. There are lots of combinations, and this chapter tries to cover a representative selection.

But this book isn't about collision detection. There are other books in this series, including van den Bergen [2003], Ericson [2005], and Eberly [2004], that contain more material than I can cover here.

13.1 Collision Geometry

The complex visual geometry in many games is too rich for speedy contact generation. Instead it is simplified into a chunky geometry created just for the physics. If this chunky geometry consists of certain geometric primitives—namely, spheres, boxes, planes, and capsules (a cylinder with hemispherical ends)—then the collision detection algorithms can be simpler than for general-purpose meshes.

This collision geometry isn't the same as the bounding volumes used in coarse collision detection. In fact, depending on the complexity of the scenes you need to deal with, you may have many different levels of simplified geometry.

The simplest shape on which to perform collision detection and contact generation on is the sphere; hence its use in coarse collision geometry. Despite being fast, however, spheres aren't always terribly useful. Boxes are also relatively quick to process and can be used in more situations. Capsules are more difficult than spheres, but can be useful in some situations. Other primitives, such as disks, cylinders, and trimmed primitives, can also be useful.

A special case we need to consider is the collision of objects with the background-level geometry. Most commonly this means collisions with the ground or some other plane (walls can typically be represented as planes too). To support these, we'll also consider collisions between primitives and planes.

It is important to remember that the primitives your game needs will depend to some extent on the game. We'll only look in detail at spheres and boxes in this chapter; otherwise the whole book would be about contact generation. The principles are the same for any object, so with the help of the collision detection books in this series, you should be able to generate contacts for a wide range of primitive pairs.

But primitives only get you so far. All primitives can only fit their objects roughly; there are some objects that don't lend themselves well to fitting with primitives. In these cases it can be more efficient to use a general convex mesh (or set of such meshes).

Each of the collision algorithms is described in detail in the text, but most of the code is reserved for the CD. Some of the algorithms are very repetitive, and the code would run to more than a hundred pages if printed.

13.1.1 Primitive Assemblies

The vast majority of objects can't easily be approximated by a single primitive shape. Some developers don't even try: they use convex meshes to approximate all objects. Another approach is to use assemblies of primitive objects as collision geometry.

Figure 13.1 shows an object approximated by an assembly of boxes and spheres. To collide two assemblies we need to find all collisions between all pairs of primitives
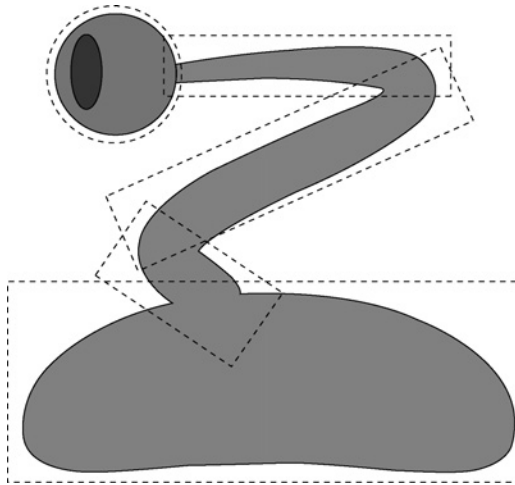
FIGURE 13.1   An object approximated by an assembly of primitives.

in each object (in this way the assembly acts something like the hierarchy of bounding volumes we saw in chapter 12).

We can represent assemblies as a list of primitives, with a transform matrix that offsets the primitive from the origin of the object.

13.1.2 Generating Collision Geometry

Generating the collision geometry to approximate an object isn't trivial. The easiest method is to ask designers to create the collision geometry manually. This doesn't require any special algorithms, but increases the already large burden on modelers and level designers. Given that most of the cost of developing a game goes into design, this may not be the most economical solution. It is no coincidence that in many games with great physics, objects under physics control tend to be simple, primitive shapes (crates and barrels, for example).

Some developers I know have created tools for generating simplified convex meshes to act as collision geometry. The algorithms they use involve some complex geometry and mathematics beyond this book. I am not aware of tools that place assemblies of collision primitives.

13.2 Contact Generation

As we have already seen in this chapter, it is important to make a distinction between collision detection and contact generation. Most books on collision detection will not

tell you how to build a physics engine collision system. In fact, most commercial or open source collision detection systems aren't suitable for physics applications.

Collision detection determines whether two objects are touching or interpenetrated, and normally provides data on the largest interpenetration point. Contact generation produces the set of points on each object that are in contact (or penetrating).

The difference is crucial. In figure 13.2 one box is lying across another: in the left part of the figure the result of a typical collision detection system is shown: the box is slightly (microscopically) twisted, so one edge generates the contact. In the right part of the figure the correct contact patch is generated.

The contact patch can be of any shape. This can make a challenging programming task when it comes to resolving contacts. To make things easier we would like to deal only with point contacts. The naive collision detection shown in figure 13.2 does this, but doesn't provide enough contacts to generate realistic physics. We need a reliable way to simplify a contact patch into a set of point contacts.

To do that we deal with a set of contact situations as shown in figure 13.3. Although the contact patch can be any shape, the simplifications in the figure generate reasonable physical behavior. While there are many situations in which they aren't optimal, there are few in which the problem is noticeable when the physics is running. In figure 13.3 these cases are arranged in order of how useful they are. If we can generate useful contacts higher in the list, we can ignore those lower down.

Of the illustrated cases, we ignore point–point contacts and point–edge contacts altogether. These can be handled relatively easily, but they are rare and are normally associated with other contact cases that can take the load. In the small number of cases where the contact can only be represented in one of these two forms, we will miss the contact. Experience shows that this isn't significant enough to be noticed.

The only occasion that we need to deal with face–face collisions is when one or the other face is curved. In all other cases the edge–edge and edge–face contacts will give the correct physics. Similarly edge–face contacts can often be replaced by a pair of edge–edge contacts (except when the edge is curved), and we will prefer to use these.
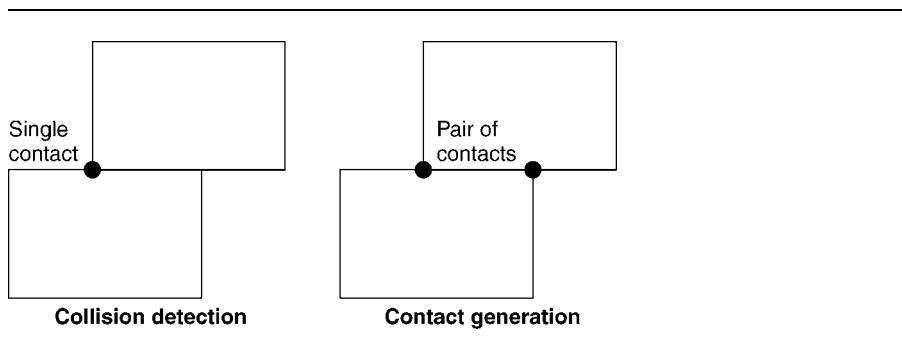
FIGURE 13.2   Collision detection and contact generation.
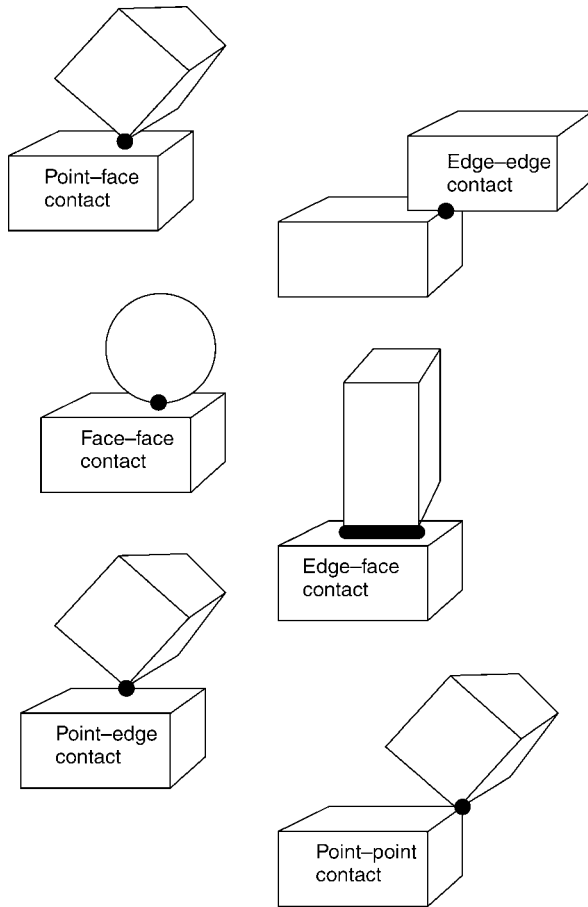
FIGURE 13.3   Cases of contact.

In each of the primitive collision cases that follow we will be looking for point–face contacts and edge–edge contacts first. Those primitives with curved sides will also use the lower cases when necessary.

13.2.1 CONTACT DATA

As we will see later in the book, there are a number of bits of data we'll need for each contact generated, as follows.

- *The collision point* This is the point of contact between the objects. In practice objects will be interpenetrating somewhat, and there may be any number
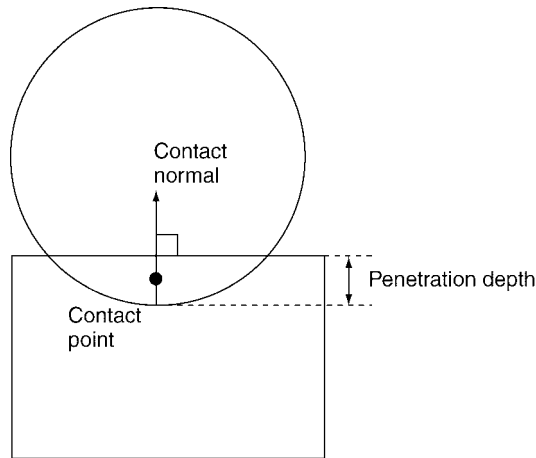
FIGURE 13.4   The relationship between the collision point, collision normal, and penetration depth.

of possible points. The selection of a point from the many options is largely arbitrary and doesn't drastically affect the physics.

■ *The collision normal* This is the direction in which an impact impulse will be felt between the two objects. When we saw collisions for non-rotating objects in chapter 7, this was the direction in which interpenetrating objects should be moved apart. In some cases (such as point–face contacts) this is simple to calculate; in other cases it isn't clear which direction the normal should be in. By convention, the contact normal points from the first object involved toward the second. We will assume this convention throughout the contact resolution code in this book.

■ *The penetration depth* This is the amount that the two objects are interpenetrating. It is measured along the direction of the collision normal passing through the collision point, as shown in figure 13.4.

These elements are stored in a contact data structure:

```
                     Excerpt from include/cyclone/contacts.h
/**
 * A contact represents two bodies in contact. Resolving a
 * contact removes their interpenetration, and applies sufficient
 * impulse to keep them apart. Colliding bodies may also rebound.
 * Contacts can be used to represent positional joints, by making
 * the contact constraint keep the bodies in their correct
 * orientation.
 */
```

```
class Contact
{
    /**
     * Holds the position of the contact in world coordinates.
     */
    Vector3 contactPoint;

    /**
     * Holds the direction of the contact in world coordinates.
     */
    Vector3 contactNormal;

    /**
     * Holds the depth of penetration at the contact point. If both
     * bodies are specified then the contact point should be midway
     * between the inter-penetrating points.
     */
    real penetration;
};
```

Before looking at the particulars of different primitive collisions, it's worth looking at each contact case in turn and how its parameters are determined.

13.2.2 POINT–FACE CONTACTS

Point–face contacts are the most common and important type of contact. Whether the face is flat or curved, the contact properties are generated in the same way. This is illustrated in figure 13.5.

The contact normal is given by the normal of the surface at the point of contact. If the object point (i.e., the point that is in contact with the face) is penetrated into the surface, then it is usually projected back onto the surface in order to determine where the contact is measured from.

The contact point is given as the point involved in the contact. In some cases a point is used that is midway between this object point and the projected point on the face. Either case works well, but using the given point is often more efficient.

The penetration depth is calculated as the distance between the object point and the projected point.

13.2.3 EDGE–EDGE CONTACTS

Edge–edge contacts are the second most important type of contact and are critical for resting contacts between objects with flat or concave sides. The contact data is shown in figure 13.6.
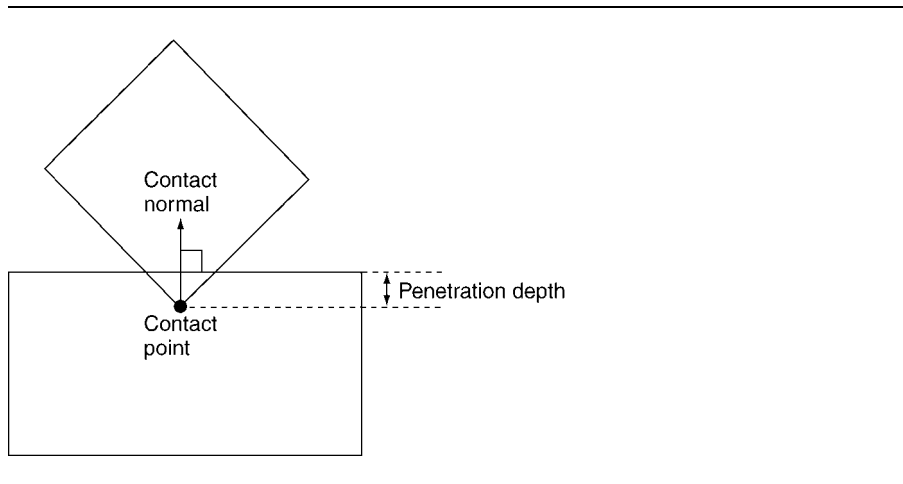
FIGURE 13.5   The point–face contact data.
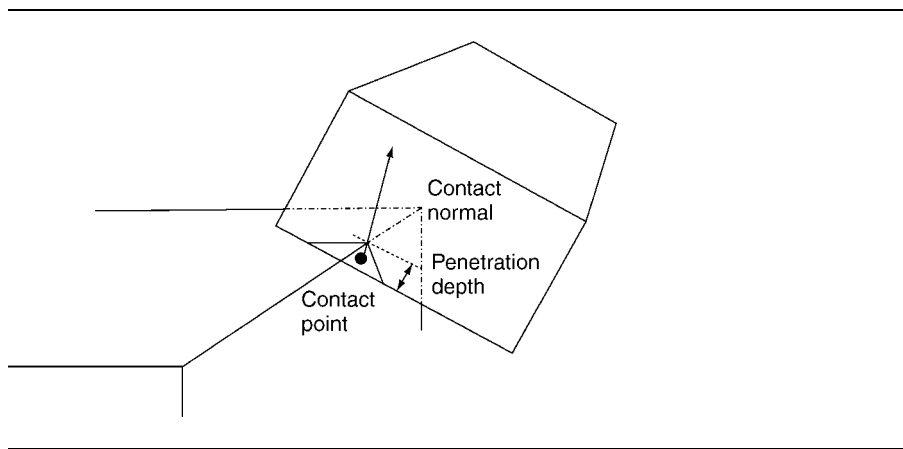
FIGURE 13.6   The edge–edge contact data.

The contact normal is at right angles to the tangents of both edges. The vector product is used to calculate this.

The contact point is typically the closest point on one edge to the other. Some developers use a point midway between the two edges, which takes longer to calculate but is marginally more accurate. The penetration depth is the distance between the two edges.
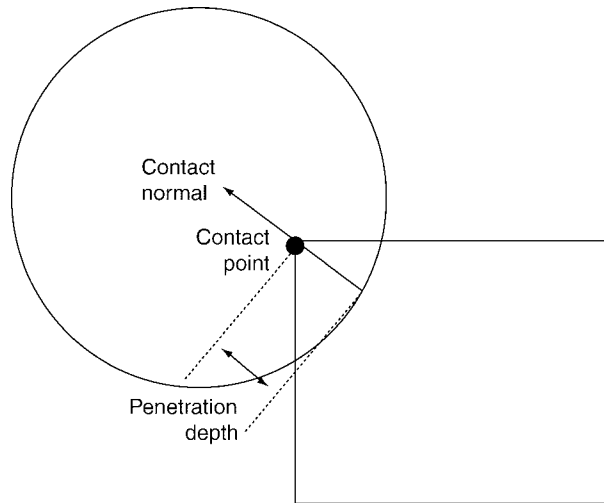
FIGURE 13.7   The edge–face contact data.

13.2.4 EDGE–FACE CONTACTS

Edge–face contacts are only used with curved surfaces (the edge of a capsule, for example, or the surface of a sphere). The contact data is generated in a very similar way to point–face contacts, shown in figure 13.7.

The contact normal is given by the normal of the face, as before. The edge direction is ignored in this calculation.

The contact point is more difficult to calculate for the general case. In the more general case we need to calculate the point of deepest penetration geometrically. For some primitives there is a quick way to get this.

Because of the way the contact point is calculated, we normally have direct access to the penetration depth. If not, then it needs to be calculated the long way, by working out the distance between the edge and the face along the direction of the normal passing through the contact point.

13.2.5 FACE–FACE CONTACTS

Face–face contacts occur when a curved surface comes in contact with another face, either curved or flat, such as a sphere on a plane. The contact data is somewhat more arbitrary than for other cases. Figure 13.8 shows the properties in detail.

The contact normal is given by the normal of the first face. In theory the faces should have opposite contact normals: two faces can't touch except where their normals are in the opposite directions. In practice, however, this isn't perfect, and the fact
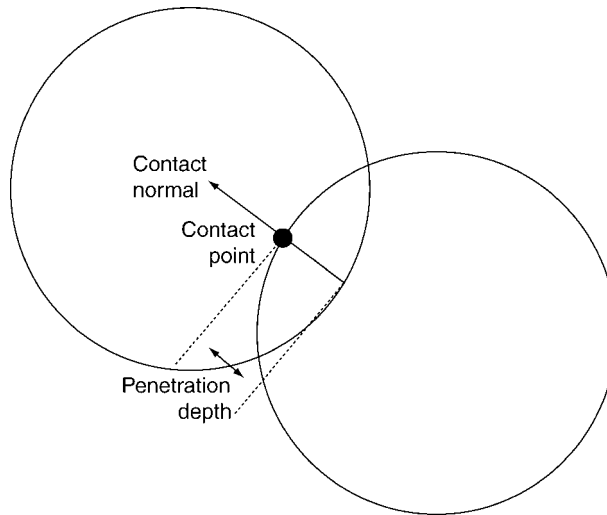
FIGURE 13.8   The face–face contact data.

that objects may interpenetrate means that the actual normals may be misaligned. It is easier to use just one contact normal consistently as long as the two objects won't swap roles in future contact generations (i.e., in the next frame we should avoid having a contact generation where objects A and B are swapped). It is unusual to find such swapping, so it is safely ignored.

The contact point is again difficult to calculate in the general case. And once again, using the primitives in this chapter, we can often get directly at the point of greatest penetration. If not, then we need to select some point (pretty arbitrarily in the code I've seen that does this) from the inside of the interpenetrating volume.

The contact point calculation will normally give us direct access to the penetration depth. In the general case we'll have to follow the full algorithm, as we saw in the last section.

13.2.6 Early-Outs

Some of the contact generation algorithms can be quite time consuming. The coarse collision detection will generate candidate pairs of objects that may later be found not to be in contact. We can make collision detection much more efficient by creating algorithms that exit early if no contact is found.

There are numerous opportunities to do this as part of the contact generation algorithms we will look at later and the code takes advantage of as many of these as possible.

Some of the primitive collisions have completely different algorithms that can determine whether there is a contact without generating the contacts themselves. If such an algorithm exists and it is fast enough, it can be useful to call it as a first stage:

```
if (inContact())
{
    findContacts();
}
```

These are the collision detection algorithms often found in books on game graphics.

In many cases, however, the work that the quick check would have to do is the same as required during contact generation. The speed-up of doing both would be minimal for the number of times we'd get a no-contact in the test. In these cases the contact generation algorithm can be used on its own.

13.3 Primitive Collision Algorithms

Each of the collision algorithms in this chapter checks for contact and generates contact data structures for different combinations of primitives. Some of the algorithms are guaranteed to return only zero or one contact (the sphere–sphere collision, for example). We could have these algorithms return the contacts directly.

Other algorithms can return zero, one, or several contacts. In this case we need a more flexible way for the algorithm to return the contacts it generates. The simplest way to do this is to start with an array or list of possible contacts. This array is then passed into each contact generation routine. If the routine finds contacts, it can write them into the array.

In the code on the CD I have encapsulated this process into a class:

Excerpt from include/cyclone/collide_fine.h

```
/**
 * A helper structure that contains information for the detector to use
 * in building its contact data.
 */
struct CollisionData
{
    /** Holds the contact array to write into. */
    Contact *contacts;

    /** Holds the maximum number of contacts the array can take. */
    unsigned contactsLeft;
};
```

Each contact generation routine has the same form:

```
void detectContacts(const Primitive &firstPrimitive,
                    const Primitive &secondPrimitive,
                    CollisionData *data);
```

where the `Primitive` type holds the data for the collision geometry. The `Primitive` class holds data that any contact generator will need to know, such as the rigid body corresponding to the geometry and the offset of the primitive from the coordinate origin of the rigid body.

```
class Primitive
{
public:
  RigidBody *body;
  Matrix4 offset;
};
```

Throughout this chapter I will assume that the offset matrix represents translation and rotation only: it has no scaling or skewing effect. We could make this much simpler and assume that the primitive is aligned perfectly with the center of the rigid body with no rotation (as would be the case if we had a cylinder representing a barrel, for example). Unfortunately this would not work for assemblies of objects or for rigid bodies with centers of mass that aren't at their geometric center. For flexibility it is best to allow primitives to be offset from their rigid bodies.

Each implemented contact generation function will use a subtype of `Primitive` with some additional data (such as `Sphere` and `Plane`). I'll introduce these types as we go along.

13.3.1 COLLIDING TWO SPHERES

Colliding two spheres is as simple as it gets. Two spheres are in contact if the distance between their centers is less than the sum of their radii.

If they are in contact, then there will be precisely one contact point: each sphere consists of one surface, so it will be a face–face contact (see figure 13.8).

The point of deepest contact is located along the line between the sphere centers. This is exactly the same algorithm we saw in chapter 7 when looking at particle collisions.

To implement this we need a data structure for a sphere. Spheres are completely defined by their center point and radius:

```
class Sphere
{
public:
```

```
  Vector3 position;
  real radius;
};
```

The center point of the sphere is given by the offset from the origin of the rigid body, the data for which is contained in the `Primitive`. The sphere implementation we'll use looks like this:

```
class Sphere : public Primitive
{
public:
  real radius;
};
```

The algorithm takes two spheres and may generate a contact in the contact data. Because the algorithm to determine whether the two spheres collide is part of determining the contact data, we don't have a separate algorithm to provide an early out:

```
────── Excerpt from src/collide_fine.cpp ──────
unsigned CollisionDetector::sphereAndSphere(
    const Sphere &one,
    const Sphere &two,
    CollisionData *data
    )
{
    // Make sure we have contacts.
    if (data->contactsLeft <= 0) return 0;

    // Cache the sphere positions.
    Vector3 positionOne = one.getAxis(3);
    Vector3 positionTwo = two.getAxis(3);

    // Find the vector between the objects.
    Vector3 midline = positionOne - positionTwo;
    real size = midline.magnitude();

    // See if it is large enough.
    if (size <= 0.0f || size >= one.radius+two.radius)
    {
        return 0;
    }

    // We manually create the normal, because we have the
```

```
    // size to hand.
    Vector3 normal = midline * (((real)1.0)/size);

    Contact* contact = data->contacts;
    contact->contactNormal = normal;
    contact->contactPoint = positionOne + midline * (real)0.5;
    contact->penetration = (one.radius+two.radius - size);

    // Write the appropriate data.
    contact->body[0] = one.body;
    contact->body[1] = two.body;
    contact->restitution = data->restitution;
    contact->friction = data->friction;

    return 1;
}
```

13.3.2 COLLIDING A SPHERE AND A PLANE

Colliding a sphere and a plane is just as simple as colliding two spheres. The sphere collides with the plane if the distance of the center of the sphere is farther from the plane than the sphere's radius. The distance of a point from a plane is given by

$$d = \boldsymbol{p} \cdot \boldsymbol{l} - l$$

where $\boldsymbol{l}$ is the normal vector of the plane and l is the offset of the plane. This is a standard way to represent a plane in 3D geometry.

We can represent the plane in code as

```
class Plane : public Primitive
{
public:
  Vector3 normal;
  real offset;
};
```

Planes are almost always associated with immovable geometry rather than a rigid body, so the rigid body pointer in the `Primitive` class will typically be NULL.

The algorithm takes a sphere and a plane and may add a contact to the contact data. Again the algorithm is simple enough not to benefit from a separate early-out algorithm.

```
                         ─── Excerpt from src/collide_fine.cpp ───
unsigned CollisionDetector::sphereAndHalfSpace(
    const Sphere &sphere,
    const Plane &plane,
    CollisionData *data
    )
{
    // Make sure we have contacts.
    if (data->contactsLeft <= 0) return 0;

    // Cache the sphere position.
    Vector3 position = sphere.getAxis(3);

    // Find the distance from the plane.
    real ballDistance =
        plane.direction * position -
        sphere.radius - plane.offset;

    if (ballDistance >= 0) return 0;

    // Create the contact - it has a normal in the plane direction.
    Contact* contact = data->contacts;
    contact->contactNormal = plane.direction;
    contact->penetration = -ballDistance;
    contact->contactPoint =
        position - plane.direction * (ballDistance + sphere.radius);

    // Write the appropriate data.
    contact->body[0] = sphere.body;
    contact->body[1] = NULL;
    contact->restitution = data->restitution;
    contact->friction = data->friction;

    return 1;
}
```

Strictly speaking, this isn't a sphere–plane collision but a sphere–half-space collision. Figure 13.9 shows the difference. Planes are rarely needed in a game (because they are infinitely big), but half-spaces are common. They are normally used in games to represent the ground and walls as part of a BSP tree.

To modify the algorithm to perform true plane–sphere collisions we need to check whether the distance is either greater than the radius of the sphere or less than the negative of that radius.
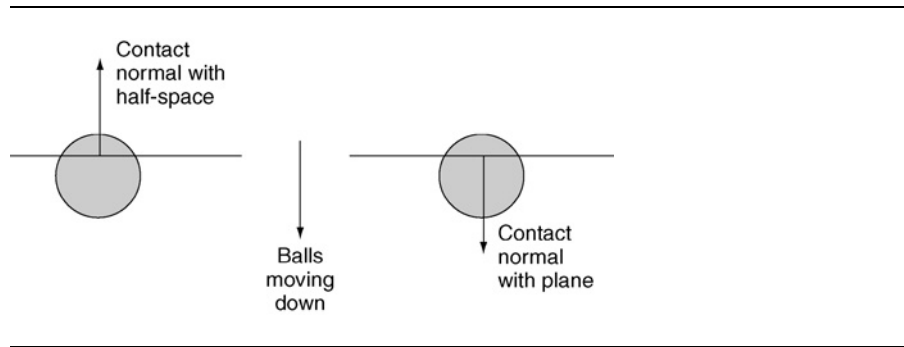
FIGURE 13.9   The difference in contact normal for a plane and a half-space.

```
                        Excerpt from src/collide_fine.cpp
unsigned CollisionDetector::sphereAndTruePlane(
    const Sphere &sphere,
    const Plane &plane,
    CollisionData *data
    )
{
    // Make sure we have contacts.
    if (data->contactsLeft <= 0) return 0;

    // Cache the sphere position.
    Vector3 position = sphere.getAxis(3);

    // Find the distance from the plane.
    real centerDistance = plane.direction * position - plane.offset;

    // Check if we're within radius.
    if (centerDistance*centerDistance > sphere.radius*sphere.radius)
    {
        return 0;
    }

    // Check which side of the plane we're on.
    Vector3 normal = plane.direction;
    real penetration = -centerDistance;
    if (centerDistance < 0)
    {
        normal *= -1;
        penetration = -penetration;
    }
```

```
        penetration += sphere.radius;

        // Create the contact - it has a normal in the plane direction.
        Contact* contact = data->contacts;
        contact->contactNormal = normal;
        contact->penetration = penetration;
        contact->contactPoint = position - plane.direction * centerDistance;

        // Write the appropriate data.
        contact->body[0] = sphere.body;
        contact->body[1] = NULL;
        contact->restitution = data->restitution;
        contact->friction = data->friction;

        return 1;
}
```

Both are implemented on the CD, but only the half-space is used in any of the demos.

13.3.3 COLLIDING A BOX AND A PLANE

The last of the very simple contact generation algorithms is between a box and a plane (strictly a half-space). This is the first algorithm that can return more than one contact.

Remember that we are trying to use contacts that are as simple to process as possible. We prefer to use point–face contacts if we can. In this case we can. Rather than return the contact of the face of one box with the half-space, we return four contacts for each corner point with the half-space. Similarly, if an edge is colliding with the plane, we treat it as two point–face contacts for each end of that edge. Thus there can be up to four contacts, and each is a point–face contact. Figure 13.10 illustrates this.

We can find the set of contacts by simply checking each vertex of the box one by one and generating a contact if it lies below the plane.[1]

The check for each vertex looks just like the check we made with the sphere–plane detector:

$$d = \boldsymbol{p} \cdot \boldsymbol{l} - l$$

Because the vertices are only points, and have no radius, we simply need to check to see whether the sign of d is positive or negative. A collision therefore occurs if

$$\boldsymbol{p} \cdot \boldsymbol{l} < l$$

---

1. Generating contacts for a true plane, rather than a half-space, is somewhat more difficult because we need to find the set of contacts on each side of the plane, determine which side the box is on, and generate contacts for the opposite set. For a half-space we simply test whether vertices are through the plane.
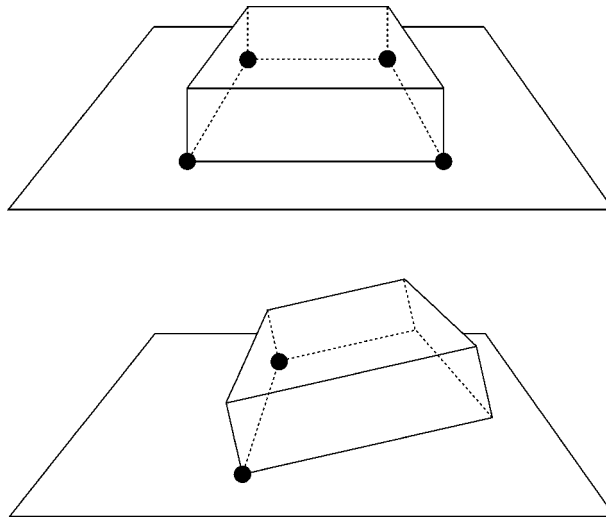
FIGURE 13.10   Contacts between a box and a plane.

For one vertex at *p* the code to generate a contact looks like this:

```
                         Excerpt from src/collide_fine.cpp
// Calculate the distance from the plane.
real vertexDistance = vertexPos * plane.direction;

// Compare this to the plane's distance.
if (vertexDistance <= plane.offset + data->tolerance)
{
    // Create the contact data.

    // The contact point is halfway between the vertex and the
    // plane - we multiply the direction by half the separation
    // distance and add the vertex location.
    contact->contactPoint = plane.direction;
    contact->contactPoint *= (vertexDistance-plane.offset);
    contact->contactPoint = vertexPos;
    contact->contactNormal = plane.direction;
    contact->penetration = plane.offset - vertexDistance;
}
```

The full algorithm runs this code for each vertex of the box. We can generate the set of vertices from a box data structure that looks like this:

```
class Box : public Primitive
{
public:
    Vector3 halfSize;
};
```

where `halfSize` gives the extent of the box along each axis. The total size of the box along each axis is twice this value, as shown in figure 13.11.

The vertices of the box are then given as follows:

```
Vector3 vertices[8] =
{
  Vector3(-halfSize.x -halfSize.y -halfSize.z),
  Vector3(-halfSize.x -halfSize.y +halfSize.z),
  Vector3(-halfSize.x +halfSize.y -halfSize.z),
  Vector3(-halfSize.x +halfSize.y +halfSize.z),
  Vector3(+halfSize.x -halfSize.y -halfSize.z),
  Vector3(+halfSize.x -halfSize.y +halfSize.z),
  Vector3(+halfSize.x +halfSize.y -halfSize.z),
  Vector3(+halfSize.x +halfSize.y +halfSize.z)
};

for (unsigned i = 0; i < 8; i++)
{
  vertices[i] = offset * vertices[i];
}
```

where `offset` is the rotation and translation matrix from the `Primitive` class.

The overall algorithm simply generates each vertex in turn and tests it against the plane, adding a contact if necessary. The full algorithm is quite repetitive, so I won't duplicate it here. You can find it on the CD.
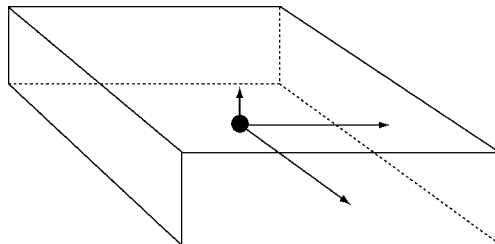
FIGURE 13.11   The half-sizes of a box.

Note that there are algorithms that avoid generating and testing all vertices. By comparing the direction of each box axis against the plane normal, we can trim down the number of vertices that need to be checked. You may find such algorithms in other books or online.

Despite the marginal theoretical advantage of such an algorithm, I have found them to have no efficiency gain in practice. Generating and testing a vertex is so fast that additional checking has a marginal effect. If you are familiar with the technique, you will also notice that this algorithm lends itself very easily to parallel implementation on SIMD hardware, which makes the alternatives even less attractive.

13.3.4 COLLIDING A SPHERE AND A BOX

A box complicates the calculations somewhat. When a sphere collides with a box, we will have just one contact. But it may be a contact of any type: a face–face contact, an edge–face contact, or a point–face contact. In each case the sphere (which has no edges or vertices) contributes a face to the contact, but it may touch either a face, edge, or point of the box. Figure 13.12 illustrates this.

Fortunately all three of these cases involve the same calculations for the normal contact and penetration depth, as long as we assume that the sphere gets to decide the contact normal in the case of a face–face contact. (Recall that, for face–face contacts, one or other object has to work out the normal.)

Once again we can simplify this query to use a point rather than a sphere. In this case we need to find the closest point in the box to the center of the sphere. This closest point may be in a corner of the box, along an edge, or on a face. If the distance between the closest point and the center of the sphere is less than the radius of the sphere, then the two objects are touching.

Because we'll deal with all three types of contact in the same way—allowing the properties of the sphere to determine the contact data—we won't need to keep track of whether the closest point in the box is on a face, edge, or vertex.
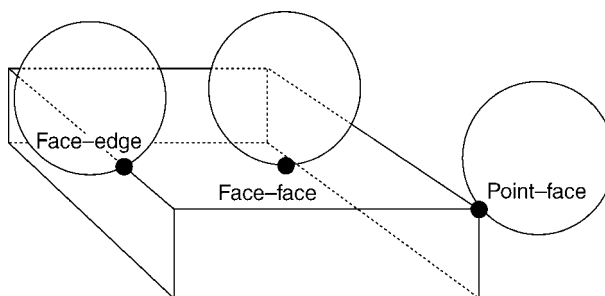
FIGURE 13.12   Contacts between a box and a sphere.

The first step of our process is to convert the center point of the sphere into the object coordinates of the box. Remember that the box can be oriented in any direction. It will be easier to process if coordinates of the point we're working with are relative to the orientation of the box. We can do this simply by using the following code.

```
——— Excerpt from src/collide_fine.cpp ———
// Transform the center of the sphere into box coordinates
Vector3 center = sphere.getAxis(3);
Vector3 relCenter = box.transform.transformInverse(center);
```

Note that we've taken into account both the orientation of the box and its center.

This then allows us to perform a quick early-out test to see whether the point will be close enough to bother with. The contact generation will be reserved for points that pass this early test. As we shall see, the contact generation algorithm is hardly complex, and we don't gain a huge amount by having the early-out test. Nevertheless it can be worth it in some cases (particularly if the coarse collision detection is inefficient and generates lots of potential collisions).

The early-out test relies on the principle of separating axes.

Separating Axes

Separating axes is one of the most useful concepts in collision detection. It has many nuances, which I can't hope to cover here. See van den Bergen [2003] or Ericson [2005] for a complete discussion and lots of helpful diagrams.

The basic idea is simple: if we can find any direction in space in which two (convex) objects are not colliding, then the two objects are not colliding at all. In our case we can simply test the three axes of the box and check if the point is too far away to be colliding. If any of these axes shows that the point is too far away, then we know there is no contact and we can exit earlier. Figure 13.13 illustrates this in 2D.

Note, however, that we can have a situation where the sphere and the box aren't in contact, but the separating axes test doesn't detect this. A case is shown in the diagram in figure 13.13.

In this case the algorithm will pass on to the more complete contact generation step and will detect that there is no contact later on. We could improve the separating axes test to check further axes in this case (the axis from the center of the box to the center of the sphere in figure 13.13 would help). Remember, however, that this step is designed to give us an early-out. We don't want to waste processing time, and we could spend just as much time trying to determine whether there is a contact in this phase as we'd save by not performing the full contact generation in the next phase.

For each test axis we simply check if the half-width of the box plus the radius of the sphere is greater than one component of the relative position of the sphere center (i.e., the transformed position we found earlier). See the next block of code.

```
                    ── Excerpt from src/collide_fine.cpp ──
// Early-out check to see if we can exclude the contact.
if (real_abs(relCenter.x) - sphere.radius > box.halfSize.x ||
    real_abs(relCenter.y) - sphere.radius > box.halfSize.y ||
    real_abs(relCenter.z) - sphere.radius > box.halfSize.z)
{
    return 0;
}
```
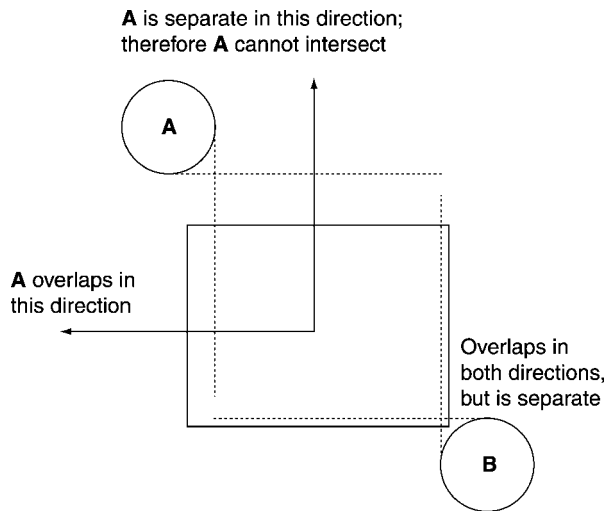

FIGURE 13.13   Separating axes between a box and a sphere.

Contact Generation

The final phase of our algorithm is to find the closest point in the box to the target point and generate the contact from it. We'll do this first in the coordinates of the box.

This is a simple process. All we need to do is to clamp each component of the test point to the half-size of the box in the same direction. With this new point we can then work out the distance from the center of the sphere to the target point, and exit if it is larger than the radius of the sphere. The code for this is simple:

```
                    ── Excerpt from src/collide_fine.cpp ──
Vector3 closestPt(0,0,0);
real dist;

// Clamp each coordinate to the box.
```

```
dist = relCenter.x;
if (dist > box.halfSize.x) dist = box.halfSize.x;
if (dist < -box.halfSize.x) dist = -box.halfSize.x;
closestPt.x = dist;

dist = relCenter.y;
if (dist > box.halfSize.y) dist = box.halfSize.y;
if (dist < -box.halfSize.y) dist = -box.halfSize.y;
closestPt.y = dist;

dist = relCenter.z;
if (dist > box.halfSize.z) dist = box.halfSize.z;
if (dist < -box.halfSize.z) dist = -box.halfSize.z;
closestPt.z = dist;

// Check we're in contact.
dist = (closestPt - relCenter).squareMagnitude();
if (dist > sphere.radius * sphere.radius) return 0;
```

The contact properties need to be given in world coordinates, so before we calculate the contact normal, we need to find the closest point in world coordinates. This simply means transforming the point we generated earlier:

—————————— **Excerpt from src/collide_fine.cpp** ——————————
```
// Compile the contact.
Vector3 closestPtWorld = box.transform.transform(closestPt);
```

We can then calculate the contact properties as before in the chapter. The final code puts all this together to look like this:

—————————— **Excerpt from src/collide_fine.cpp** ——————————
```
unsigned CollisionDetector::boxAndSphere(
    const Box &box,
    const Sphere &sphere,
        CollisionData *data
        )
{
    // Transform the center of the sphere into box coordinates.
    Vector3 center = sphere.getAxis(3);
    Vector3 relCenter = box.transform.transformInverse(center);

    // Early-out check to see if we can exclude the contact.
    if (real_abs(relCenter.x) - sphere.radius > box.halfSize.x ||
        real_abs(relCenter.y) - sphere.radius > box.halfSize.y ||
```

```
        real_abs(relCenter.z) - sphere.radius > box.halfSize.z)
{
    return 0;
}

Vector3 closestPt(0,0,0);
real dist;

// Clamp each coordinate to the box.
dist = relCenter.x;
if (dist > box.halfSize.x) dist = box.halfSize.x;
if (dist < -box.halfSize.x) dist = -box.halfSize.x;
closestPt.x = dist;

dist = relCenter.y;
if (dist > box.halfSize.y) dist = box.halfSize.y;
if (dist < -box.halfSize.y) dist = -box.halfSize.y;
closestPt.y = dist;

dist = relCenter.z;
if (dist > box.halfSize.z) dist = box.halfSize.z;
if (dist < -box.halfSize.z) dist = -box.halfSize.z;
closestPt.z = dist;

// Check we're in contact.
dist = (closestPt - relCenter).squareMagnitude();
if (dist > sphere.radius * sphere.radius) return 0;

// Compile the contact.
Vector3 closestPtWorld = box.transform.transform(closestPt);

Contact* contact = data->contacts;
contact->contactNormal = (center - closestPtWorld);
contact->contactNormal.normalize();
contact->contactPoint = closestPtWorld;
contact->penetration = sphere.radius - real_sqrt(dist);

// Write the appropriate data.
contact->body[0] = box.body;
contact->body[1] = sphere.body;
contact->restitution = data->restitution;
contact->friction = data->friction;
```

```
        return 1;
}
```

13.3.5 COLLIDING TWO BOXES

The collision of two boxes is the most complex case we'll consider in detail in this chapter. Although we're still dealing with a very basic shape, the techniques we need to handle a pair of boxes are the ones that are used when the shapes to collide are more complex. At the end of this section we'll look at how the box–box collision algorithm can be extended to any pair of concave shapes.

There are six possible types of contact between two boxes, as shown in figure 13.14.

We're trying to avoid face–face and edge–face contacts because they are not stable. A single contact point between two planes allows the planes to rotate; it is unlikely that the contact normal will be generated so that it passes through both centers of gravity. If we use three or four contacts instead, then the resulting contact resolution will give a visibly more stable result.

We can always replace a face–face contact with up to four point–face or edge–edge contacts, as shown in figure 13.15. Similarly we can replace face–edge contacts with up to two point–face or edge–edge contacts.

The remaining two contacts, point–point and point–edge, cannot be replaced with others. We could add the code to detect these, but neither of them has an obvious way of calculating the collision normal (there are an infinite number of valid collision normals for each). We'd need some extra hacked code to get a collision normal.

In systems I've built, I have never bothered to do this for two reasons. First, these situations are highly unlikely to come up in practice unless deliberately contrived. The chance of one box colliding with another box perfectly vertex to vertex is very slim indeed: it's a very small target (point–edge is more likely, but still incredibly rare). Second, if we ignore these contacts, an instant later the boxes will interpenetrate slightly. In this case one of the other contacts (normally a point–face contact) will be generated, which is handled normally by the physics. The result is physically believable, so the extra work simply isn't needed.

Ignoring these two types of contact means you can't construct scenarios where boxes are carefully balanced corner to corner or edge to corner. But since that's hardly likely to be a high priority in your game design, you can probably save yourself the work and live with it.

The algorithm for generating contacts between boxes has the same format as that for a sphere and a box, with one additional step:

1. We perform an early-out test using the principle of separating axes. In this case the contact generation is complex enough that this is very likely to be worth the effort.

2. We perform the full collision detection and contact resolution to get a single detected contact between the two boxes.
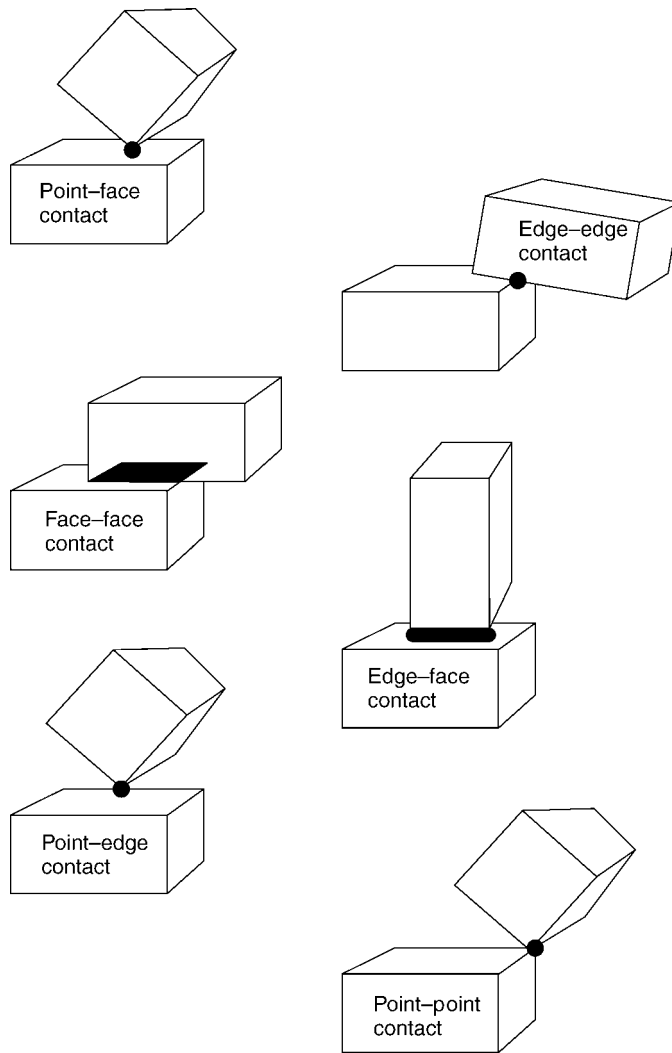
FIGURE 13.14 Contact between two boxes.

3. We combine the newly detected contact with previously detected contacts between the two boxes to form a complete set of contacts

I'll return to the logic behind storing the detected contacts later. First let's look at how to perform a separating axis test on the two boxes.
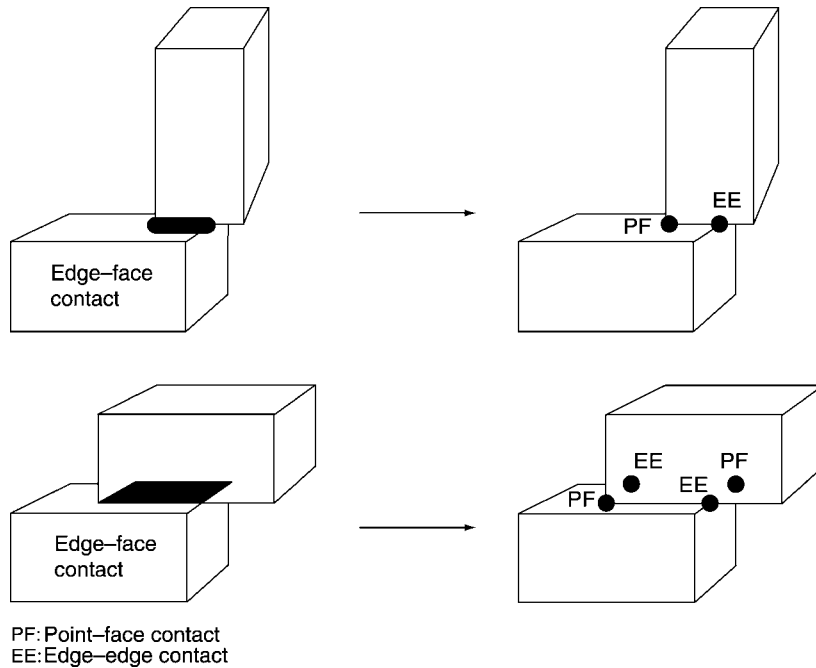
FIGURE 13.15   Replacing face–face and edge–face contacts between boxes.

Separating Axes

The separating axis theorem says that two objects cannot possibly be in contact as long as there is some axis on which the objects can be projected where they are not in contact.

As we saw for the sphere–box case, this check therefore has three parts: first we choose an axis, second we project the objects onto the axis (this was trivial in the case of a sphere, but will be more complex here), and third we check to see whether the projections are overlapping. Figure 13.16 shows this.

If the projections are overlapping on this axis, it does not mean that the objects are touching. But if they are not overlapping, then we know for sure that the objects aren't touching. This acts as an early-out.

This test can be implemented for boxes by projecting the half-size of the box onto the separating axis in this way:

```
──────────── Excerpt from src/collide_fine.cpp ────────────
real transformToAxis(const Box &box,
                     const Vector3 &axis)
{
    return
```

```
            box.halfSize.x * real_abs(axis * box.getAxis(0)) +
            box.halfSize.y * real_abs(axis * box.getAxis(1)) +
            box.halfSize.z * real_abs(axis * box.getAxis(2));
}

bool overlapOnAxis(
    const Box &one,
    const Box &two,
    const Vector3 &axis
    )
{
    // Project the half-size of one onto axis.
    real oneProject = transformToAxis(one, axis);
    real twoProject = transformToAxis(two, axis);

    // Find the vector between the two centers.
    Vector3 toCenter = two.getAxis(3) - one.getAxis(3);

    // Project this onto the axis.
    real distance = real_abs(toCenter * axis);

    // Check for overlap.
    return (distance < oneProject + twoProject);
}
```

For convex objects we can go one stage farther: if the objects are not touching, then there must be at least one axis that would show this. Objects that are concave in
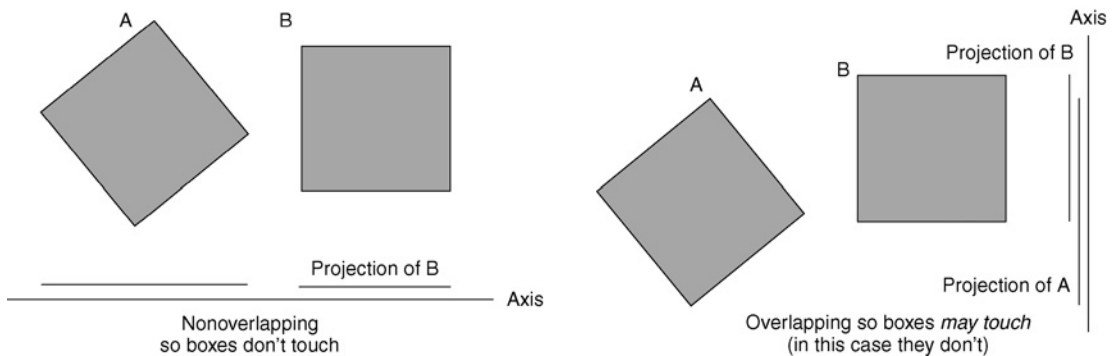
FIGURE 13.16   The projection of two boxes onto separating axes.

places can pass every separating axis test and still not be touching. For general objects this isn't particularly practical knowledge, of course, because we can't hope to test every, possible axis. That would defeat the purpose of using this test to provide an early-out.

When colliding two boxes, things are easier for us. There are fifteen axes we can test. If none of the fifteen shows the objects separately, then we know the objects touch.

These axes include the three principal axes of each box; there are also another nine axes to test that are perpendicular to each pair of principal axes from each box. We generate these further nine axes by taking the cross product of each pair of principal axes (since the direction of the cross product is perpendicular to both its vectors).

The full implementation simply calculates these axes in turn and sends them to the `separateAlongAxis` function for checking. The whole test returns false when the first such function call fails.

Coherence and Contact Generation

The most efficient algorithms for calculating the collision between two boxes (and any pair of convex objects) use shortcuts and optimizations that terminate the algorithm when the point of deepest interpenetration is found. This is a single contact.

To generate a complete set of contacts is more difficult. One contact can be interpreted in several ways. A point–face contact could be interpreted as another point–face contact on a different face, with a different penetration depth. The situation is even more complex with edge–edge contacts.

To get the contact set, we need to provide a whole set of reasonably complex code that checks to see whether new potential contacts have already been found and, if so, whether the new way of interpreting them is better. And in the worst-case scenario reinterpreting one contact may mean that all the other contacts in the set need reinterpreting. If there weren't a simpler way, this might be worth the effort, but fortunately we can avoid the whole issue.

At each frame we generate a single contact, based on the maximum penetration of the two boxes. This is resolved in the normal way. Because we only have one contact resolved, the boxes will likely reinterpenetrate in the next frame but often in a different way. This new interpenetration is detected as a new contact, so now we have two contacts. This continues a third time, when we have three contacts, sufficient to keep the two boxes stacked in a stable manner. Figure 13.17 shows this in action for two contacts in two dimensions.

The chances that an existing contact will be useful in the next frame are high when the boxes are stable, but when they are moving, the contact may be completely wrong at the following frame.

To avoid having to throw away the contacts when boxes are moving (which would mean we'd be back to generating just a single contact at each frame), we don't store the complete contact but only the features that are touching.

We've met features throughout this chapter: point, edge, and face. A point–edge contact is a contact between a certain vertex and a certain face. To take advantage
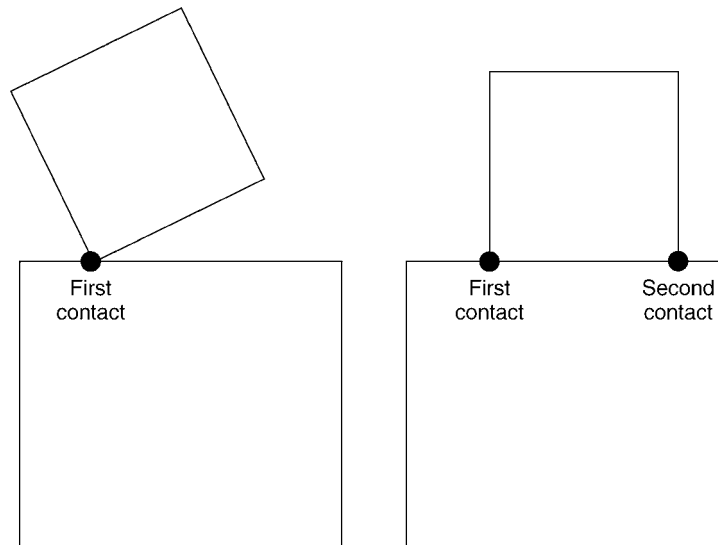
FIGURE 13.17   Sequence of contacts over two frames.

of the coherence between contacts, we store the type of contact (in our case only point–edge and edge–edge) and which particular feature was involved for each object. Figure 13.17 shows this in action. We have a point–face contact, and in the next frame we have the same contact (i.e., a contact between the same point and face) but with different contact properties.

If the deepest-penetration algorithm returns a contact that we already have, then we update the data for that contact. Otherwise the new contact is added to the contacts we already have cached for those two objects, and the complete set is sent to the collision resolver algorithm.

For each contact in our cache we need to update the values that change (i.e., the contact normal and the interpenetration depth). If we find that the interpenetration depth is larger than some fixed value, then the contact has moved apart by some distance, and thus it makes no sense to consider it further. So we remove contacts from the cache if, when updated, they have moved too far apart.

How far is too far? If we strictly use a value of zero, then we are doing too much work. When a contact is resolved, its interpenetration is brought back to zero or less; and we don't want the contact resolution to cause us to remove contacts from the cache because we'd be back to the one-contact-per-frame case. As we'll see when we look at contact resolution, it can be helpful to consider contacts that aren't interpenetrating when trying to resolve a whole set of contacts on one object.

So we use a small negative value. Getting the right value is, unfortunately, a matter of tweaking.

Contact Generation

So we've reduced the complex task of generating a set of contacts between two boxes to the problem of finding the point of deepest interpenetration. In addition, however, we find that the point of deepest penetration has to be able to return the features on which the contact takes place.

A range of algorithms can help us do this for both boxes and general convex objects. V-Clip is one of the best known, but the detection phase of Lin-Canny, and Gilbert, Johnson, and Keerthi's algorithm (GJK) are also useful. V-Clip is described in detail in Ericson [2005], while GJK is more comprehensively described in van den Bergen [2003], reflecting each author's use of their favorite technique in their collision detection libraries.

All these algorithms are *distance* algorithms: they can return the shortest distance between two polyhedra. In our case this is useful when the distance returned is less than zero: the objects are interpenetrating. Lin-Canny in particular has problems with this case; it needs to be tweaked to catch the interpenetration case and avoid an infinite loop.

Both V-Clip and GJK intelligently search the possible combinations of features that can be in contact. They are sufficiently intelligent about this that there is little speed advantage in performing a separating axis test as an early-out for boxes.

To build a robust and efficient collision detection system for general polyhedra, you will eventually need to use something like these techniques. Be careful, however. Various efficient collision detection algorithms, including V-Clip and some of its variants, are protected by patents. If you implement these algorithms in your code, you may need to get a license agreement or pay a licensing fee.

To allow us to test out the physics without writing a full collision detection system, we can build a naive algorithm that checks all possible interpenetrations. This is workable for a box with six faces, eight vertices, and twelve edges but shouldn't be considered production-ready for anything more complex.

Our naive contact generator simply checks for both of the contact types we are interested in: point–face contacts and edge–edge contacts. If it finds more than one such contact, then the contact that has the greatest interpenetration is returned. Remember that we aren't returning multiple contacts, because getting a self-consistent set of such contacts can be difficult.

To exhaustively check for contacts we perform two kinds of check: a point–face check of each vertex on each object against the other object, and an edge–edge check of each edge on one object against the other.

Point–Face Contact

Point–face contacts are easy to detect: they use the same logic as for sphere–box collisions but with a zero-radius sphere. We simply project each vertex from one box into the principal axes of the other. If the vertex is inside the box, then a point–face collision is required. If the vertex is inside the box on more than one axis, then the axis with the shallowest penetration is used. Figure 13.18 shows this.

Shallow interpenetration

Deep interpenetration

Deep interpenetration

Deep interpenetration
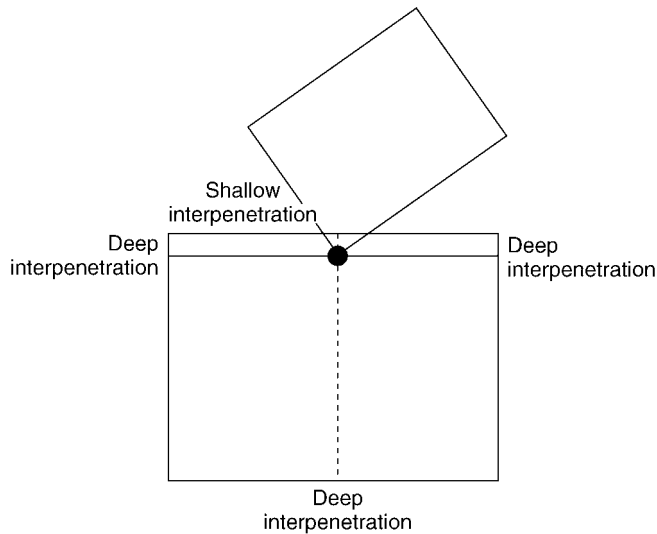
FIGURE 13.18   Projection of a point–face contact.

The algorithm works in this way:

1. Consider each vertex of object A.
2. Calculate the interpenetration of that vertex with object B.
3. The deepest such interpenetration is retained.
4. Do the same with object B's vertices against object A.
5. The deepest interpenetration overall is retained.

The point–face detection code therefore looks like this:

```
_____ Excerpt from src/collide_fine.cpp _____
unsigned CollisionDetector::boxAndPoint(
    const Box &box,
    const Vector3 &point,
    CollisionData *data
    )
{
    // Transform the point into box coordinates.
    Vector3 relPt = box.transform.transformInverse(point);

    Vector3 normal;

    // Check each axis, looking for the axis on which the
```

```
    // penetration is least deep.
    real min_depth = box.halfSize.x - real_abs(relPt.x);
    if (min_depth < 0) return 0;
    normal = box.getAxis(0) * ((relPt.x < 0)?-1:1);

    real depth = box.halfSize.y - real_abs(relPt.y);
    if (depth < 0) return 0;
    else if (depth < min_depth)
    {
        min_depth = depth;
        normal = box.getAxis(1) * ((relPt.y < 0)?-1:1);
    }

    depth = box.halfSize.z - real_abs(relPt.z);
    if (depth < 0) return 0;
    else if (depth < min_depth)
    {
        min_depth = depth;
        normal = box.getAxis(2) * ((relPt.z < 0)?-1:1);
    }

    // Compile the contact.
    Contact* contact = data->contacts;
    contact->contactNormal = normal;
    contact->contactPoint = point;
    contact->penetration = min_depth;

    // Write the appropriate data.
    contact->body[0] = box.body;

    // Note that we don't know what rigid body the point
    // belongs to, so we just use NULL. Where this is called
    // this value can be left, or filled in.
    contact->body[1] = NULL;

    contact->restitution = data->restitution;
    contact->friction = data->friction;

        return 1;
}
```

Notice that we generate the contact data for the point–face collision exactly as we did in the sphere–box test.

Edge–Edge Contact Generation

Determining edge–edge contacts is not much more complex. We again take each edge from one object in turn and check it against the other object. In this case we check it against edges in the other object.

We can easily calculate the distance between the two edges, but the distance alone doesn't tell us whether the edges are separated by that distance or interpenetrating by that distance. We need to add an extra check. The distance calculation also provides us with the point on each edge that is closest to the other edge. If the edges are interpenetrating, then this point on object A will be closer to the center of object B than object B's edge point, and vice versa. Figure 13.19 illustrates this. We use this check to determine the sign of the distance and to work out whether it is interpenetrating or not.

Just as we saw in figure 13.18 for point–face contacts, an edge–edge contact will be detected for multiple edges. We only need to use the shallowest edge–edge contact. The algorithm works in this way:

1. Consider each edge E of object A.

2. Work out its interpenetration with each edge of object B.

3. The shallowest such interpenetration is the interpenetration of E.

4. The edge from object A with the deepest such interpenetration is retained.

We don't need to repeat the process from object B's point of view because we are checking edges against edges; we will have checked all edge combinations already.
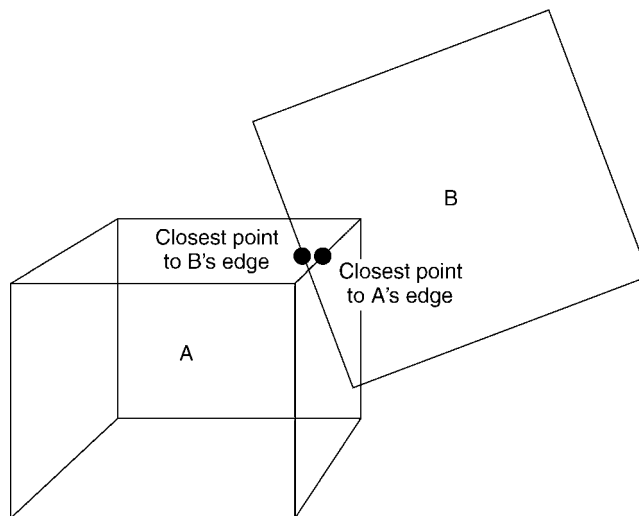
FIGURE 13.19   Determining edge–edge contacts.

The Final Contact

Now we have a winner from the point–face calculations and a winner from the edge–edge calculations. The contact that gets returned will be the deeper of these two options. This can then be fed into the caching algorithm described previously to generate a complete set of contacts over successive frames.

13.3.6 EFFICIENCY AND GENERAL POLYHEDRA

Obviously the naive contact generation algorithm described earlier is considerably less efficient than V-Clip, GJK, or the handful of other variations used in comprehensive collision detection libraries.

You'll find this algorithm on the CD, along with a more efficient general-purpose contact generation routine that you can simply copy and paste into your own code.

Neither the naive algorithm nor its more efficient cousins are limited to contact generation between boxes. They are general-purpose algorithms that can generate contacts between any convex 3D objects made up of flat surfaces. This is convenient because most game assets are made up of polygons (although native curved surfaces are becoming increasingly common).

As the number of faces increases, however, the algorithms slow down by an increasingly large amount. The naive algorithm shows this particularly acutely; for anything more than a simple box it performs very badly. But both V-Clip and GJK also suffer from the same problem.

For this reason, even when using a very efficient collision detection algorithm with a comprehensive coarse collision detection layer, it is not advisable to use the same set of geometry for collision detection as you do for rendering. Even complex 3D assets can normally be represented to the collision detector as either an assembly of primitives or an assembly of simply convex polyhedra. A general polyhedra contact generator, combined with the caching system we've built, can then generate a set of contacts that will lead to realistic physics in a reasonable amount of time.

13.4 SUMMARY

These collision detection cases only scratch the surface of what is possible (and what may be needed) in a full game engine. Collision detection, and particularly contact generation, is a large field in its own right, and although the physics engine relies on it, it is a quite separate piece of software engineering.

You can use the collision detection system we've built in the last two chapters in its own right, and extend it with the additional tests you need (the other books in this series contain lots of collision detection tests). Or you can elect to bring in a third-party collision detection system.

There are good open source collision detection and contact generation algorithms available (such as SOLID and RAPID). You can use these alongside the physics you're

developing in this book, or take their algorithms as inspiration for creating your own code.

Be warned though: There are so many cases, and so many optimizations possible, that it will end up a bigger job to write a comprehensive collision detection system than it will be to create your physics engine to process the contacts.

With a collision detection system running correctly, it's time to return to our physics engine and process the collisions that we've found. Chapter 14 continues to cover the issue of impact collisions and shows how to build code to support collisions for full rotating rigid bodies.

PART V

Contact Physics

This page intentionally left blank
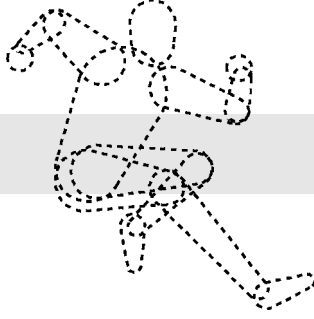
14

COLLISION
RESOLUTION

I t's time to look at the final, and most complex, stage of our physics system. We
have a set of contact data from the collision detector (from chapter 13), and we
have the rigid-body equations of motion, including torques and forces (from chap-
ter 10). We are now ready to combine the two and have rotating objects respond to
contacts.

Just as in chapter 7, we will first look in detail at the physics of collisions. We are
building a micro-collision physics engine, one in which resting contacts are handled
with numerous mini-collisions (plus a bit of extra special-purpose code). Before we
can get the micro-collisions of resting contacts working, we need to look in detail at
basic collision handling.

This chapter builds the first stage of our contact resolution system to handle colli-
sions. Chapter 15 goes on to incorporate the collision response into more general and
robust contact handling.

Because all contact handling in the engine is based on collisions, this chapter takes
up the largest part of finishing our engine, in terms of book pages, of mathematical
complexity, and of implementation difficulty. If you find this chapter hard going, then
try to persevere: it's mostly downhill from here on.

14.1 IMPULSES AND IMPULSIVE TORQUES

Recall that when a collision occurs between two objects in the real world, the material
from which they are made compresses slightly. Whether it is a rubber ball or a stone,
the molecules near the point of collision are pushed together fractionally. As they
compress they exert a force to try to return to their original shape.

Different objects have a different resistance to being deformed and a different tendency to return to their original shape. Combined together the two tendencies give an object its characteristic bounce. A rubber ball can be easily deformed but has a high tendency to return to its original shape, so it bounces well. A stone has a high tendency to return to its original shape but has a very high resistance to being deformed: it will bounce, but not very much. A lump of clay will have a low resistance to being deformed and no tendency to return to its original shape: it will not bounce at all.

The force that resists the deformation causes the objects to stop colliding: their velocities are reduced until they are no longer moving together. At this point the objects are at their most compressed. If there is a tendency to return to their original shape, then the force begins to accelerate them apart until they are no longer deformed. At this point the objects are typically moving apart.

All this happens in the smallest fraction of a second, and for reasonably stiff objects (such as two pool balls) the compression distances are tiny fractions of a millimeter. In almost all cases the deformation cannot be seen with the naked eye; it is too small and over too quickly. From our perspective we simply see the two objects collide and instantly bounce apart.

I have stressed what is happening at the minute level because it makes the mathematics more logical. It would be impractical for us to simulate the bounce in detail, however. The compression forces are far too stiff, and as we've seen with stiff springs, the results would be disastrous.

In chapter 7 we saw that two point objects will bounce apart at a velocity that is a fixed multiple of their closing velocity immediately before the impact. To simulate this we instantly change the velocity of each object in the collision. The change in velocity is called an "impulse."

14.1.1 Impulsive Torque

Now that we are dealing with rotating rigid bodies, things are a little more difficult. If you bounce an object that is spinning on the ground, you will notice that the object not only starts to move back upward, but its angular velocity will normally change too.

It is not enough to apply the collision equations from chapter 7 because they only take into account linear motion. We need to understand how the collision affects both linear and angular velocities.

Figure 14.1 shows a long rod being spun into the ground (we'll come back to collisions between two moving objects in a moment). Let's look closely at what would happen in the real world at the moment of collision. The second part of the figure shows the deformation of the object at the point of collision. This causes a compression force to push in the direction shown.

Looking at D'Alembert's principle in chapter 10 we saw that any force acting on an object generates both linear and angular acceleration. The linear component is
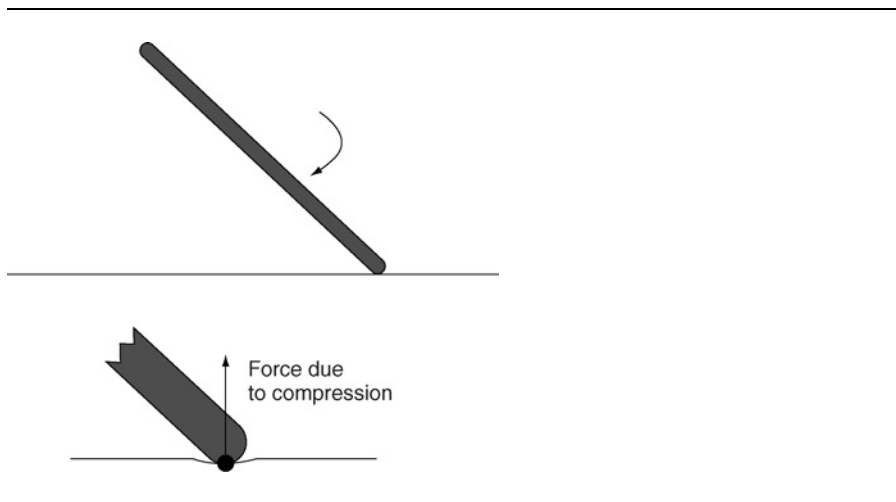
FIGURE 14.1   The rotational and linear components of a collision.

given by

$$\ddot{p} = \frac{1}{m}f$$

and the angular component by the torque

$$\tau = p_f \times f$$

where the torque generates angular acceleration by

$$\ddot{\theta} = I^{-1}\tau$$

which is equation 10.3 from chapter 10.

In the case of the collision it stands to reason that the collision will generate a linear change in velocity (the impulse) and an angular change in velocity. An instantaneous angular change in velocity is called an "impulsive torque" (also rarely called "moment of impulse" or "impulsive moment," which sounds more like a drunken Vegas wedding to me).[1]

In the same way as we have

$$\tau = I\ddot{\theta}$$

for torques we have

$$u = I\dot{\theta}$$

---

1. Strictly speaking, what we've called impulse is "impulsive force." We could also call it "linear and angular impulse," but I'll continue to use just "impulse" to refer to the linear version.

where $\boldsymbol{u}$ is the impulsive torque, I is the inertia tensor, and $\dot{\boldsymbol{\theta}}$ is the angular velocity, as before. This is the direct equivalent of equation 7.5, which dealt with linear impulses. And correspondingly the change in angular velocity $\Delta\dot{\boldsymbol{\theta}}$ is

$$\Delta\dot{\boldsymbol{\theta}} = I^{-1}\boldsymbol{u} \qquad\qquad [14.1]$$

In all these equations I should be in world coordinates, as discussed in section 10.2.3.

Impulses behave just like forces. In particular for a given impulse there will be both a linear component and an angular component. Just as the amount of torque is given by

$$\boldsymbol{\tau} = \boldsymbol{p}_f \times \boldsymbol{f}$$

so the impulsive torque generated by an impulse is given by

$$\boldsymbol{u} = \boldsymbol{p}_f \times \boldsymbol{g} \qquad\qquad [14.2]$$

In our case, for collisions the point of application ($\boldsymbol{p}_f$) is given by the contact point and the origin of the object:

$$\boldsymbol{p}_f = \boldsymbol{q} - \boldsymbol{p}$$

where $\boldsymbol{q}$ is the position of the contact in world coordinates and $\boldsymbol{p}$ is the position of the origin of the object in world coordinates.

14.1.2 ROTATING COLLISIONS

The effect of the impulse at the collision is to have the points of each object in collision bounce apart. The movement of the colliding objects at the collision point still follows the same equations that we met in chapter 7. In other words, if we tracked the two collision points (one from each object) around the time of the collision, we'd see that their separating velocity is given by

$$v_s' = -cv_s$$

where v_s is the relative velocity of the objects immediately before the collision, v_s' is the relative velocity after the collision, and c is the coefficient of restitution. In other words, the separation velocity is always in the opposite direction to the closing velocity, and is a constant proportion of its magnitude. The constant c depends on the materials of both objects involved.

Depending on the characteristics of the objects involved, and the direction of the contact normal, this separation velocity will be made up of a different degree of linear and rotational motion. Figure 14.2 shows different objects engaged in the same collision (again illustrated with an unmoving ground for clarity). In each part of the
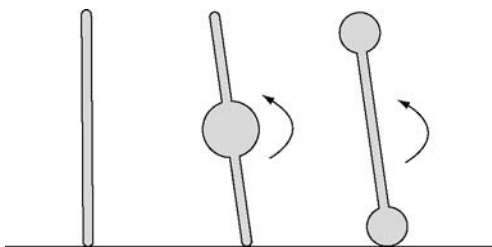
FIGURE 14.2   Three objects with different bounce characteristics.

figure the closing velocity and the coefficient of restitution at the point of contact are the same, so the separating velocity is the same too.

The first object is lightweight and is colliding almost head on. For any force that is generated during the collision the corresponding torque will be small, because f is almost parallel to p_f. Its bounce will be mostly linear, with only a small rotational component.

The second object is heavier but has a very small moment of inertia about the Z axis. It is colliding off center. Here the torque generated will be large, and because the moment of inertia is very small, there will be a big rotational response. The rotational response is so large, in fact, that the linear component isn't large enough to bounce the object upward. Although the point of contact bounces back up (at the same velocity as the point of contact in each other case), it is the rotation of the object that is doing most of the separating, so the linear motion continues downward at a slightly slower rate. You can observe this if you drop a ruler on the ground in the configuration shown in figure 14.2. The ruler will start spinning away from the point of contact rapidly, but as a whole it will not leap back into the air. The rotation is taking the bulk of the responsibility for separating the points of contact.

The third object in the figure collides in the same way as the second. In this case, however, although the mass is the same, its moment of inertia is much greater. It represents an object with more mass in its extreme parts. Here the compression force causes a much lower amount of rotation. The linear impulse is greater and the impulsive torque is smaller. The object bounces linearly and the compression force reverses the direction of rotation, but the resulting angular velocity is very small.

14.1.3 Handling Rotating Collisions

Just as for particle collisions, we need two parts to our collision response. First we need to resolve the relative motion of the two objects by applying impulse and impulsive torque. When we process a collision for velocity, we need to calculate four values: the impulse and impulsive torque for both objects in the collision. Calculating the balance of linear and angular impulse to apply is a complex task and involves some complicated mathematics, as we'll see in the next section.

Because we only check for collisions at the end of each frame, objects may have already passed into one another. So, second, we need to resolve any interpenetration that has occurred. The interpenetration can be handled in a very similar way to interpenetration for particles. But the impulse calculations we need to do anyway allow us to derive a more physically realistic interpenetration resolution. We'll return to this process in section 14.3

14.2   Collision Impulses

To resolve the relative motion of the two objects we need to calculate four impulses: linear and angular impulses for each object. If there is only one object involved in the collision (if an object is colliding with an immovable object, such as the ground), then we need only two values: the impulse and impulsive torque for the single object.

To calculate the impulse and impulsive force on each object we go through a series of steps:

1. We work in a set of coordinates that are relative to the contact. This makes much of the mathematics a lot simpler. We create a transform matrix to convert into and out of this new set of coordinates.

2. We work out the change in velocity of the contact point on each object per unit impulse. Because the impulse will cause linear and angular motion, this value needs to take account of both components.

3. We will know the velocity change we want to see (in the next step), so we invert the result of the last stage to find the impulse needed to generate any given velocity change.

4. We work out what the separating velocity at the contact point should be, what the closing velocity currently is, and the difference between the two. This is the desired change in velocity.

5. From the desired change in velocity we can calculate the impulse that must be generated.

6. We split the impulse into its linear and angular components and apply them to each object.

Let's look at each of these stages in turn.

14.2.1   Change to Contact Coordinates

Our goal is to work out what impulse we need to apply as a result of the collision. The impulse will generate a change in velocity, and we need to find the impulse that generates the change in velocity we are looking for.

We are not interested in the linear and angular velocity of the whole object at this stage. For the purpose of the collision we are only interested in the separating velocity
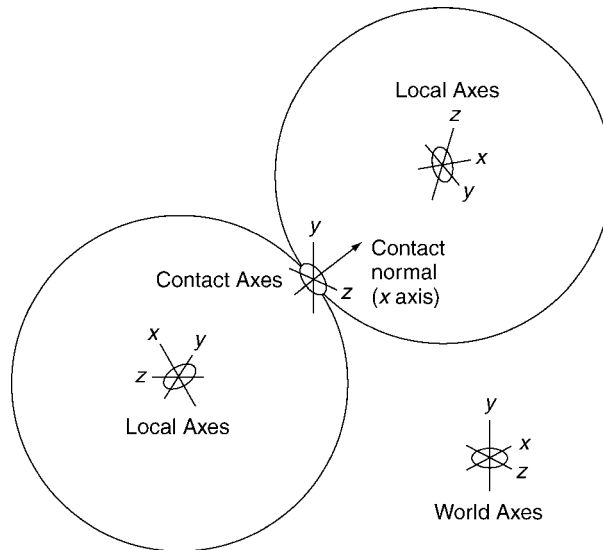
FIGURE 14.3   The three sets of coordinates: world, local, and contact.

of the contact points. As we saw in the previous section, we have an equation that tells us what the final separating velocity needs to be, so we'd like to be able to apply it simply.

The velocity of a point on an object is related to both its linear and angular velocity, according to equation 9.5:

$$\dot{q} = \dot{\theta} \times (q - p) + \dot{p}$$

Because we are only interested in the movement of the colliding points at this stage, we can simplify the mathematics by doing calculations relative to the point of collision.

Recall from chapter 13 that each contact has an associated contact point and contact normal. If we use this point as the origin, and the contact normal as one axis, we can form an orthonormal basis around it: a set of three axes. Just as we have a set of world coordinates and a set of local coordinates for each object, we will have a set of contact coordinates for each contact.

Figure 14.3 shows the contact coordinates for one contact. Notice that we are ignoring interpenetration at this stage. As part of the contact generation, we calculated a single representative point for each contact.

The Contact Coordinate Axes

The first step of converting to contact coordinates is to work out the direction of each axis. We do this using the algorithm to calculate an orthonormal basis, as shown in

section 2.1.9. The X axis we know already: it is the contact normal generated by the collision detector. The Y axis and Z axis need to be calculated.[2]

Unfortunately there can be any number of different Y and Z axes generated from one X axis. We'll need to select just one. If we are working with anisotropic friction (friction that is different in different directions), then there will be one set of basis vectors that is most suitable. For the isotropic friction in this book, and for frictionless simulations, any set is equally valid. Since we are ignoring friction for now, we create an arbitrary set of axes by starting with the Y axis pointing down the world Y axis (recall that the algorithm required the base axis, in our case the X axis, plus an initial guess at a second axis, which may end up being altered):

```
/**
 * Creates an orthonormal basis where the x-vector is given
 * and the y-vector is suggested, but can be changed. Both
 * y and z vectors are written to in this function. We assume
 * that the vector x is normalized when this function is called.
 */
void makeOrthonormalBasis(const Vector &x, Vector *y, Vector *z)
{
    // Calculate z from the vector product of x and y.
    z = (*x) % (*y);

    // Check for y and z in parallel
    if (z->squaredMagnitude() == 0.0) return;

    // Calculate y from the vector product of z and x.
    (*y) = (*z) % x;

    // Normalize the output vectors
    y->normalize();
    z->normalize();
}
```

This follows the algorithm given in section 2.1.9. The Y axis assumption is provided when the function is called:

```
Vector y(0, 1.0, 0), z;
makeOrthonormalBasis(contactNormal, &y, &z);
```

---

2. This is just a convention adopted in this book. There is no reason why the X axis has to be the contact normal. Some people prefer to think of the Y axis as the contact normal. If you are one of them, the rest of this section can be adjusted accordingly.

This algorithm can't cope when the x and y vectors are parallel. We can easily modify the algorithm to do that, rather than returning when check for zero fails. We could use a different y value (by inverting one or swapping two of its non-zero elements, for example) and recalculate z. We will return to this issue later, after making some improvements to the calculation code itself.

We can improve the efficiency of this longhand form by manually performing the vector products (rather than calling the vector product operator in the Vector3 class). First notice that if the initial Y axis is pointing along the Y axis, then any value for the resulting Z axis must be at right angles to the Y axis. This can only happen if the resulting Z axis has a zero Y component.

Second, rather than normalizing the vectors at the end, we can ensure that they are normalized as we go. We do this by making sure that the calculation of the Z axis ends with a normalized vector. Because the vector product obeys the equation

$$|y| = |z \times x| = |z||x| \sin \theta$$

and we know the X and Z axes are at right angles ($\sin \theta = 1$) and both are normalized ($|z| = |x| = 1$) then the resulting Y axis must have a magnitude of 1: it is normalized. The shorthand code looks like this:

```
// The output axes
Vector y, z;

// Scaling factor to ensure the results are normalized.
const real s = 1.0/real_sqrt(x.z*x.z + x.x*x.x);

// The new Z axis is at right angles to the world Y axis.
z.x = x.z*s;
z.y = 0;
z.z = -x.x*s;

// The new Y axis is at right angles to the new X and Z axes.
y.x = x.y*z.x;
y.y = x.z*z.x - x.x*z.z;
y.z = -x.y*z.x;
```

There is one further problem to address. If the contact normal passed in (as the X axis) is already pointing in the direction of the world-space Y axis, then we will end up with zero for all three components of the Z axis. In this case, using the world-space Y axis is not a good guess; we need to use another. We can use either the world-space X axis or Z axis. The code I've implemented uses the world-space X axis.

To make the algorithm as stable as possible we switch between using the world-space Y axis and the X axis as a best guess, depending on which the contact normal

is nearest to. We could just switch when the contact normal is exactly in the direction of the Y axis, but if it were very close to being in that direction, we might end up with numerical problems and an inaccurate result.

```
if (real_abs(x.x) > real_abs(x.y))
{
  // We're nearer the X axis, so use the Y axis as before.
  // ...
}
else
{
  // We're nearer the Y axis, so use the X axis as a guess.
  // ...
}
```

The Basis Matrix

Before we look at the complete code for calculating the basis, we need to review what it needs to output. So far I've assumed we'll end up with three vectors that make up an orthonormal basis.

It is often more convenient to work with a matrix rather than a set of three vectors. Recall from section 9.4.2 that a matrix can be thought of as a transformation from one set of axes to another.

At several points in this section we will need to convert between the contact axes (called "contact coordinates" or "contact space") and world space. To do this we need a matrix that performs the conversion.

We saw in section 9.4.2 that a transform matrix from local space into world space can be constructed by placing the three local-space axes as columns in the matrix. So, if we have an orthonormal basis consisting of the three vectors

$$\widehat{x}_{local} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \qquad \widehat{y}_{local} = \begin{bmatrix} d \\ e \\ f \end{bmatrix}, \quad \text{and} \quad \widehat{z}_{local} = \begin{bmatrix} g \\ h \\ i \end{bmatrix}$$

we can combine them into a transform matrix:

$$M_{basis} = \begin{bmatrix} a\ d\ g \\ b\ e\ h \\ c\ f\ i \end{bmatrix}$$

If we have a set of coordinates expressed in local space, and we want the coordinates of the same point in world space, we can simply multiply the transform matrix

by the coordinate vector:

$$M\boldsymbol{p}_{\text{local}} = \boldsymbol{p}_{\text{world}}$$

In other words, the basis matrix converts local coordinates to world coordinates.

We can put this together into code. This function operates on the contact normal to create a set of orthonormal axes and then generates a basis matrix representing the contact coordinate scheme. The matrix can act as a transformation to convert contact coordinates into world coordinates:

```
────────── Excerpt from src/contacts.cpp ──────────

/**
 * Constructs an arbitrary orthonormal basis for the contact.
 * This is stored as a 3x3 matrix, where each vector is a column
 * (in other words the matrix transforms contact space into world
 * space). The x direction is generated from the contact normal,
 * and the y and z directions are set so they are at right angles to
 * it.
 */
void Contact::calculateContactBasis()
{
    Vector3 contactTangent[2];

    // Check whether the Z axis is nearer to the X or Y axis.
    if(real_abs(contactNormal.x) > real_abs(contactNormal.y))
    {
        // Scaling factor to ensure the results are normalized.
        const real s = (real)1.0f/
            real_sqrt(contactNormal.z*contactNormal.z +
            contactNormal.x*contactNormal.x);

        // The new X axis is at right angles to the world Y axis.
        contactTangent[0].x = contactNormal.z*s;
        contactTangent[0].y = 0;
        contactTangent[0].z = -contactNormal.x*s;

        // The new Y axis is at right angles to the new X and Z axes.
        contactTangent[1].x = contactNormal.y*contactTangent[0].x;
        contactTangent[1].y = contactNormal.z*contactTangent[0].x -
            contactNormal.x*contactTangent[0].z;
        contactTangent[1].z = -contactNormal.y*contactTangent[0].x;
    }
    else
    {
        // Scaling factor to ensure the results are normalized.
        const real s = (real)1.0/
```

```
            real_sqrt(contactNormal.z*contactNormal.z +
            contactNormal.y*contactNormal.y);

        // The new X axis is at right angles to the world X axis.
        contactTangent[0].x = 0;
        contactTangent[0].y = -contactNormal.z*s;
        contactTangent[0].z = contactNormal.y*s;

        // The new Y axis is at right angles to the new X and Z axes.
        contactTangent[1].x = contactNormal.y*contactTangent[0].z -
            contactNormal.z*contactTangent[0].y;
        contactTangent[1].y = -contactNormal.x*contactTangent[0].z;
        contactTangent[1].z = contactNormal.x*contactTangent[0].y;
    }

    // Make a matrix from the three vectors.
    contactToWorld.setComponents(
        contactNormal,
        contactTangent[0],
        contactTangent[1]);
}
```

where the setComponents method of the Matrix3 class sets the columns in the matrix.
It is implemented as

```
                  ___ Excerpt from include/cyclone/core.h ___
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

    /**
     * Sets the matrix values from the given three vector components.
     * These are arranged as the three columns of the vector.
     */
    void setComponents(const Vector3 &compOne, const Vector3 &compTwo,
        const Vector3 &compThree)
    {
        data[0] = compOne.x;
```

```
        data[1] = compTwo.x;
        data[2] = compThree.x;
        data[3] = compOne.y;
        data[4] = compTwo.y;
        data[5] = compThree.y;
        data[6] = compOne.z;
        data[7] = compTwo.z;
        data[8] = compThree.z;


    }
};
```

The Inverse Transformation

It is worth recapping the result we saw in section 9.4.3 here—namely, that the inverse of a rotation matrix is the same as its transpose. Why are we interested in the inverse? Because, in addition to converting from contact coordinates to world coordinates, we may have to go the other way as well.

To convert world coordinates into contact coordinates we use the inverse of the basis matrix created in the previous code. Inverting a matrix in general, as we have seen, is complex. Fortunately the basis matrix as we've defined it here represents a rotation only: it is a 3×3 matrix, so it can't have a translational component; and because both the contact axes and the world axes are orthonormal, there is no skewing or scaling involved.

This means that we can perform the transformation from world coordinates into contact coordinates by using the transpose of the basis matrix:

$$M_{\text{basis}}^{\top} = \begin{bmatrix} a\ d\ g \\ b\ e\ h \\ c\ f\ i \end{bmatrix}^{\top} = \begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix}$$

This important result allows us to convert at will between contact coordinates and world coordinates.

Whenever some calculation is easier in one than another, we can simply convert between them. We'll use this important result in the next section, and a great deal more in the next chapter.

14.2.2 VELOCITY CHANGE BY IMPULSE

Remember that the change in motion of both objects in a collision is caused by the forces generated at the collision point by compression and deformation. Because we are representing the whole collision event as a single moment in time, we use impulses rather than forces. Impulses cause a change in velocity (both angular and linear, according to D'Alembert's principle, just like forces).

So, if our goal is to calculate the impulse at the collision, we need to understand what effect an impulse will have on each object. We want to end up with a mathematical structure that tells us what the change in velocity of each object will be for any given impulse.

For the frictionless contacts we're considering in this chapter, the only impulses generated at the contact are applied along the contact normal. We'd like to end up with a simple number, then, that tells us the change in velocity at the contact, in the direction of the contact normal, for each unit of impulse applied in the same direction.

As we have seen, the velocity change per unit impulse has two components: a linear component and an angular component. We can deal with these separately and combine them at the end.

It is also worth noting that the value depends on both bodies. We'll need to find the linear and angular velocity change for each object involved in the collision.

The Linear Component

The linear component is very simple. The linear change in velocity for a unit impulse will be in the direction of the impulse, with a magnitude given by the inverse mass:

$$\Delta \dot{p}_d = m^{-1}$$

For collisions involving two objects, the linear component is simply the sum of the two inverse masses:

$$\Delta \dot{p}_d = m_a^{-1} + m_b^{-1}$$

Remember that this equation holds only for the linear component of velocity—they are not the complete picture yet!

The Angular Component

The angular component is more complex. We'll need to bring together three equations we have met at various points in the book. For convenience we'll use q_{rel} for the position of the contact relative to the origin of an object:

$$q_{\mathrm{rel}} = q - p$$

First, equation 14.2 tells us the amount of impulsive torque generated from a unit of impulse:

$$u = q_{\mathrm{rel}} \times \widehat{d}$$

where d is the direction of the impulse (in our case the contact normal).

Second, equation 14.1 tells us the change in angular velocity for a unit of impulsive torque:

$$\Delta \dot{\theta} = I^{-1} u$$

And finally, equation 9.5 tells us the total velocity of a point. If we remove the linear component, we get the equation for the linear velocity of a point due only to its rotation:

$$\dot{q} = \dot{\theta} \times q_{\text{rel}}$$

The rotation-induced velocity of a point ($\dot{q}$) depends on its position relative to the origin of the object ($q - p$) and on the object's angular velocity ($\dot{\theta}$).

So we now have a set of equations that can get us from a unit of impulse, via the impulsive torque it generates and the angular velocity that the torque causes, through to the linear velocity that results.

Converting these three equations into code, we get

```
Vector3 torquePerUnitImpulse =
    relativeContactPosition % contactNormal;

Vector3 rotationPerUnitImpulse =
    inverseInertiaTensor.transform(torquePerUnitImpulse);

Vector3 velocityPerUnitImpulse =
    rotationPerUnitImpulse % relativeContactPosition;
```

The result will be the velocity caused by rotation per unit impulse. As it stands, the result is a vector: it is a velocity in world space. We are only interested in the velocity along the contact normal.

We need to transform this vector into contact coordinates using the transpose basis matrix we saw earlier. This would give us a vector of velocities that a unit impulse would cause. We are only interested at this stage in the velocity in the direction of the contact normal. In contact coordinates this is the X axis, so our value is the *x* component of the resulting vector:

```
Vector3 velocityPerUnitImpulseContact =
    contactToWorld.transformTranspose(velocityPerUnitImpulse);

real angularComponent = velocityPerUnitImpulseContact.x;
```

where the `transformTranspose` method is a convenience method that combines the effect of transforming a vector by the transpose of a matrix.[3]

---

3. It works by performing a regular matrix transformation, but it selects the components of the matrix by row rather than column order. See the code on the CD for its implementation.

Although we could implement it in this way, there is a faster way of doing it. If we have a matrix multiplication

$$\begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

then the x component of the result is $xa + yb + zc$. This is equivalent to the scalar product

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

where the vector

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

is the contact normal, as we saw when creating the basis matrix. So we can replace the full matrix transformation with code of the form

```
real angularComponent =
    velocityPerUnitImpulse * contactNormal;
```

There is another way to think of this final step. The velocityPerUnitImpulse is given in world coordinates. Performing the scalar product is equivalent to finding the component of this value in the direction of the contact normal, where the contact normal as a vector is also given in world coordinates.

It is better, in my opinion, to think in terms of the change of coordinates, because as we introduce friction in the next chapter, the simple scalar product trick can no longer be used. It is important to realize that we are going through the same process in the non-friction case: finishing with a conversion from world to contact coordinates.

Putting It Together

So for each object in the collision we can now find the change in velocity of the contact point per unit impulse.

For contacts with two objects involved we have four values: the velocity caused by linear motion and by angular motion for each object. For contacts where only one rigid body is involved (i.e., contacts with immovable fixtures such as the ground), we have just two values.

In both cases we add the resulting values together to get an overall change in velocity per unit impulse value. The whole process can be implemented in this way:

```
                        ── Excerpt from src/contacts.cpp ──
// Build a vector that shows the change in velocity in
// world space for a unit impulse in the direction of the contact
// normal.
Vector3 deltaVelWorld = relativeContactPosition[0] % contactNormal;
deltaVelWorld = inverseInertiaTensor[0].transform(deltaVelWorld);
deltaVelWorld = deltaVelWorld % relativeContactPosition[0];

// Work out the change in velocity in contact coordinates.
real deltaVelocity = deltaVelWorld * contactNormal;

// Add the linear component of velocity change.
deltaVelocity += body[0]->getInverseMass();

// Check whether we need to consider the second body's data.
if (body[1])
{
    // Find the inertia tensor for this body.
    body[1]->getInverseInertiaTensorWorld(&inverseInertiaTensor[1]);

    // Go through the same transformation sequence again.
    Vector3 deltaVelWorld = relativeContactPosition[1] % contactNormal;
    deltaVelWorld = inverseInertiaTensor[1].transform(deltaVelWorld);
    deltaVelWorld = deltaVelWorld % relativeContactPosition[1];

    // Add the change in velocity due to rotation.
    deltaVelocity += deltaVelWorld * contactNormal;

    // Add the change in velocity due to linear motion.
    deltaVelocity += body[1]->getInverseMass();
}
```

In this code the first body is considered. Its rotational component of velocity change is calculated and placed in the deltaVelocity component, followed by its linear component. If a second body is present in the contact, then the same process is repeated, and the deltaVelocity is incremented with the two components for body 2. At the end of the process deltaVelocity contains the total velocity change per unit impulse.

14.2.3 IMPULSE CHANGE BY VELOCITY

For frictionless collisions this step is incredibly simple. If we have a single value for the velocity change per unit impulse (call it *d*), then the impulse needed to achieve

a given velocity change is

$$g = \frac{v}{d} \qquad [14.3]$$

where v is the desired change in velocity and g is the impulse required.

14.2.4 Calculating the Desired Velocity Change

This stage of the algorithm has two parts. First we need to calculate the current closing velocity at the contact point. Second we need to calculate the exact change in velocity we are looking for.

Calculating the Closing Velocity

Before we can calculate the velocity change we need, we have to know what the current velocity at the contact is.

As we saw earlier, velocity has both a linear and an angular component. To calculate the total velocity of one object at the contact point we need both. We calculate its linear velocity and the linear velocity of the contact point due to rotation alone.

We can retrieve the linear velocity from an object directly; it is stored in the rigid body. To retrieve the velocity due to rotation we need to use equation 9.5 again. The total velocity of the contact point for one object is given by

```
Vector3 velocity = body->getRotation() % relativeContactPosition;
velocity += body->getVelocity();
```

If there are two bodies involved in the collision, then the second body's values can be added to the velocity vector.

This gives us a total closing velocity in world coordinates. We need the value in contact coordinates because we need to understand how much of this velocity is in the direction of the contact normal and how much is at a tangent to this. The components of the velocity that are not in the direction of the contact normal represent how fast the objects are sliding past one another: they will become important when we consider friction.

The conversion uses the basis matrix in the now familiar way:

```
contactVelocity = contactToWorld.transformTranspose(velocity);
```

For frictionless collisions we will only use the component of this vector that lies in the direction of the contact normal. Because the vector is in contact coordinates, this value is simply the x component of the vector.

Calculating the Desired Velocity Change

As I mentioned at the start of the chapter, the velocity change we are looking for is given by the same equation we used for particles:

$$v_s' = -cv_s \Rightarrow \Delta v_s = -v_s - cv_s = -(1+c)v_s$$

In other words, we need to remove all the existing closing velocity at the contact, and keep going so that the final velocity is c times its original value but in the opposite direction. In code this is simply

```
real deltaVelocity = -contactVelocity.x * (1 + restitution);
```

If the coefficient of restitution, c, is zero, then the change in velocity will be sufficient to remove all the existing closing velocity but no more. In other words, the objects will end up not separating. If the coefficient is near 1, the objects will separate at almost the same speed at which they were closing.

The value of the coefficient depends on the materials involved in the collision. Values around 0.4 look visibly very bouncy, like a rubber ball on a hard floor. Values above this can start to look unrealistic.

14.2.5 CALCULATING THE IMPULSE

With the desired velocity change in hand, the impulse is given by equation 14.3.

Because we are not concerned with friction, we are only concerned with the impulse in the direction of the contact normal. In contact coordinates the contact normal is the X axis, so the final impulse vector is

$$g_{\text{contact}} = \begin{bmatrix} g \\ 0 \\ 0 \end{bmatrix}$$

where g is the impulse, as earlier. This is implemented as

```
                    ─── Excerpt from src/contacts.cpp ───
// Calculate the required size of the impulse.
impulseContact.x = desiredDeltaVelocity / deltaVelocity;
impulseContact.y = 0;
impulseContact.z = 0;
```

At this stage it is convenient to convert out of contact coordinates into world coordinates. This makes applying the impulse in the final stage simpler. We can do this using our basis matrix to change coordinates:

$$g_{\text{world}} = M g_{\text{contact}}$$

which is implemented as

```
─────── Excerpt from src/contacts.cpp ───────
// Convert impulse to world coordinates.
Vector3 impulse = contactToWorld.transform(impulseContact);
```

With the impulse calculated in world coordinates we can go ahead and apply it to the objects in the collision.

14.2.6 APPLYING THE IMPULSE

To apply the impulse, we use equations 7.5 and 14.1. The first tells us that linear impulses change the linear velocity of the object according to the formula

$$\dot{p} = \frac{g}{m}$$

So the velocity change for the first object in the collision will be

```
Vector3 velocityChange = impulse * body[0]->getInverseMass();
```

The rotation change is given by equation 14.1 as

$$\Delta\dot{\theta} = I^{-1}u$$

We first need to calculate the impulsive torque, u, using equation 14.2 again:

$$u = q_{\text{rel}} \times g$$

In code this looks like

```
Vector3 impulsiveTorque = impulse % relativeContactPosition;
Vector3 rotationChange =
    inverseInertiaTensor.transform(impulsiveTorque);
```

These calculations work for the first object in the collision but not for the second, if there is one. To apply the impulse to the second object, we first need to make an observation. We have calculated a single value for the impulse, but there may be two objects involved in the collision.

Just as in chapter 7, both objects involved in a collision will receive the same sized impulse, but in opposite directions. And as we saw in chapter 2, changing the direction of a vector to its opposite is equivalent to changing the sign of all its components.

We have worked so far using the contact normal as it was generated by the collision detector. By convention the collision detector generates a contact normal from

the first body's point of view. So the calculated impulse will be correct for the first body. The second body should receive the impulse in the opposite direction.

We can use the same code we used for the second body, but first we need to change the sign of the impulse.

```
// Calculate velocity and rotation change for object one.
// ...

impulse *= -1;

// Calculate velocity and rotation change for object two.
// ...
```

Finally, the velocity and rotation changes calculated for each object can be directly applied to the velocity and angular velocity of the rigid body. For example:

```
body->velocity += velocityChange;
body->rotation += rotationChange;
```

14.3 Resolving Interpenetration

We have covered the procedure for representing the change in velocity when a collision happens. If the objects in our simulation were truly solid, this would be all that is needed.

Unfortunately the objects can pass into one another before we detect that a collision has occurred. The simulation proceeds in time steps, during which no checking takes place. By the end of a time step when collision detection occurs, two objects can have touched and passed into one another. We need to resolve this interpenetration in some way; otherwise objects in the game will not appear solid.

This is the same set of requirements we saw in the mass-aggregate engine. In that case, when two objects were interpenetrating, it was quite easy to move them apart. We moved each object back along the line of the contact normal to the first point where they no longer intersected.

14.3.1 Choosing a Resolution Method

For rotating rigid bodies the situation is a little more complex. There are several strategies we could employ to resolve interpenetration.

Penetration   Linear   Realistic
                  projection   (rotation and
                                   linear movement)
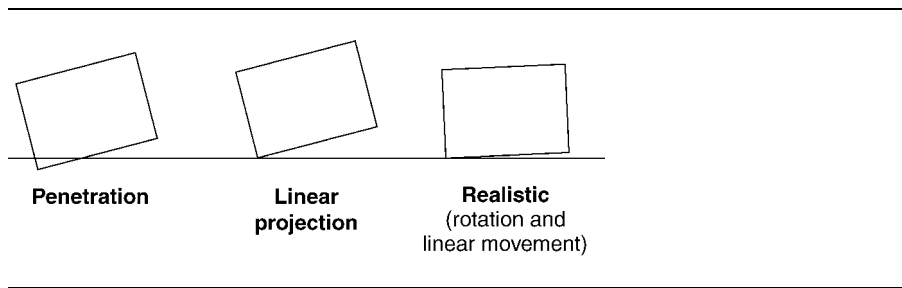
FIGURE 14.4   Linear projection causes realism problems.

Linear Projection

We could use the same algorithm as before: changing the position of each object so that it is moved apart in the direction of the contact normal. The amount of movement should be the smallest possible such that the objects no longer touch.

For collisions involving two objects the amount each one moves is proportional to its inverse mass. Therefore a light object has to take on more of the movement than a heavy object.

This approach works and is very simple to implement (in fact it uses the same code as for the mass-aggregate engine). Unfortunately it isn't very realistic. Figure 14.4 shows a block that has been knocked into the ground by another collision. If we use the linear projection interpenetration resolution method, the situation after the collision is resolved will be as shown. This is in contrast to the third part of the figure that shows how a real box would behave.

Using linear projection makes objects appear to twitch strangely. If you only have to deal with spherical objects, it is useful and very fast. For any other object we need something more sophisticated.

Velocity-Based Resolution

Another strategy used in some physics engines is to take into account the linear and angular velocities of the objects in the collision.

At some point in their motion the two objects will have just touched. After that time they will continue interpenetrating to the end of the time step. To resolve the interpenetration we could move them back to the point of first collision.

In practice, calculating this point of first collision is difficult and not worth worrying about. We can approximate it by considering only the contact point on each object as generated by the collision detector. We can move these two points back along the paths they followed until they no longer overlap in the direction of the contact normal.[4]

---

4.   This isn't the same as finding the first collision point, because it is often not the contact points generated by the collision detector that are the first to touch: in fact it can be completely different parts of the objects that touch first.
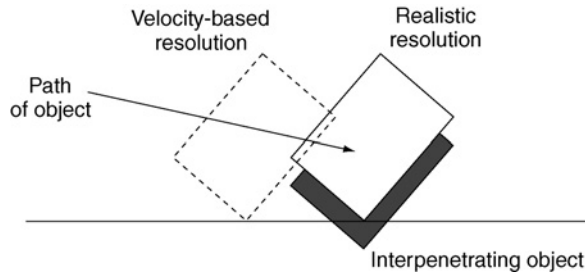
FIGURE 14.5   Velocity-based resolution introduces apparent friction.

To move the objects back we need to keep track of the velocity and rotation of each object before any collision resolution began. We can then use these values to work out an equation for the path each contact point takes, and work out when they first crossed over (i.e., when interpenetration began).

While this is a sensible strategy and can give good results, it has the effect of introducing additional friction into the simulation. Figure 14.5 shows an example of this. The object penetrates the ground while moving sideways at high speed. The velocity-based resolution method would move it back along its path as shown. To the user it would appear that the object hits the ground and sticks, even if no friction was set for the collision.

Nonlinear Projection

A third option, and the one I will employ in this chapter, is based on the linear projection method. Rather than just moving the objects back linearly, we use a combination of linear and angular movement to resolve the penetration.

The theory is the same: we move both objects in the direction of the contact normal until they are no longer interpenetrating. The movement, rather than being exclusively linear, can also have an angular component.

For each object in the collision we need to calculate the amount of linear motion and the amount of angular motion so the total effect is exactly enough to resolve the interpenetration. Just as for linear projection, the amount of motion each object makes will depend on the inverse mass of each object. Unlike linear projection, the balance between linear and angular velocity will depend on the inverse inertia tensor of each object.

An object with a high moment of inertia tensor at the contact point will be less likely to rotate, so will take more of its motion as linear motion. If the object rotates easily, however, then angular motion will take more of the burden.

Figure 14.6 shows nonlinear projection applied to the same situation we saw in figures 14.4 and 14.5. The result is still not exactly as it would be in reality, but the result is more believable and usually doesn't look odd. Figure 14.7 shows the shallow
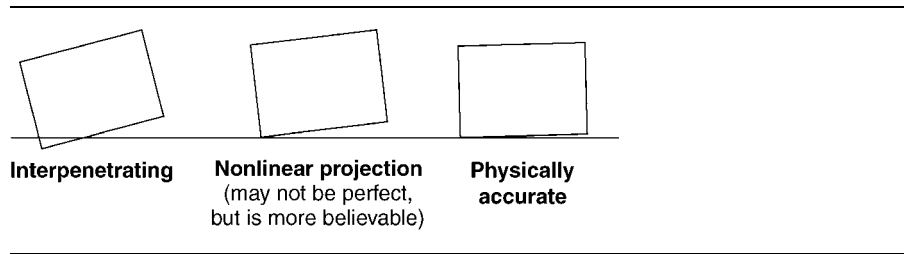
FIGURE 14.6   Nonlinear projection is more believable.
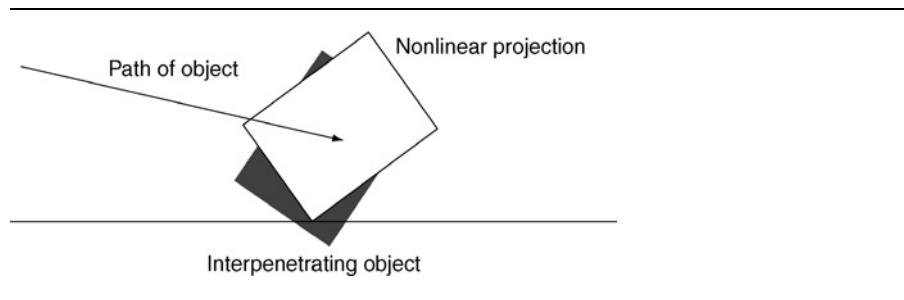
FIGURE 14.7   Nonlinear projection does not add friction.

impact situation: the nonlinear projection method doesn't introduce any additional friction. In fact it slightly diminishes the friction by allowing the object to slide farther than it otherwise would. I don't know why, but in practice this is far less noticeable than extra friction.

I will return to the details of implementing this algorithm later in the section.

Relaxation

Relaxation isn't, strictly speaking, a new resolution method. Relaxation resolves only a proportion of the interpenetration at one go, and can be used in combination with any other method. It is most commonly used with nonlinear projection, however.

Relaxation is useful when there are lots of contacts on one object. As each contact is considered and resolved, it may move the object in such a way that other objects are now interpenetrated. For a brick in a wall, any movement in any direction will cause it to interpenetrate with another brick. This can cause problems with the order in which interpenetration resolution is carried out, and can leave the simulation with contacts that still have noticeable penetration.

By performing more interpenetration resolution steps, but having each one only resolve a proportion of the interpenetration, a set of contacts can have a more equitable say over where an object ends up. Each gets to resolve a little, and then the others take their turns. This typically is repeated several times. In situations where

previously there would have been one or two contacts with obvious interpenetration, this method shares the interpenetration among all the conflicting contacts, which may be less noticeable.

Unfortunately relaxation also makes it more likely that interpenetration is noticeable at the end of an update when there are few collisions. It is beneficial to have all contacts share a small degree of interpenetration when the alternative is having one very bad contact, but in most cases it is undesirable and a full-strength resolution step is more visually pleasing.

It is relatively simple to add relaxation to your engine (you just multiply the penetration to resolve by a fixed proportion before performing the normal resolution algorithm). I'd advise you to build the basic system without relaxation, and then add it only if you find that you need to.

14.3.2   Implementing Nonlinear Projection

Let's look in more detail at the nonlinear projection method and get it working in our code. We start knowing the penetration depth of the contact: this is the total amount of movement we'll need to resolve the interpenetration. Our goal is to find the proportion of this movement that will be contributed by linear and angular motion for each object.

We first make an assumption: we imagine the objects we are simulating were pushed together so that they deformed. Rather than the amount of interpenetration, we treat the penetration depth of the contact as if it were the amount of deformation in both bodies. As we saw in the section on resolving velocities, this deformation causes a force that pushes the objects apart. This force, according to D'Alembert's principle, has both linear and angular effects. The amount of each depends on the inverse mass and the inverse inertia tensor of each object.

Treating interpenetration in this way allows us to use the same mathematics we saw in section 14.2. We are effectively modeling how the two objects would be pushed apart by the deformation forces: we are using a physically realistic method to find the linear and angular components we need.

Calculating the Components

For velocity we were interested in the amount of velocity change caused by the rotation change from a single unit of impulse. This quantity is a measure of how resistant the object is to being rotated when an impulse or force is applied at the contact point. To resolve penetration we use exactly the same sequence of equations. We find the resistance of the object to being moved in both a linear and angular way.

Recall that the resistance of an object to being moved is called its "inertia." So we are interested in finding the inertia of each object in the direction of the contact normal. This inertia will have a linear component and a rotational component.

The linear component of inertia is, as before, simply the inverse mass. The angular component is calculated using the same sequence of operations that we used previously. Together the code looks like this:

—————— **Excerpt from src/contacts.cpp** ——————

```cpp
// We need to work out the inertia of each object in the direction
// of the contact normal, due to angular inertia only.
for (unsigned i = 0; i < 2; i++) {
    if (body[i]) {
        Matrix3 inverseInertiaTensor;
        body[i]->getInverseInertiaTensorWorld(&inverseInertiaTensor);

        // Use the same procedure as for calculating frictionless
        // velocity change to work out the angular inertia.
        Vector3 angularInertiaWorld =
            relativeContactPosition[i] % contactNormal;
        angularInertiaWorld =
            inverseInertiaTensor.transform(angularInertiaWorld);
        angularInertiaWorld =
            angularInertiaWorld % relativeContactPosition[i];
        angularInertia[i] = angularInertiaWorld * contactNormal;

        // The linear component is simply the inverse mass.
        linearInertia[i] = body[i]->getInverseMass();

        // Keep track of the total inertia from all components.
        totalInertia += linearInertia[i] + angularInertia[i];
    }
}
```

At the end of this loop we have the four values (two for single-body collisions) that tell us the proportion of the penetration to be resolved by each component of each rigid body. The actual amount each object needs to move is found by

```cpp
real inverseInertia = 1 / totalInertia;
linearMove[0] = penetration * linearInertia[0] * inverseInertia;
linearMove[1] = -penetration * linearInertia[1] * inverseInertia;
angularMove[0] = penetration * angularInertia[0] * inverseInertia;
angularMove[1] = -penetration * angularInertia[1] * inverseInertia;
```

The penetration value is negative for the second object in the collision for the same reason we changed the sign of the impulse for velocity resolution: the movement is given from the first object's point of view.

Applying the Movement

Applying the linear motion is simple. The linear move value gives the amount of motion required, and the contact normal tells us the direction in which the movement should take place:

```
body[i]->position += contactNormal * linearMove[i];
```

The angular motion is a little more difficult. We know the amount of linear move-ment we are looking for; we need to calculate the change in the orientation quaternion that will give it to us.

We do this in three stages. First we calculate the rotation needed to move the con-tact point by one unit. Second, we multiply this by the number of units needed (i.e., the `angularMove` value). Finally we apply the rotation to the orientation quaternion.

We can calculate the direction in which the object needs to rotate, using the same assumption that the rotation is caused by some kind of impulse (even though velocity does not change, only position and orientation). If an impulse were exerted at the contact point, the change in rotation would be

$$\Delta \dot{\boldsymbol{\theta}} = I^{-1} \boldsymbol{u} = I^{-1}(q_{\text{rel}} \times \boldsymbol{g})$$

where q_{rel} is the relative position of the contact point, $\boldsymbol{u}$ is the impulsive torque gen-erated by the impulse, and $\boldsymbol{g}$ is the impulse in the direction of the contact normal, as before. In code, this is

```
Vector3 inverseInertiaTensor;
body->getInverseInertiaTensorWorld(&inverseInertiaTensor);

Vector3 impulsiveTorque = relativeContactPosition % contactNormal;
Vector3 impulsePerMove =
    inverseInertiaTensor.transform(impulsiveTorque);
```

This tells us the impulsive torque needed to get one unit of motion. We are not really interested in impulses, however. (Because we already know the total distance that needs to be moved, and we can directly change the object, we don't need to worry about how forces get translated into motion.)

To find the rotation needed to get one unit of movement we simply multiply through by the inertia:

```
Vector3 rotationPerMove = impulsePerMove * 1/angularInertia;
```

The `rotationPerMove` vector now tells us the rotation we need to get one unit of move-ment. And we know the total movement we want is `angularMove`, so we know the total rotation to apply is

```
Vector3 rotation = rotationPerMove * angularMove;
```

To apply this rotation we use equation 9.8, via the quaternion function `updateByVector` that we defined earlier.

14.3.3 AVOIDING EXCESSIVE ROTATION

There are two issues to address with the algorithm presented so far. The first is an assumption that slipped in without being commented on, and the second is a potential problem that can cause instability and odd-looking behavior. Both are related to objects being rotated too much as they are moved out of penetration.

Figure 14.8 shows an object that has been severely interpenetrated. If the moment of inertia of the object is small but its mass is large, then most of the extraction will be down to angular movement. Clearly, no matter how much angular movement is imposed, the contact point will never get out of the object. The example is extreme, of course, but the problem is very real.

The instant an object begins rotating from an impulsive torque, the contact point will also begin to move. We have assumed that we can take the instantaneous change in position of the contact point (i.e., its velocity) and use that to work out how much rotation is needed. Making this assumption means that there will always be a solution for how much rotation to apply, even in cases where no solution really exists (such as in figure 14.8).

In effect we have assumed that the contact point would continue to move in its initial direction forever at the same rate. Clearly this is a wrong assumption: the contact point would change its direction of motion as it rotates around the center of mass. For small rotations the assumption is quite good. And we hope that most interpenetrations aren't too large.

For large rotations we have another problem, however. We have the possibility that we might rotate the object so far that the contact point will start to get closer

No rotation
can resolve
interpenetration
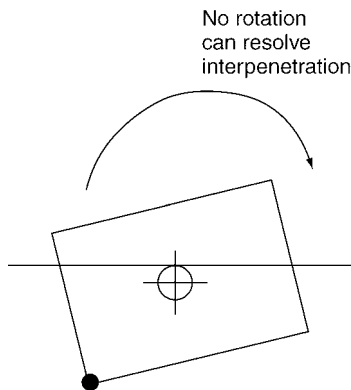
FIGURE 14.8 Angular motion cannot resolve the interpenetration.

Rotating will cause
the other corner
to interpenetrate
before the contact
is resolved
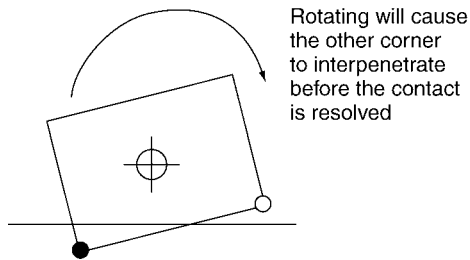
FIGURE 14.9   Angular resolution causes other problems.

again, or that another part of the object will come into penetration. Figure 14.9 shows this case. Even a modest rotation of the object can cause another penetration to occur.

For both issues we need to limit the amount of rotation that can be part of our penetration resolution. Keeping this value small means that our small-rotation assumption is valid, and that we minimize the chance of causing other interpenetrations while resolving one.

The amount of linear and angular motion we want is calculated and stored in four variables (two for single-body collisions):

```
linearMove[0]
linearMove[1]
angularMove[0]
angularMove[1]
```

We can simply check that the values of `angularMove` are not too great. If they are, we can transfer some of the burden from them onto the corresponding `linearMove` component.

But what is "too great"? This is where we descend into the black art of tuning the physics engine. I haven't come across a sensible logical argument for choosing any particular strategy.

Some developers use a fixed amount: not allowing the angular move value to be greater than 0.5, for example. This works well as long as the objects in the simulation are all roughly the same size. If some objects are very large, then a suitable limit for them may be unsuitable for smaller objects, and vice versa.

It is also possible to express the limit in terms of a fraction of a revolution that the object can make. We could limit the rotation so that the object never turns through more than 45°, for example. This accounts for differences in size, but it is more complex to work out the equivalent angular move for a specific angle of rotation.

A simple alternative is to scale the angular move by the size of the object (where the size of the object can be approximated by the magnitude of the relative contact

position vector). Therefore larger objects can have more angular movement. This is the approach I have used in the code on the CD.

```
real limit = angularLimitConstant *
                relativeContactPosition.magnitude();

// Check the angular move is within limits.
if (real_abs(angularMove) > limit)
{
    real totalMove = linearMove + angularMove;

    // Set the new angular move, with the same sign as before.
    if (angularMove >= 0) {
        angularMove = limit;
    } else {
        angularMove = -limit;
    }

    // Make the linear move take the extra slack.
    linearMove = totalMove - angularMove;
}
```

The value for angularLimitConstant needs to be determined by playing with your particular simulation. I have found that values around 0.2 give good results, although lower values are better when very bouncy collisions are needed.

14.4 The Collision Resolution Process

So far we have looked at resolving particular collisions for both velocity and interpenetration. Handling one collision on its own isn't very useful.

The collision detector generates any number of contacts, and all these need to be processed. We need to build a framework in which any number of collisions can be processed at once. This final section of the chapter ties the previous algorithms together to that end. We will end up with a complete collision resolution system that can be used for simulations that don't need friction. In the next two chapters we will extend the engine to handle friction, to improve speed, and to increase stability for objects resting on one another.

I mentioned in the introduction to the book that the choice of how to resolve a series of collisions is at the heart of how a physics system is engineered. Most of the commercial physics middleware packages process all the collisions at the same time (or at least batch them into groups to be processed simultaneously). This allows them to make sure that the adjustments made to one contact don't disturb others.

We will steer a slightly different course. Our resolution system will look at each collision in turn, and correct it. It will process collisions in order of severity (fast collisions are handled first). It may be that resolving one collision in this way will cause others to be made worse. We will have to structure the code so that it can take account of this problem.

In chapter 18 I will look at the simultaneous resolution approaches. There is a good chance that your physics needs will not require their sophistication, however. While they are more stable and accurate than the methods in this part of the book, they are very much more complex and can be considerably slower.

14.4.1 The Collision Resolution Pipeline

Figure 14.10 shows a schematic of the collision resolution process. Collisions are generated by the collision detector based on the collision geometry of the objects involved. These collisions are passed into a collision resolution routine, along with the rigid-body data for the objects involved.

The collision resolution routine has two components: a velocity resolution system and a penetration resolution system. These correspond to the two algorithms that have made up the majority of this chapter.

These two steps are independent of each other. Changing the velocity of the objects doesn't affect how deeply they are interpenetrating, and vice versa.[5] Physics en-
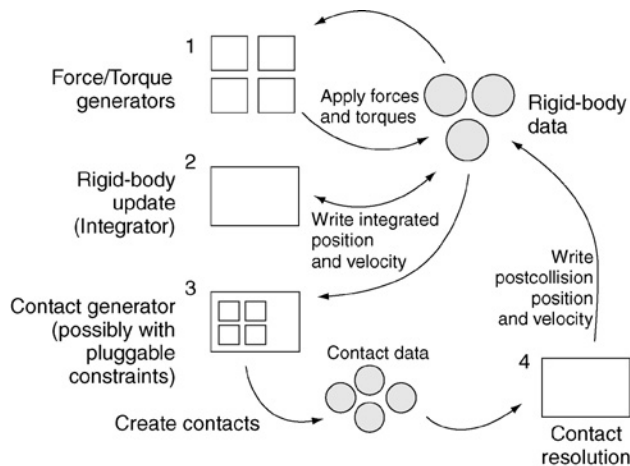
FIGURE 14.10   Data flow through the physics engine.

---

5. Actually this is not strictly true: changing the position of objects can change the relative position of their contacts, which can affect the velocity calculations we've used in our algorithm. The effect is usually tiny, however, and for practical purposes we can ignore the interdependence.

gines that do very sophisticated velocity resolution, with all collisions handled at the same time, often have a separate penetration resolver that uses the algorithms we implemented earlier.

The collision resolver we implement in this chapter is set in a class: `CollisionResolver`. It has a method, `resolveContacts`, that takes the whole set of collisions and the duration of the frame, and it performs the resolution in three steps: first it calculates internal data for each contact; then it passes the contacts to the penetration resolver; and then they go to the velocity resolver:

Excerpt from include/cyclone/contacts.h

```
/**
 * The contact resolution routine. One resolver instance
 * can be shared for the whole simulation, as long as you need
 * roughly the same parameters each time (which is normal).
 */
class ContactResolver
{
public:
    /**
     * Resolves a set of contacts for both penetration and velocity.
    void resolveContacts(Contact *contactArray,
        unsigned numContacts,
        real duration);
};
```

Excerpt from src/contacts.cpp

```
#include <cyclone/contacts.h>
void ContactResolver::resolveContacts(Contact *contacts,
                                      unsigned numContacts,
                                      real duration)
{
    // Make sure we have something to do.
    if (numContacts == 0) return;

    // Prepare the contacts for processing
    prepareContacts(contacts, numContacts, duration);

    // Resolve the interpenetration problems with the contacts.
    adjustPositions(contacts, numContacts, duration);

    // Resolve the velocity problems with the contacts.
    adjustVelocities(contacts, numContacts, duration);
}
```

We also add a friend declaration to the contact data structure to allow the resolver to have direct access to its internals:

```
                    ———— Excerpt from include/cyclone/contacts.h ————
    class Contact
    {
        // ... Other data as before ...

        /**
         * The contact resolver object needs access into the contacts to
         * set and effect the contact.
         */
        friend ContactResolver;
    };
```

14.4.2 Preparing Contact Data

Because we may be performing a penetration resolution step as well as a velocity resolution step for each contact, it is useful to calculate information that both steps need in one central location. In addition, extra information required to work out the correct order of resolution must be calculated.

In the first category are two bits of data:

- The basis matrix for the contact point, calculated in the `calculateContact-Basis` method, is called `contactToWorld`.

- The position of the contact is relative to each object. I called this `relative-ContactPosition` in the previous code.

The relative velocity at the contact point falls into the second category. We need this to resolve velocity, so we can just calculate it in the appropriate method. If we're going to resolve collisions in order of severity (the fastest first), we'll need this value to determine which collision to consider first. So it benefits from being calculated once and reused when needed.

We can store these data in the `Contact` data structure:

```
                    ———— Excerpt from include/cyclone/contacts.h ————
    class Contact
    {
        // ... Other data as before ...

    protected:

        /**
         * A transform matrix that converts coordinates in the contact's
         * frame of reference to world coordinates. The columns of this
         * matrix form an orthonormal set of vectors.
```

```
         */
        Matrix3 contactToWorld;

        /**
         * Holds the closing velocity at the point of contact. This is
         * set when the calculateInternals function is run.
         */
        Vector3 contactVelocity;

        /**
         * Holds the required change in velocity for this contact to be
         * resolved.
         */
        real desiredDeltaVelocity;

        /**
         * Holds the world space position of the contact point
         * relative to the center of each body. This is set when
         * the calculateInternals function is run.
         */
        Vector3 relativeContactPosition[2];
    };
```

The preparation routine only needs to call each contact in turn and ask it to calculate the appropriate data:

—————— **Excerpt from include/cyclone/contacts.h** ——————

```
/**
 * The contact resolution routine. One resolver instance
 * can be shared for the whole simulation, as long as you need
 * roughly the same parameters each time (which is normal).
 */
class ContactResolver
{
protected:
    /**
     * Sets up contacts ready for processing. This makes sure their
     * internal data is configured correctly and the correct set of
     * bodies is made alive.
     */
    void prepareContacts(Contact *contactArray, unsigned numContacts,
        real duration);
};
```

```
                   ── Excerpt from include/cyclone/contacts.h ──
#include <cyclone/contacts.h>
void ContactResolver::prepareContacts(Contact* contacts,
                                      unsigned numContacts,
                                      real duration)
{
    // Generate contact velocity and axis information.
    Contact* lastContact = contacts + numContacts;
    for(Contact* contact=contacts; contact < lastContact; contact++)
    {
        // Calculate the internal contact data (inertia, basis, etc).
        contact->calculateInternals(duration);
    }
}
```

In the calculateInternals method of the contact we need to calculate each of the three bits of data: the contact basis, the relative position, and the relative velocity:

```
                        ── Excerpt from src/contacts.cpp ──
void Contact::calculateInternals(real duration)
{
    // Check if the first object is NULL, and swap if it is.
    if (!body[0]) swapBodies();
    assert(body[0]);

    // Calculate a set of axes at the contact point.
    calculateContactBasis();

    // Store the relative position of the contact relative to each body.
    relativeContactPosition[0] = contactPoint - body[0]->getPosition();
    if (body[1]) {
        relativeContactPosition[1] =
            contactPoint - body[1]->getPosition();
    }

    // Find the relative velocity of the bodies at the contact point.
    contactVelocity = calculateLocalVelocity(0, duration);
    if (body[1]) {
        contactVelocity -= calculateLocalVelocity(1, duration);
    }

    // Calculate the desired change in velocity for resolution.
    calculateDesiredDeltaVelocity(duration);
}
```

The contact basis method was described in section 14.2.1. The relative position calculation should be rather straightforward. The remaining two components—swapping bodies and calculating relative velocity—deserve some comment.

Swapping Bodies

The first two lines make sure that, if there is only one object in the collision, it is in the zero position of the array. So far we have assumed that this is true. If your collision detector is guaranteed to only return single-object collisions in this way, then you can ignore this code.

To swap the bodies we need to move the two body references and also reverse the direction of the contact normal. The contact normal is always given from the first object's point of view. If the bodies are swapped, then this needs to be flipped:

```
                    ── Excerpt from src/contacts.cpp ──
/**
 * Swaps the bodies in the current contact, so body 0 is at body 1 and
 * vice versa. This also changes the direction of the contact normal, but
 * doesn't update any calculated internal data. If you are calling this
 * method manually, then call calculateInternals afterward to make sure
 * the internal data is up to date.
 */
void Contact::swapBodies()
{
    contactNormal *= -1;

    RigidBody *temp = body[0];
    body[0] = body[1];
    body[1] = temp;
}
```

Calculating Relative Velocity

The relative velocity we are interested in is the total closing velocity of both objects at the contact point. This will be used to work out the desired final velocity after the objects bounce.

The velocity needs to be given in contact coordinates. Its *x* value will give the velocity in the direction of the contact normal, and its *y* and *z* values will give the amount of sliding that is taking place at the contact. We'll use these two values in the next chapter when we meet friction.

Velocity at a point, as we have seen, is made up of both linear and angular components:

$$\dot{q}_{\text{rel}} = \dot{\theta} \times q_{\text{rel}} + \dot{p}$$

where $\boldsymbol{q}_{\mathrm{rel}}$ is the position of the point we are interested in, relative to the object's center of mass; $\boldsymbol{p}$ is the position of the object's center of mass (i.e., $\dot{\boldsymbol{p}}$ is the linear velocity of the whole object); and $\dot{\boldsymbol{\theta}}$ is the object's angular velocity.

To calculate the velocity in contact coordinates we use this equation and then transform the result by the transpose of the contact basis matrix:

```
────────────── Excerpt from src/contacts.cpp ──────────────
Vector3 Contact::calculateLocalVelocity(unsigned bodyIndex, real duration)
{
    RigidBody *thisBody = body[bodyIndex];

    // Work out the velocity of the contact point.
    Vector3 velocity =
        thisBody->getRotation() % relativeContactPosition[bodyIndex];
    velocity += thisBody->getVelocity();

    // Turn the velocity into contact coordinates
    Vector3 contactVelocity = contactToWorld.transformTranspose(velocity);

    // And return it.
    return contactVelocity;
}
```

The `calculateInternals` method finds the overall closing velocity at the contact point, by subtracting the second body's closing velocity from the first:

```
// Find the relative velocity of the bodies at the contact point.
contactVelocity = calculateLocalVelocity(0, duration);
if (body[1]) {
    contactVelocity -= calculateLocalVelocity(1, duration);
}
```

Because this algorithm uses both the contact basis matrix and the relative contact positions, it must be done last.

14.4.3 RESOLVING PENETRATION

We have visited each contact and calculated the data we'll need for both resolution steps. We now turn our attention to resolving the interpenetration for all contacts. We will do this by taking each contact in turn and calling a method (`applyPosition-Change`) that contains the algorithm in section 14.3 for resolving a single contact.

We could simply do this in the same way as for `prepareContacts`:
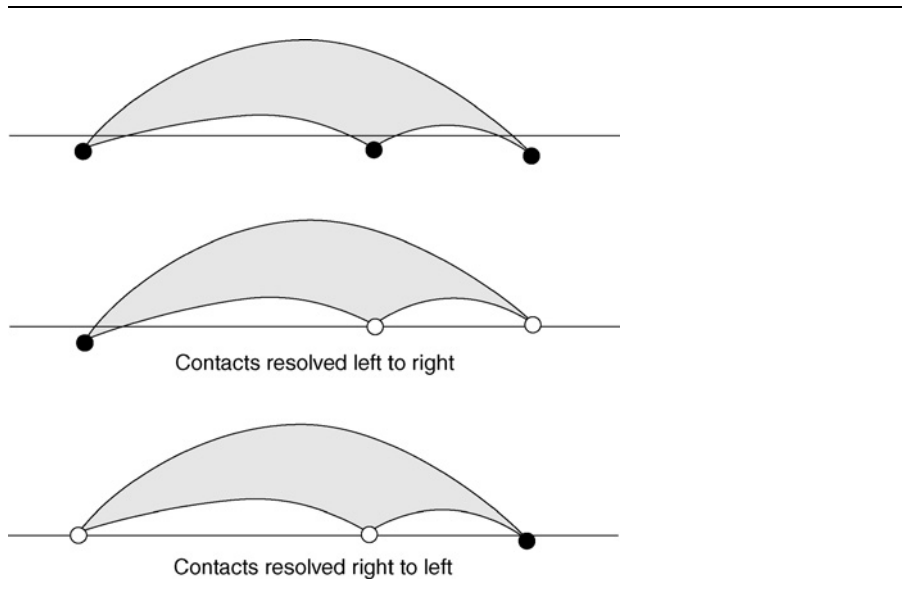
FIGURE 14.11 Resolution order is significant.

```
Contact* lastContact = contacts + numContacts;
for(Contact* contact=contacts; contact < lastContact; contact++)
{
    contact->applyPositionChange();
}
```

This will work but isn't optimal.

Figure 14.11 shows three interpenetrating contacts in a row. The middle part of the figure shows what happens when the contacts are resolved in order: a large interpenetration remains. The final part of the figure shows the same set of contacts after resolving in reverse order. There is still some interpenetration visible, but it is drastically reduced.

Rather than go through the contacts in order and resolve their interpenetration, we can resolve the collisions in penetration order. At each iteration we search through to find the collision with the deepest penetration value. This is handled through its applyPositionChange method in the normal way. The process is then repeated up to some maximum number of iterations (or until there are no more interpenetrations to resolve, whichever comes first).

This algorithm can revisit the same contacts several times. Figure 14.12 shows a box resting on a flat plane. Each corner is penetrating the surface. Moving the first corner up will cause the second to descend farther. Moving the second will cause
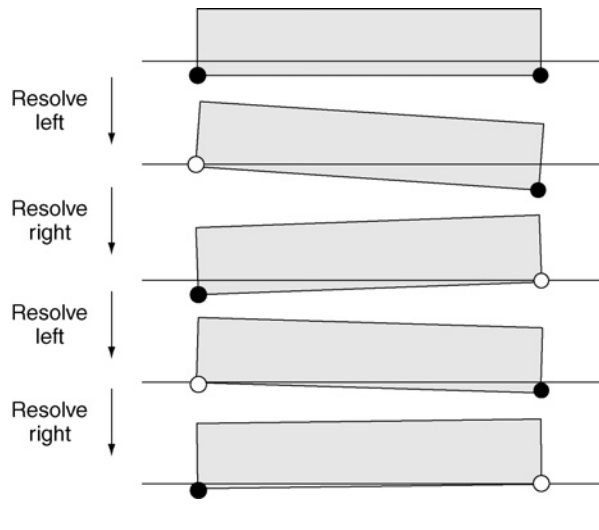
FIGURE 14.12   Repeating the same pair of resolutions.

the first to penetrate again, and so on. Given enough iterations, this situation will be resolved so neither corner is penetrating. It is more likely that the iteration limit will be reached, however. If you check the number of iterations actually used, you will find this kind of situation is common and will consume all available iterations.

The same issue can also mean that a contact with a small penetration never gets resolved: the resolution algorithm runs out of iterations before considering the contact. To avoid this situation, and to guarantee that all contacts get considered, we can run a single pass through all the contacts and then move on to the best-first iterative algorithm. In practice, however, this is rarely necessary, and a best-first resolution system works well on its own. Problems may arise, though, for fast-moving, tightly packed objects; for simulations with longer time steps; or when there are very small limits on the number of iterations.

Typically objects that gradually sink into surfaces and then suddenly jump out a short way are symptomatic of penetration resolution not getting to shallow contacts (i.e., the contacts are ignored until they get too deep, whereupon they are suddenly resolved). If this happens, you can add a pass through all contacts before the iterative algorithm.

Iterative Algorithm Implemented

To find the contact with the greatest penetration we can simply look through each contact in the list. The contact found can then be resolved:

```
Contact* lastContact = contacts + numContacts;
for (unsigned i = 0; i < positionIterations; i++)
{
    Contact* worstContact = NULL;
    real worstPenetration = 0;
    for(Contact* contact = contacts; contact < lastContact; contact++)
    {
        if (contact->penetration > worstPenetration)
        {
            worstContact = contact;
            worstPenetration = contact->penetration;
        }
    }
    if (worstContact) worstContact->applyPositionChange();
    else break;
}
```

This method looks through the whole list of contacts at each iteration. If this were all we needed, then we could do better by sorting the list of contacts first, and then simply work through them in turn.

Unfortunately the preceding algorithm doesn't take into account the fact that one adjustment may change the penetration of other contacts. The penetration data member of the contact is set during collision detection. Movement of the objects during resolution can change the penetration depth of other contacts, as we saw in figures 14.9 and 14.12.

To take this into account we need to add an update to the end of the algorithm:

```
for (unsigned i = 0; i < positionIterations; i++)
{
    // Find worstContact (as before) ...

    if (!worstContact) break;

    worstContact->applyPositionChange();

    updatePenetrations();
}
```

where updatePenetrations recalculates the penetrations for each contact. To implement this method accurately we'd need to go back to the collision detector and work out all the contacts again. Moving an object out of penetration may cause another
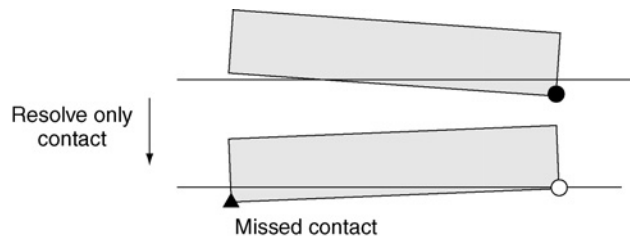
FIGURE 14.13   Resolving penetration can cause unexpected contact changes.

contact to disappear altogether, or bring new contacts that weren't expected before. Figure 14.3 shows this in action.

Unfortunately collision detection is far too complex to be run for each iteration of the resolution algorithm. We need a faster way.

Updating Penetration Depths

Fortunately there is an approximation we can use that gives good results. When the penetration for a collision is resolved, only one or two objects can be moved: the one or two objects involved in the collision. At the point where we move these objects (in the `applyPositionChange` method), we know how much they are moving both linearly and angularly.

After resolving the penetration, we keep track of the linear and angular motion we applied to each object. Then we check through all contacts and find those that also apply to either object. Only these contacts are updated based on the stored linear and angular movements.

The update for one contact involves the assumption we've used several times in this chapter: that the only point involved in the contact is the point designated as the contact point. To calculate the new penetration value we calculate the new position of the relative contact point for each object, based on the linear and angular movements we applied. The penetration value is adjusted based on the new position of these two points: if they have moved apart (along the line of the contact normal), then the penetration will be less; if they have overlapped, then the penetration will be increased.

If the first object in a contact has changed, then the update of the position will be

```
cp = rotationChange[0].vectorProduct(
  c[i].relativeContactPosition[0])
  ;

cp += velocityChange[0];
```

```
c[i].penetration -= rotationAmount[0]*cp.scalarProduct(
  c[i].contactNormal
  );
```

If the second object has changed, the code is similar, but the value is added at the end:

```
cp = rotationChange[1].vectorProduct(
  c[i].relativeContactPosition[1]);

cp += velocityChange[1];

c[i].penetration += rotationAmount[1]*cp.scalarProduct(
  c[i].contactNormal
  );
```

Finally we need some mechanism for storing the adjustments made in the applyPositionChange method for use in this update. The easiest method is to add data members to the ContactResolver class.

The complete code puts these stages together: finding the worst penetration, resolving it, and then updating the remaining contacts. The full code looks like this:

```
                      ── Excerpt from src/contacts.cpp ──
void ContactResolver::adjustPositions(Contact *c,
                                      unsigned numContacts,
                                      real duration)
{
    unsigned i,index;
    Vector3 velocityChange[2], rotationChange[2];
    real rotationAmount[2];
    real max;
    Vector3 cp;

    // Iteratively resolve interpenetration in order of severity.
    positionIterationsUsed = 0;
    while(positionIterationsUsed < positionIterations)
    {
        // Find biggest penetration.
        max = positionEpsilon;
        index = numContacts;
        for(i=0;i<numContacts;i++) {
            if(c[i].penetration > max)
            {
                max=c[i].penetration;
```

```
                    index=i;
                }
        }
        if (index == numContacts) break;

        // Match the awake state at the contact.
        //c[index].matchAwakeState();

        // Resolve the penetration.
        c[index].applyPositionChange(velocityChange,
            rotationChange,
            rotationAmount,
            max);//-positionEpsilon);

        // Again this action may have changed the penetration of other
        // bodies, so we update contacts.
        for(i=0; i<numContacts; i++)
        {
            if(c[i].body[0])
            {
                if(c[i].body[0]==c[index].body[0])
                {
                    cp = rotationChange[0].vectorProduct(c[i].
                        relativeContactPosition[0]);

                    cp += velocityChange[0];

                    c[i].penetration -=
                        rotationAmount[0]*cp.scalarProduct(c[i].
                        contactNormal);
                }
                else if(c[i].body[0]==c[index].body[1])
                {
                    cp = rotationChange[1].vectorProduct(c[i].
                        relativeContactPosition[0]);

                    cp += velocityChange[1];

                    c[i].penetration -=
                        rotationAmount[1]*cp.scalarProduct(c[i].
                        contactNormal);
                }
            }
            if(c[i].body[1])
```

```
            {
                if(c[i].body[1]==c[index].body[0])
                {
                    cp = rotationChange[0].vectorProduct(c[i].
                        relativeContactPosition[1]);

                    cp += velocityChange[0];

                    c[i].penetration +=
                        rotationAmount[0]*cp.scalarProduct(c[i].
                        contactNormal);
                }
                else if(c[i].body[1]==c[index].body[1])
                {
                    cp = rotationChange[1].vectorProduct(c[i].
                        relativeContactPosition[1]);

                    cp += velocityChange[1];

                    c[i].penetration +=
                        rotationAmount[1]*cp.scalarProduct(c[i].
                        contactNormal);
                }
            }
        }
        positionIterationsUsed++;
    }
}
```

14.4.4 Resolving Velocity

With penetration resolved we can turn our attention to velocity. This is where different physics engines vary the most, with several different but excellent strategies for resolving velocity. We'll return to some of them in chapter 18.

For this chapter I have aimed for the simplest end of the spectrum: a velocity resolution system that works and is stable, is as fast as possible, but avoids the complexity of simultaneously resolving multiple collisions. The algorithm is almost identical to the one for penetration resolution.

The algorithm runs in iterations. At each iteration it finds the collision with the greatest closing velocity. If there is no collision with a closing velocity, then the algorithm can terminate. If there is a collision, then it is resolved in isolation, using the method we saw in the first three sections of the chapter. Other contacts are then

updated based on the changes that were made. If there are more velocity iterations available, then the algorithm repeats.

Updating Velocities

The largest change from the penetration version of this algorithm lies in the equations for updating velocities. As before we search through to find only those contacts with an object that has just been altered.

If the first object in the contact has changed, the update of the velocity looks like this:

```
cp = rotationChange[0].vectorProduct(
  c[i].relativeContactPosition[0]
  );

cp += velocityChange[0];

c[i].contactVelocity += c[i].contactToWorld.transformTranspose(cp);
c[i].calculateDesiredDeltaVelocity(duration);
```

The corresponding code for the second object looks like this:

```
cp = rotationChange[0].vectorProduct(
  c[i].relativeContactPosition[1]
  );

cp += velocityChange[0];

c[i].contactVelocity -= c[i].contactToWorld.transformTranspose(cp);
c[i].calculateDesiredDeltaVelocity(duration);
```

The `calculateDesiredDeltaVelocity` function is implemented as

```
                    ──── Excerpt from src/contacts.cpp ────
// Calculate the acceleration-induced velocity accumulated this frame.
real velocityFromAcc =
    body[0]->getLastFrameAcceleration() * duration * contactNormal;

if (body[1])
{
    velocityFromAcc -=
        body[1]->getLastFrameAcceleration() * duration * contactNormal;
}
```

```
// If the velocity is very slow, limit the restitution.
real thisRestitution = restitution;
if (real_abs(contactVelocity.x) < velocityLimit)
{
    thisRestitution = (real)0.0f;
}


// Combine the bounce velocity with the removed
// acceleration velocity.
desiredDeltaVelocity =
    -contactVelocity.x
    -thisRestitution * (contactVelocity.x - velocityFromAcc);
```

Once again, both cases must be able to take their adjustment from either the first or second object of the contact that has been adjusted.

The complete code listing is very similar to that shown for penetration, so I won't include it here. You can find it in the `src/contacts.cpp` file on the CD.

14.4.5   ALTERNATIVE UPDATE ALGORITHMS

I must confess I have a natural distaste for algorithms that repeatedly loop over arrays finding maxima, or that search through an array finding matching objects to adjust. Over years of programming I've learned to suspect that there is probably a much better way. Both these red flags crop up in the penetration and velocity resolution algorithms.

I spent a good deal of time preparing this book, implementing alternatives and variations that would improve the theoretical speed of the algorithm. One such alternative that provides good performance is to keep a sorted list of the contacts. By way of illustration I'll describe it here.

The list of contacts is built as a doubly linked list, by adding two pointers in the contact data structure: pointing to the next and previous contacts in the list.

Taking the penetration resolution algorithm as an example (although exactly the same thing happens for velocity resolution), we initially sort all the contacts into the doubly linked list in order of decreasing penetration.

At each iteration of the algorithm, the first contact in the list is chosen and resolved (it will have the greatest penetration). Now we need to update the penetrations of contacts that might have been affected. To do this I use another pair of linked lists in the contact data structure. These linked lists contain all the contacts that involve one particular object. There must be two such lists because each contact has up to two objects involved. To hold the start of these lists, I add a pointer in the `RigidBody` class.

This means that, if we know which rigid bodies were adjusted, we can simply walk through their list of contacts to perform the update. In (highly abbreviated) code it looks something like this:

```
class Contact
{
    // Holds the doubly linked list pointers for the ordered list.
    Contact * nextInOrder;
    Contact * previousInOrder;

    // Holds pointers to the next contact that involves each rigid body.
    Contact * nextObject[2];

    // ... Other data as before ...
}

class RigidBody.
{
    // Holds the list of contacts that involve this body.
    Contact * contacts;

    // ... Other data as before ...
}
```

At this point we have a set of contacts whose penetration values have changed. One contact has changed because it has been resolved, and possibly a whole set of other contacts have been changed as a consequence of that resolution. All of these may now be in the wrong position in the ordered list. The final stage of this algorithm is to adjust their positions.

The easiest way to do this is to extract them from the main ordered list. Sort them as a new sublist, and then walk through the main list, inserting them in order at the correct point. In abbreviated code, this looks like:

```
Contact *adjustedList;
Contact *orderedList;

orderedList = sort(contacts);

for (unsigned i = 0; i < positionIterations; i++)
{
    // Make sure the worst contact is bad.
    if (orderedList->penetration < 0) break;

    // Adjust its position.
    orderedList->applyPositionChange();
```

```
        // Move it to the adjusted list.
        moveToAdjusted(orderedList);

        // Loop through the contacts for the first body.
        Contact *bodyContact = orderedList->body[0].contacts;
        while (bodyContact)
        {
            // Update the contact.
            bodyContact->updatePenetration(positionChange, orientationChange);

            // Schedule it for adjustment.
            moveToAdjusted(bodyContact);

            // Find out which linked list to move along on, then follow
            // it to get the next contact for this body.
            unsigned index = 0;
            if (bodyContact->body[0] != orderedList->body[0]) index = 1;
            bodyContact = bodyContact->nextObject[index];
        }

        if (orderedList->body[1])
        {
            // Do the same thing for the second body
            // (omitted for brevity).
        }

        // Now sort the adjusted set
        sortInPlace(adjustedList);

        // And insert them at the correct place.
        Contact *orderedListEntry = orderedList;
        while (orderedListEntry)
        {
            if (adjustedList->penetration > orderedListEntry->penetration)
            {
                Contact *contactToInsert = adjustedList;
                adjustedList = adjustedList->nextInOrder;
                insertIntoList(contactToInsert, orderedListEntry);
            }
        }
}
```

I've assumed that standard sorting and list manipulation routines are available, along with some extra methods I've used to hide the actual updates for the sake of brevity (we saw the code for these earlier).

Performance

There are tens of variations of this kind of ordering system and lots of different ways to sort, keep lists, and perform updates. I implemented six different methods while experimenting for this book.

The best performance gain was achieved using the method just described. Unfortunately it was very minor. For frames with few contacts, and using some of the more general optimization techniques described in chapter 16, the performance of the linked list version is considerably worse than the naïve approach. With many tens of contacts, among a set of tightly packed objects,[6] it becomes more efficient. For several hundred contacts among tightly packed objects, it becomes significantly faster.

For the simulations I come across in the games I'm involved with, it simply isn't worth the extra development effort. I'd rather not have the extra pointers hanging around in the contact and rigid-body data structures. You may come across situations where the scale of the physics you are working with makes it essential. For anything in game development it is essential to profile your code before trying to optimize it.

14.5 SUMMARY

Collision resolution involves some of the most complex mathematics we've met so far. For a single contact we do it in two steps: resolving the interpenetration between objects and turning their closing velocity into rebounding velocity.

The velocity resolution algorithm involves working out the effect of applying an impulse to the contact point. This can then be used to work out the impulse that will generate the desired effect. The result is a single impulse value that modifies both the linear and angular velocity of each object involved.

Unlike the velocity resolution algorithm, penetration resolution does not correspond to a physical process (since rigid objects cannot interpenetrate in reality). Because of this, there are lots of different approaches to get visibly believable behavior. In this chapter we implemented an approach derived from the same compression and impulse mathematics used for velocity resolution.

Resolving one contact alone isn't much use. To resolve the complete set of contacts, we used two similar algorithms: one to resolve all penetrations and the second to resolve all velocities. Each algorithm considered collisions in order of their severity (i.e., penetration depth or closing velocity). The worst collision was resolved in isola-

---

6. The significance of tightly packed objects will become apparent in chapter 16: if objects are not tightly packed, then it is possible to consider contacts in smaller batches, which is much more efficient.

tion, and then other collisions that would be affected were updated. Each algorithm continued up to a fixed maximum number of iterations.

The resulting physics system is quite usable, and if you are following along writing your own code, I recommend that you have a go at creating a demonstration program and see the results. The simulation has no friction, so objects slide across one another. For simple sets of objects it is likely to work fine. For more complex scenarios you may notice problems with vibrating objects or slow performance.

We will address these three limitations in the next two chapters. Chapter 15 looks at the difference between the collisions we have been dealing with so far and with resting contacts (this is part of the vibration problem). It also introduces friction. Once friction is introduced, more stability problems become visible. Chapter 16 addresses vibration and friction stability, and looks at some simple techniques for dramatically improving the performance of the engine.

15

RESTING CONTACTS AND FRICTION

So far I've used the terms *contacts* and *collisions* interchangeably. The collision detector finds pairs of objects that are touching (i.e., in contact) or interpenetrating. The collision resolution algorithm manipulates these objects in physically believable ways.

From this point on I will make a distinction between the two terms: a *contact* is any location in which objects are touching or interpenetrating; a *collision* is a type of contact, one in which the objects are moving together at speed (this is also called an "impact" in some physics systems). This chapter introduces another type of contact: the resting contact. This is a contact where the objects involved are moving neither apart nor together.

For the sake of completeness there is a third type of contact, the separating contact, where the objects involved are already moving apart. There is no need to perform any kind of velocity resolution on a separating contact, so it is often ignored.

The collisions we've seen up to this point are relatively easy to handle: the two objects collide briefly and then go on their own way again. At the point of contact we calculate an impulse that causes the contact to turn from a collision into a separating contact (if the coefficient of restitution is greater than zero) or a resting contact (if it is exactly zero).

When two objects are in contact for a longer period of time (i.e., longer than a single physics update), they are said to have resting contact. In this case they need to be kept apart while making sure each object behaves normally.

15.1 RESTING FORCES

When an object is resting on another, Newton's third and final law of motion comes into play. Newton 3 states: "For every action there is an equal and opposite reaction." We already used this law in chapter 14 for collisions involving two objects. When we calculated the impulse on one object, we applied the same impulse in the opposite direction to the second object. Collisions between objects and the immovable environment used the assumption that any movement of the environment would be so small that it could be safely ignored. In reality, when an object bounces on the ground, the whole earth is also bouncing: the same impulse is being applied to the earth. Of course the earth is so heavy that if we tried to work out the amount of motion that the earth undergoes, it would be vanishingly small, so we ignore it.

When we come to resting contacts, a similar process happens. If an object is resting on the ground, then the force of gravity is trying to pull it through the ground. We feel this force as weight: the force that gravity is applying on a heavy object is great. What isn't as obvious is that there is an equal and opposite force keeping the object on the ground. This is called the "reaction force," and Newton 3 tells us that it is exactly the same as its weight. If this reaction force were not there, then the object would accelerate down through the ground. Figure 15.1 shows the reaction force.

Whenever two objects are in resting contact and not accelerating, there will be a balance of forces at the point of contact. Any force that one object applies to the other will be met with an equal reaction force back. If this balance of forces isn't present, then both objects will be accelerating. We can work out the acceleration using Newton's second law of motion, after working out the total force (including reaction forces) on each object.

There is something of a circular process here, and it gives a taste of some issues to come. If reaction forces can be as large as necessary (we're assuming rigid bodies will never crumble or compress), and acceleration depends on the total forces applied, how do we calculate how big the reaction forces actually are at any time? For simple situations like that in figure 15.1, this isn't a problem, and in most high school and
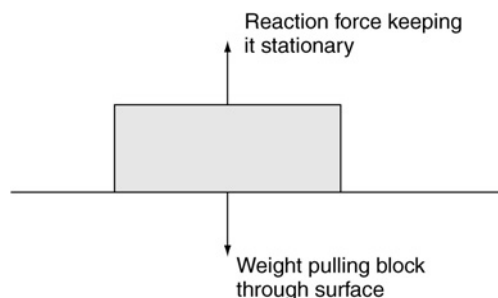
FIGURE 15.1   A reaction force at a resting contact.

undergraduate mathematics the problem is never mentioned. For complex scenarios with lots of interacting objects and especially friction, it is significant, as we will see.

Notice that the reaction force between the ground and an object is a real force. It isn't an impulse: there is no change in velocity. So far in our collision resolution system we've only applied impulses. This reaction force cannot be represented in the same way. We need to consider it more fully.

15.1.1 Force Calculations

The most obvious approach to resting contacts is to calculate the reaction forces. That way we can add the forces into the equations of motion of our rigid bodies (using D'Alembert's principle, as in section 10.3). With the reaction forces working alongside the regular forces we apply, the body will behave correctly.

Many physics systems do exactly this. Given a set of contacts, they try to generate a set of reaction forces that will keep the objects from accelerating together. For colliding contacts they use one of two methods: they use either the same impulse method we saw in chapter 14; or they use the fact that an impulse is simply a force applied over a small moment of time—if we know the time (i.e., the duration of the physics update), then the impulse can be turned into a one-off force and resolved in the same way as other forces.

This approach is okay if you can accurately calculate the reaction forces every time. For simple situations, such as an object resting on the ground, this is very easy. But it rapidly gets more complex. Figure 15.2 shows a stack of objects. There are many internal reaction forces in this stack. The reaction forces at the bottom of the stack depend on the reaction forces at the top of the stack. The forces that need to be applied at a contact may depend heavily on contacts at a completely different location in the simulation, with no common objects between them.

To calculate reaction forces we cannot use an iterative algorithm like those in the last chapter. We have to take a more global view, representing all the force interactions in one mathematical structure and creating a one-for-all solution. In most cases this can be done, and it is the mathematical core of most commercial physics middleware packages. We'll look at the techniques used when we get to chapter 18.

In some cases, especially when there is friction at resting contacts, there is no solution. The combination of reaction forces cannot be solved. This often occurs when the simulation drifts (through numerical calculation errors or because of stepping through time and missing the exact time of contact) into a state that could not occur in reality. The computer is trying to solve a problem that is literally impossible.

The same problem can also occur because we are assuming perfectly rigid bodies, where in reality all objects can be compressed to some degree. And finally it can occur when what appears to be a resting contact would in reality be a collision. If you slide an object along a rough surface, for example, you may be able to get it to suddenly leap into the air. This occurs because a contact that appears to be a resting contact with the ground may in reality be a collision against a patch of high friction.
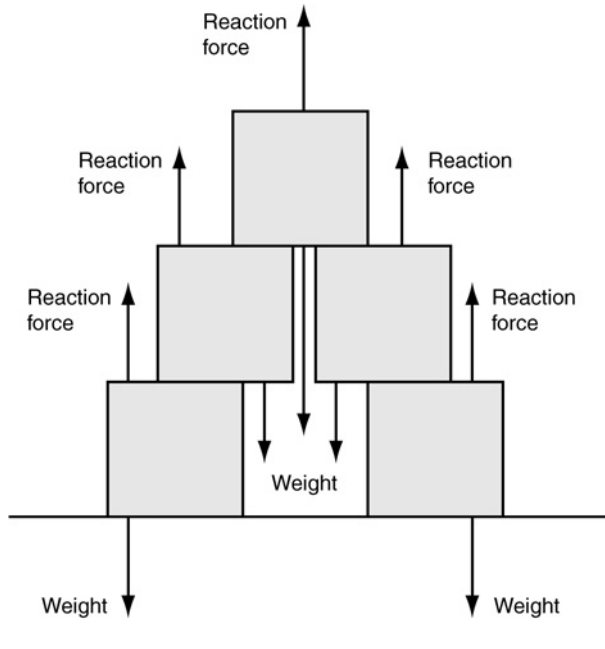
FIGURE 15.2 The long-distance dependence of reaction forces.

Each of these situations leads to problems in solving the mathematics to get a set of reaction forces. Special-case code or tailored solving algorithms are needed to detect impossibilities and react differently to them (typically by introducing an impulse of some kind).

If this sounds complex, it's because it is. Fortunately there is a much simpler (though slightly less accurate) solution. Rather than resolving all contacts using forces (i.e., converting collision impulses into forces), we can do the opposite and treat resting contacts as if they were collisions.

15.2 Micro-collisions

Micro-collisions replace reaction forces by a series of impulses: one per update. In the same way that an impulse can be thought of as a force applied in a single moment of time, a force can be thought of as a whole series of impulses. Applying a force of $10f$ N to an object over 10 updates is equivalent to applying impulses of f Ns at each update.

Rather than calculate a set of reaction forces at each time step, we allow our impulse resolution system to apply impulses. Figure 15.3 shows this in practice. The block should be resting on the ground. At each frame (ignoring interpenetration for
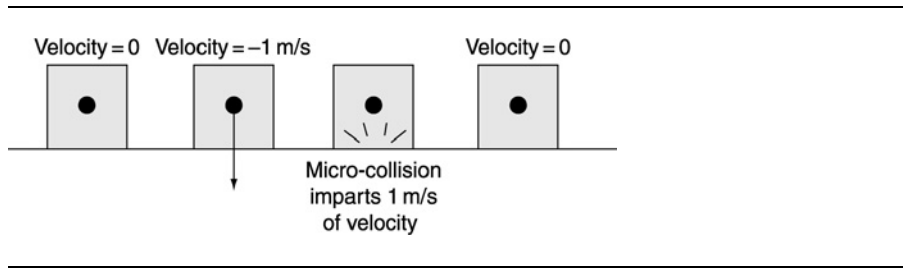
FIGURE 15.3   Micro-collisions replace reaction forces.

a while), the block accelerates so it has a velocity into the ground. The velocity resolution system calculates the impulse needed to remove that velocity.

These little impulses are sometimes called "micro-collisions." Their use is a well-known technique for generating reaction forces, but suffers from an undeserved reputation for producing unstable simulations.

If you run the physics engine from chapter 14, you will see that objects don't sink into one another, even though there is no reaction force at work. Micro-collisions are already at work: at each update objects are building up velocity, only to have the velocity resolution algorithm treat contacts as collisions and remove the velocity.

There are two significant problems with treating resting contacts as collisions. The first has to do with the way collisions bounce. Recall that the separation speed at a contact point is calculated as a fixed ratio of the closing speed, in the opposite direction. This ratio is the coefficient of restitution.

If we have a contact such as that shown in figure 15.3, after the rigid-body update, the velocity into the ground will have built up. During the velocity resolution process this velocity will be removed. The desired final velocity will be

$$v'_s = -cv_s$$

where v_s is the velocity before the collision is processed, v'_s is the same velocity after processing, and c is the coefficient of restitution.

So whatever velocity built up over the course of the interval between updates will cause a little "bounce" to occur. If our sphere on the ground had a high c value, the downward velocity would generate an upward velocity.

But in reality the downward velocity never gets a chance to build up. The sphere can't really accelerate into the ground. The velocity it accumulates is physically impossible. This has the effect of making resting contacts appear to vibrate. The objects accelerate together, building up velocity that then causes the collision algorithm to give them a separating velocity. The sphere on the ground bounces up until gravity brings it back down, whereupon it bounces again. It will never settle to rest, but will appear to vibrate.

Setting a lower coefficient of restitution will help but limits the kinds of situations that can be modeled. A more useful solution involves making two changes:

■ We remove any velocity that has been built up from acceleration in the previous rigid-body update.

■ We artificially decrease the coefficient of restitution for collisions involving very low speeds.

Independently each of these can solve the vibration problem for some simulations but will still show problems in others. Together they are about as good as we can get.

15.2.1 Removing Accelerated Velocity

To remove the velocity due to the previous frame's acceleration, we need to keep track of the acceleration at each rigid-body update. We can do this with a new data member for the rigid body, `accelerationAtUpdate`, which stores the calculated linear acceleration. The rigid-body update routine is then modified to keep a record in this variable of the acceleration generated by all forces and gravity:

Excerpt from src/body.cpp

```
// Calculate linear acceleration from force inputs.
lastFrameAcceleration = acceleration;
lastFrameAcceleration.addScaledVector(forceAccum, inverseMass);
```

We could extend this to keep a record of both linear and angular acceleration. This would make it more accurate, but since most reaction forces are generated by the gravity (which is always linear), the extra calculations don't normally give any visible benefit. In fact some developers choose to ignore any force except gravity when calculating the velocity added in the last frame. This makes the calculation simpler still, as we can read the acceleration due to gravity from the `acceleration` data member directly.

When we calculate the desired change in velocity for a contact, we subtract the acceleration-induced velocity, in the direction of the contact normal:

$$\Delta v = -v_{\mathrm{acc}} - (1 + c)(v_s - v_{\mathrm{acc}})$$

The desired change in velocity is modified from

```
deltaVelocity = -(1+restitution) * contactVelocity;
```

to

```
real velocityFromAcc = body[0]->accelerationAtUpdate * contactNormal;

if (body[1])
{
    velocityFromAcc -= body[1]->accelerationAtUpdate * contactNormal;
```

```
    }

    real deltaVelocity = -contactVelocity.x -
                      restitution *
                      (contactVelocity.x - velocityFromAcc);
```

Making this simple adjustment reduces the amount of visual vibration for objects resting on the ground. When objects are in tight groups, such as stacks, the vibration can return. To solve that problem we'll perform the second step—reducing the coefficient of restitution.

We'll return to the velocity caused by acceleration later in this chapter. We will need another calculation of this kind to solve a problem with friction at resting contacts.

15.2.2 Lowering the Restitution

The change we made in the previous section effectively reduces the restitution at contacts. Before reducing the velocity we have collisions with greater separating velocity than closing velocity: the objects are pushed apart even when they begin resting. This occurs when there is a coefficient of restitution above 1. The smaller the coefficient, the less bounce there will be.

When the acceleration compensation alone doesn't work, we can manually lower the coefficient of restitution to discourage vibration. This can be done in a very simple way:

```
real appliedRestitution = restitution;
if (contactVelocity.magnitude() < velocityLimit)
{
    appliedRestitution = (real)0.0f;
}
```

We could use a more sophisticated method, where the restitution is scaled so that it is smaller for smaller velocities, but the version here works quite well in practice. If you see visible transitions between bouncing and sticking as objects slow down, try reducing the velocity limit (I use a value of around 0.1 in my engine). If this introduces vibration, then the scaling approach may be useful to you.

15.2.3 The New Velocity Calculation

Combining both techniques for resting contacts, we end up with the following code in our adjustVelocities method:

```
_____ Excerpt from src/contacts.cpp _____
// Calculate the acceleration-induced velocity accumulated this frame.
real velocityFromAcc = body[0]->getLastFrameAcceleration() *
                       duration * contactNormal;

if (body[1])
{
    velocityFromAcc -= body[1]->getLastFrameAcceleration() *
                       duration * contactNormal;
}

// If the velocity is very slow, limit the restitution.
real thisRestitution = restitution;
if (real_abs(contactVelocity.x) < velocityLimit)
{
    thisRestitution = (real)0.0f;
}

// Combine the bounce velocity with the removed
// acceleration velocity.
desiredDeltaVelocity =
    -contactVelocity.x -
    thisRestitution * (contactVelocity.x - velocityFromAcc);
```

I have placed this series of operations in its own method, `calculateDesired-DeltaVelocity`, which is called as part of the `calculateInternals` method. This is preferable to having the calculations performed each time the velocity resolver tries to find the most severe collision.

This approach removes almost all the visible vibrations in the cyclone physics engine. One of the optimization techniques we'll meet in the next chapter removes the rest.

15.3 Types of Friction

I've mentioned friction several times throughout the book, and now it's time to tackle it head on.

Friction is the force generated when one object moves or tries to move in contact with another. No matter how smooth two objects look, microscopically they are rough, and these small protrusions catch one another, causing a decrease in the relative motion or a resistance to motion beginning.

Friction is also responsible for a small part of drag, when air molecules try to move across the surface of an object. (Drag has a number of different factors, such as turbulence, induced pressure, and collisions between the object and air molecules.)

There are two forms of friction: static and dynamic. They behave in slightly different ways.

15.3.1 Static and Dynamic Friction

Static friction is a force that stops an object from moving when it is stationary. Consider a block that is resting on the ground. If the block is given some force, friction between the block and the ground will resist this force. This is a kind of reaction force: the more you push, the more friction pushes back. At some point, however, the pushing force is too much for friction, and the object begins to move.

Because static friction keeps objects from moving, it is sometimes called "stiction." The static friction force depends on the materials at the point of contact and the reaction force:

$$|f_{\text{static}}| \leqslant \mu_{\text{static}}|r|,$$

where r is the reaction force in the direction of the contact normal, f_{static} is the friction force generated, and μ_{static} is the coefficient of static friction.

The coefficient of friction encapsulates all the material properties at the contact in a single number. The value depends on both objects; it cannot be generated simply by adding a coefficient for one object to one for another. In fact it is an empirical quantity: it is discovered by experiment and cannot be reliably calculated.

In physics reference books you will often find tables of the coefficient of friction for different pairs of materials. In a game development setting the coefficient for a particular contact is more often the result of guesswork or trial and error. I have included a table of friction coefficients that I find useful in appendix B.

Notice that the preceding formula is an inequality: it uses the $\leqslant$ symbol. This means that the magnitude of the friction force can be anything up to and including $\mu|r|$. In fact, up to this limit it will be exactly the same as the force exerted on the object. So the overall expression for the static friction force is

$$f_{\text{static}} = \begin{cases} -f_{\text{planar}} \\ \widehat{f}_{\text{planar}} - \mu_{\text{static}}|r| \end{cases} \quad \text{whichever is smaller in magnitude,}$$

where f_{planar} is the total force on the object in the plane of the contact only, because the resulting force in the direction of the contact normal is generating the reaction force. The reaction force and the planar force can be calculated from the total force applied:

$$r = -f \cdot \widehat{d}$$

where $\widehat{d}$ is the contact normal and f is the total force exerted, and

$$f_{\text{planar}} = f + r$$

In other words, the resulting planar force is the total force with the component in the direction of the contact normal removed. In the equation this component is removed by adding the reaction force, which is equal and opposite to the force in the direction of the contact and therefore cancels it out.

The dependence of static friction on the normal reaction force is an important result. It allows rock climbers to walk up a (more than) vertical slope by pushing against another wall at their back—the increase in reaction force means increased friction. Push hard enough and there'll be enough friction to overcome your weight and keep you from falling.

Another important feature of the previous equations is that friction doesn't depend on the area that is in contact with the ground. A rock climber with bigger feet doesn't stick better. Despite being slightly counterintuitive (for me at least), this is fortunate because nowhere in our engine have we considered the size of the contact area. Contact area does become important in some cases where the objects can deform at the point of contact (tire models spring immediately to mind), but they are very complex and well beyond the scope of this book, so we'll stick with the basic formula.

Returning to our block on the ground: as we exert more force, friction pushes back until we reach $\mu_{\text{static}}|\boldsymbol{r}|$, the limit of static friction. If we increase the force input by a fraction, the friction force drops suddenly and we enter the world of dynamic friction.

Dynamic Friction

Dynamic friction, also called "kinetic friction," behaves in a similar way to static friction but has a different coefficient of friction.

When objects at the contact are moving relative to one another, they are typically leaving contact at the microscopic level. Figure 15.4 shows static and dynamic friction magnified many times. Once the object is in motion, the roughness on each object isn't meshing as closely, so dynamic friction is a less powerful force.

Dynamic friction always obeys the equation

$$\boldsymbol{f}_{\text{dynamic}} = -\hat{\boldsymbol{v}}_{\text{planar}}\,\mu_{\text{dynamic}}|\boldsymbol{r}|$$

where μ_{dynamic} is the coefficient of dynamic friction. Notice that the direction of friction has changed. Rather than acting in the opposite direction to the planar force (as it did for static friction), it now acts in the opposite direction to the velocity of the object. This is significant: if you stop exerting a force on a stationary object, then the friction force will instantly stop too. If you stop exerting a force on a moving object, friction will not stop: the object will be slowed to a halt by dynamic friction.

Just like static friction, dynamic friction coefficients can be found in some physics reference books for different combinations of materials.

It is rare in game physics engines to distinguish in practice between static and dynamic friction. They tend to be rolled together into a generic friction value. When the object is stationary, the friction acts as static friction, acting against any force exerted.
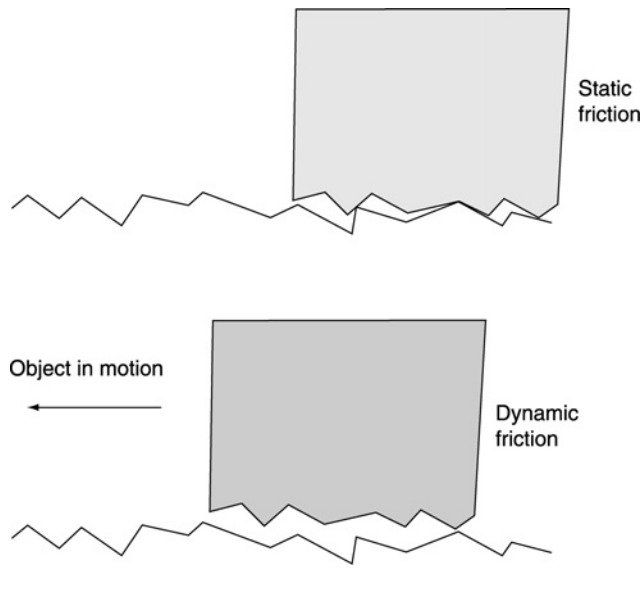
FIGURE 15.4   A microscopic view of static and dynamic friction.

When the object is moving, the friction acts as dynamic friction, acting against the direction of motion.

The friction model we'll develop in this chapter will follow this model and combine both types of friction into one value. In what follows I will point out where we are using static friction and where it is dynamic, so you can replace the single value with two values if you need to.

Rolling Friction

There is one further type of friction that is important in dynamic simulation. Rolling friction occurs when one object is rolling along another. It is most commonly used for high-quality tire models for racing simulations (in the sense of simulations performed by motor-racing teams rather than in racing games).

I have not come across physics engines for games with a comprehensive tire model that includes rolling friction. I have worked with one non-game physics engine that included it, however. Because we are focusing on game applications, I will ignore rolling friction for the rest of the book.

15.3.2   Isotropic and Anisotropic Friction

There is one additional distinction between types of friction that we need to recognize: friction can be either isotropic or anisotropic. Isotropic friction has the same
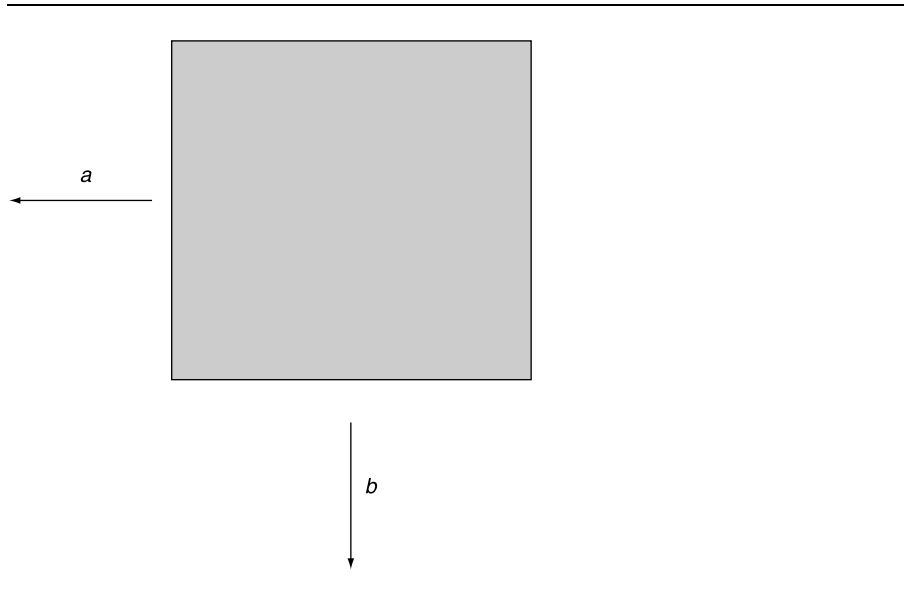
FIGURE 15.5 Anisotropic friction.

coefficient in all directions. Anisotropic friction can have different coefficients in different directions.

Figure 15.5 shows a block on the ground from above. If it is pushed in the first direction, then the friction force will have a coefficient of μ_a; if it is pushed in the second direction, then the friction force will have a coefficient of μ_b. If $\mu_a = \mu_b$, then the friction is isotropic; otherwise it is anisotropic.

The vast majority of game simulations only need to cope with isotropic friction. In fact most engines I've used either are purely isotropic or make the programmer jump through extra hoops to get anisotropic friction. Even then, the anisotropic friction model is highly simplified. We'll stick with isotropic friction in this book.

15.4 IMPLEMENTING FRICTION

Introducing friction into a physics simulation depends on how the existing physics is implemented. In our case we have an impulse-based engine with micro-collisions for resting contacts. This means we have no calculated normal reaction forces at resting contacts. In addition, we introduce impulses rather than forces at contacts to generate believable behavior.

This makes it difficult to directly carry across the friction equations we have seen so far: we have no calculation of the reaction force, and we have no easy way of applying forces at the contact point (remember that in our engine the forces for the current physics update are applied before collision detection begins).

If you are working with a force-based engine, especially one that calculates the reaction forces for all contacts, then friction can become another force in the calculation, and the equations we have seen can be applied directly. Although this sounds simpler, there are consequences that make it even more difficult to calculate the required forces. I'll return to friction-specific force calculation in chapter 17. At this stage it is simply worth noting that, despite the modifications we'll have to make to convert friction into impulses, if we had gone through the force-only route initially, it wouldn't have made friction any easier.

15.4.1 FRICTION AS IMPULSES

To handle friction in our simulation we must first understand what friction is doing in terms of impulses and velocity.

Static friction stops a body from moving when a force is applied to it. It acts to keep the velocity of the object at zero in the contact plane.

In our simulation we allow velocity to build up, and then we remove it with a micro-collision. We can simulate static friction by removing sliding velocity along with collision velocity. We already adjust the velocity of the object in the direction of the contact normal. We could do something similar in the other two contact directions (i.e., the directions that are in the plane of the contact). If we went through the same process for each direction as we did for the contact normal, we could ensure that the velocity in these directions was zero. This would give the effect of static friction.

Rather than having a single value for the change in velocity, we now have a vector, expressed in contact coordinates:

```
real deltaVelocity; // ... Calculate this as before ...

Vector3 deltaVelocityVector(deltaVelocity,
                            -contactVelocity.y,
                            -contactVelocity.z);
```

I'll come back later to the changes needed in the resolution algorithm to cope with this.

This approach would remove all sliding. But static friction has a limit: it can only prevent objects from sliding up to a maximum force. When dealing with the collision, we don't have any forces, only velocities. How do we decide the maximum amount of velocity that can be removed?

Recall that velocity is related to impulse:

$$\Delta \dot{p} = m^{-1} g$$

where g is impulse, m is mass, and $\dot{p}$ is velocity. So, if we know the amount of velocity we need to remove, we can calculate the impulse required to remove it.

In the same way, impulse is a force exerted over a short period of time:

$$g = ft$$

where f is force and t is time. Given the impulse required to remove the velocity and the duration of the update, we can calculate the force required to remove the velocity.

The equation still requires a normal reaction force. This can be calculated in the same way, but looking at the contact normal. The normal reaction force can be approximately calculated from the amount of velocity removed in the direction of the contact normal.

If the desired change in velocity at the contact normal is v, then the reaction force can be approximated as

$$f = \Delta vmt$$

The velocity resolution algorithm we already have involves calculating the impulse needed to achieve the desired change in velocity. This impulse is initially found in contact coordinates. Since we will be working in impulses, we can combine the preceding equations with the friction equations, to end up with

$$g_{max} = \Delta g_{normal} \, \mu$$

where Δg_{normal} is the impulse in the direction of the contact normal (i.e., the impulse we are currently calculating in our velocity resolution algorithm). Notice that these impulse values are scalar. This tells us the total impulse we can apply with static friction. In code this looks like

```
Vector3 impulseContact;

// ... Find the impulse required to remove all three components of
// velocity (we'll return to this algorithm later) ...

real planarImpulse = real_sqrt(impulseContact.y*impulseContact.y +
                               impulseContact.z*impulseContact.z);

// Check whether we're within the limit of static friction.
if (planarImpulse > impulseContact.x * friction)
{
    // Handle as dynamic friction.
}
```

Dynamic friction can be handled by scaling the y and z components of the impulse so that their combined size is exactly μ times the size of the x impulse:

```
impulseContact.y /= planarImpulse;
impulseContact.z /= planarImpulse;

impulseContact.y *= friction * impulseContact.x;
impulseContact.z *= friction * impulseContact.x;
```

Dividing by `planarImpulse` scales the y and z components so that they have a unit size; this is done to preserve their direction while removing their size. Their size is given by the friction equation—`friction * impulseContact.x`. Multiplying the direction by the size gives the new values for each component.

15.4.2 Modifying the Velocity Resolution Algorithm

In the previous section I glossed over how we might calculate the impulses needed to remove velocity in the plane of the contact. We already have code that does this for the contact normal, and we could simply duplicate this for the other directions.

Unfortunately, along with being very long-winded, this wouldn't work very well. An impulse in one direction can cause an object to spin, and its contact point can begin moving in a completely different direction. As long as we were only interested in velocity in the direction of the contact normal, this didn't matter. Now we need to handle all three directions at the same time, and we need to take into account the fact that an impulse in one direction can increase the velocity of the contact in a different direction. To resolve the three velocities we need to work through the resolution algorithm for each simultaneously.

The resolution algorithm has the following steps, as before:

1. We work in a set of coordinates that are relative to the contact: this makes much of the mathematics a lot simpler. We create a transform matrix to convert into and out of this new set of coordinates.

2. We work out the change in velocity of the contact point on each object per unit impulse. Because the impulse will cause linear and angular motion, this value needs to take account of both components.

3. We will know the velocity change we want to see (in the next step), so we invert the result of the last stage to find the impulse needed to generate any given velocity change.

4. We work out what the separating velocity at the contact point should be, what the closing velocity currently is, and the difference between the two. This is the desired change in velocity.

5. From the change in velocity we can calculate the impulse that must be generated.

6. We split the impulse into its linear and angular components and apply them to each object.

We have inserted a new step between 5 and 6, checking whether the impulse respects the static friction equation and using dynamic friction if it doesn't.

Step 2 requires modification. Currently it works out the change in velocity given a unit impulse in the direction of the contact normal. We are now dealing with all three contact directions. We need to calculate the change in velocity given any combination of impulses in the three contact directions. The impulse can be expressed as a vector in contact coordinates:

```
Vector3 contactImpulse;
```

The *x* component represents the impulse in the direction of the contact normal, and the *y* and *z* components represent the impulse in the plane of the contact.

The result of step 2 will be a matrix: it will transform a vector (the impulse) into another vector (the resulting velocity). With this matrix the rest of the algorithm is simple. In step 3 we will find the inverse of the matrix (i.e., the matrix that transforms the desired change in velocity into a required impulse), and in step 5 we will transform the desired velocity vector to get the `contactImpulse` vector.

So how do we calculate the matrix? We follow through the same steps we saw in section 14.2.2. We calculate the velocity change as a result of angular motion, and the velocity change as a result of linear motion.

Velocity from Angular Motion

In section 14.2.2 we saw the algorithm for calculating rotation-derived velocity from impulse:

```
Vector3 torquePerUnitImpulse =
    relativeContactPosition % contactNormal;

Vector3 rotationPerUnitImpulse =
    inverseInertiaTensor.transform(torquePerUnitImpulse);

Vector3 velocityPerUnitImpulse =
    rotationPerUnitImpulse % relativeContactPosition;

Vector3 velocityPerUnitImpulseContact =
    contactToWorld.transformTranspose(velocityPerUnitImpulse);
```

The first stage calculates the amount of torque for a unit impulse in the direction of the contact normal. The second stage converts this torque into a velocity using the inertia tensor. The third stage calculates the linear velocity of the contact point from the resulting rotation. And the final stage converts the velocity back into contact coordinates.

Rather than use the contact normal in the first stage, we need to use all three directions of the contact: the basis matrix. But if the contact normal is replaced by a matrix, how do we perform the cross product?

The answer lies in an alternative formation of the cross product. Remember that transforming a vector by a matrix gives a vector. The cross product of a vector also gives a vector. It turns out that we can create a matrix form of the vector product.

For a vector

$$v = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

the vector product

$$v \times x$$

is equivalent to the matrix-by-vector multiplication:

$$\begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix} x$$

This matrix is called a "skew-symmetric" matrix, and an important result about cross products is that the cross product is equivalent to multiplication by the corresponding skew-symmetric matrix.

Because, as we have seen, $v \times x = -x \times v$; and if we already have the skew-symmetric version of v, we can calculate $x \times v$ without building the matrix form of x. It is simply

$$x \times v = - \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix} x$$

In fact we can think of the cross product in the first stage of our algorithm as turning an impulse into a torque. We know from equation 10.1 that a force vector can be turned into a torque vector by taking its cross product with the point of contact:

$$\tau = p_f \times f$$

(which is just equation 10.1 again).

The skew-symmetric matrix can be thought of as this transformation, turning force into torque.

It is useful to have the ability to set a matrix's components from a vector, so we add a convenience function to the `Matrix3` class:

```
                        ── Excerpt from include/cyclone/core.h ──
/**
 * Holds an inertia tensor, consisting of a 3x3 row-major matrix.
 * This matrix is not padding to produce an aligned structure, since
 * it is most commonly used with a mass (single real) and two
 * damping coefficients to make the 12-element characteristics array
 * of a rigid body.
 */
class Matrix3
    // ... Other Matrix3 code as before ...

    /**
     * Sets the matrix to be a skew-symmetric matrix based on
     * the given vector. The skew-symmetric matrix is the equivalent
     * of the vector product. So if a,b are vectors, a x b = A_s b
     * where A_s is the skew-symmetric form of a.
     */
    void setSkewSymmetric(const Vector3 vector)
    {
        data[0] = data[4] = data[8] = 0;
        data[1] = -vector.z;
        data[2] = vector.y;
        data[3] = vector.z;
        data[5] = -vector.x;
        data[6] = -vector.y;
        data[7] = vector.x;
    }
};
```

Now we can work the whole basis matrix through the same series of steps:

```
// Create the skew-symmetric form of the cross product.
Matrix3 impulseToTorque;
impulseToTorque.setSkewSymmetric(relativeContactPosition);

// This was a cross product.
Matrix3 torquePerUnitImpulse = impulseToTorque * contactToWorld;

// This was a vector transformed by the tensor matrix - now it's
// just plain matrix multiplication.
Matrix3 rotationPerUnitImpulse =
    inverseInertiaTensor * torquePerUnitImpulse;

// This was the reverse cross product, so we'll need to multiply the
```

```
// result by -1.
Matrix3 velocityPerUnitImpulse =
    rotationPerUnitImpulse * impulseToTorque;
velocityPerUnitImpulse *= -1;

// Finally convert the result into contact coordinates.
Matrix3 velocityPerUnitImpulseContact =
    contactToWorld.transpose() * velocityPerUnitImpulse;
```

The resulting matrix, velocityPerUnitImpulseContact, can be used to transform an impulse in contact coordinates into a velocity in contact coordinates. This is exactly what we need for this stage of the algorithm.

In practice there may be two objects involved in the contact. We can follow the same process through each time and combine the results. The most efficient way to do this is to notice that only the impulseToTorque and inverseInertiaTensor matrices will change for each body. The contactToWorld matrices are the same in each case. We can therefore separate them out and multiply them after the two objects have been processed independently. The code looks like this:

───── **Excerpt from src/contacts.cpp** ─────

```
// The equivalent of a cross product in matrices is multiplication
// by a skew symmetric matrix - we build the matrix for converting
// between linear and angular quantities.
Matrix3 impulseToTorque;
impulseToTorque.setSkewSymmetric(relativeContactPosition[0]);

// Build the matrix to convert contact impulse to change in velocity
// in world coordinates.
Matrix3 deltaVelWorld = impulseToTorque;
deltaVelWorld *= inverseInertiaTensor[0];
deltaVelWorld *= impulseToTorque;
deltaVelWorld *= -1;

// Check if we need to add body two's data
if (body[1])
{
    // Find the inertia tensor for this body.
    body[1]->getInverseInertiaTensorWorld(&inverseInertiaTensor[1]);

    // Set the cross product matrix.
    impulseToTorque.setSkewSymmetric(relativeContactPosition[1]);

    // Calculate the velocity change matrix.
    Matrix3 deltaVelWorld2 = impulseToTorque;
```

```
    deltaVelWorld2 *= inverseInertiaTensor[1];
    deltaVelWorld2 *= impulseToTorque;
    deltaVelWorld2 *= -1;

    // Add to the total delta velocity.
    deltaVelWorld += deltaVelWorld2;
}

// Do a change of basis to convert into contact coordinates.
Matrix3 deltaVelocity = contactToWorld.transpose();
deltaVelocity *= deltaVelWorld;
deltaVelocity *= contactToWorld;
```

where the same matrix is reused for intermediate stages of the calculation, as in chapter 14.

What we are effectively doing here is performing all the calculations in world coordinates (i.e., we end up with a matrix that transforms impulse into velocity, both in world coordinates) for each body. Then we add the two results together, and then change the basis of this matrix so that it transforms impulse into velocity in contact coordinates. Recall from section 9.4.6 that we change the basis of a matrix by

$$BMB^{-1}$$

where B is the basis matrix and M is the matrix being transformed. This is equivalent to $BMB^\top$ when B is a rotation matrix only (as it is for the contactToWorld matrix). Hence the last three lines of the code snippet.

Velocity from Linear Motion

So far we only have the change in velocity caused by rotation. We also need to include the change in linear velocity from the impulse. As before, this is simply given by the inverse mass:

$$\Delta \dot{p} = m^{-1} g$$

This again is a transformation from a vector (impulse) into another vector (velocity). Because we are trying to end up with one matrix combining linear and angular components of velocity, it would be useful to express inverse mass as a matrix so that it can be added to the angular matrix we already have.

This can be done simply. Multiplying a vector by a scalar quantity k is equivalent to transforming it by the matrix

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

You can manually check this by trying a vector multiplication.

To combine the linear motion with the angular motion we already have, we need only add the inverse mass to the diagonal entries of the matrix:

```
deltaVelocity.data[0] += inverseMass;
deltaVelocity.data[4] += inverseMass;
deltaVelocity.data[8] += inverseMass;
```

15.4.3 PUTTING IT ALL TOGETHER

We are now ready to put together all the modifications we need to support isotropic friction. These modifications are only made to the applyVelocityChange method of the contact: they are all handled as a micro-collision. The final code looks like this:

```
─────────── Excerpt from src/contacts.cpp ───────────
real inverseMass = body[0]->getInverseMass();

// The equivalent of a cross product in matrices is multiplication
// by a skew-symmetric matrix - we build the matrix for converting
// between linear and angular quantities.
Matrix3 impulseToTorque;
impulseToTorque.setSkewSymmetric(relativeContactPosition[0]);

// Build the matrix to convert contact impulse to change in velocity
// in world coordinates.
Matrix3 deltaVelWorld = impulseToTorque;
deltaVelWorld *= inverseInertiaTensor[0];
deltaVelWorld *= impulseToTorque;
deltaVelWorld *= -1;

// Check whether we need to add body 2's data.
if (body[1])
{
    // Find the inertia tensor for this body.
    body[1]->getInverseInertiaTensorWorld(&inverseInertiaTensor[1]);

    // Set the cross product matrix.
    impulseToTorque.setSkewSymmetric(relativeContactPosition[1]);

    // Calculate the velocity change matrix.
    Matrix3 deltaVelWorld2 = impulseToTorque;
    deltaVelWorld2 *= inverseInertiaTensor[1];
    deltaVelWorld2 *= impulseToTorque;
    deltaVelWorld2 *= -1;
```

```
    // Add to the total delta velocity.
    deltaVelWorld += deltaVelWorld2;

    // Add to the inverse mass.
    inverseMass += body[1]->getInverseMass();
}

// Do a change of basis to convert into contact coordinates.
Matrix3 deltaVelocity = contactToWorld.transpose();
deltaVelocity *= deltaVelWorld;
deltaVelocity *= contactToWorld;

// Add in the linear velocity change.
deltaVelocity.data[0] += inverseMass;
deltaVelocity.data[4] += inverseMass;
deltaVelocity.data[8] += inverseMass;

// Invert to get the impulse needed per unit velocity.
Matrix3 impulseMatrix = deltaVelocity.inverse();

// Find the target velocities to kill.
Vector3 velKill(desiredDeltaVelocity,
                -contactVelocity.y,
                -contactVelocity.z);

// Find the impulse to kill target velocities.
impulseContact = impulseMatrix.transform(velKill);

// Check for exceeding friction.
real planarImpulse = real_sqrt(impulseContact.y*impulseContact.y +
                               impulseContact.z*impulseContact.z);
if (planarImpulse > impulseContact.x * friction)
{
    // We need to use dynamic friction.
    impulseContact.y /= planarImpulse;
    impulseContact.z /= planarImpulse;

    impulseContact.x = deltaVelocity.data[0] +
        deltaVelocity.data[1] * friction * impulseContact.y +
        deltaVelocity.data[2] * friction * impulseContact.z;
    impulseContact.x = desiredDeltaVelocity / impulseContact.x;
    impulseContact.y *= friction * impulseContact.x;
```

```
      impulseContact.z *= friction * impulseContact.x;
}
```

The impulse is then applied in exactly the same way as for the non-friction case.

15.5 FRICTION AND SEQUENTIAL CONTACT RESOLUTION

With the modifications in this chapter our physics engine has taken a huge leap forward. It is now capable of modeling rigid bodies with all kinds of contacts and isotropic friction.

There are still some lingering stability issues that we can look at and a huge increase in performance we can expect. We'll examine both of these in the next chapter.

At this stage we can also see the main unavoidable limitation of the micro-collision approach to physics. And no amount of tweaking will make this go away completely.

Figure 15.6 shows a typical situation in which two boxes are in contact with each other. Neither of the boxes is moving, and all contacts have very high friction (let's say it is infinite: the static friction can never be overcome).
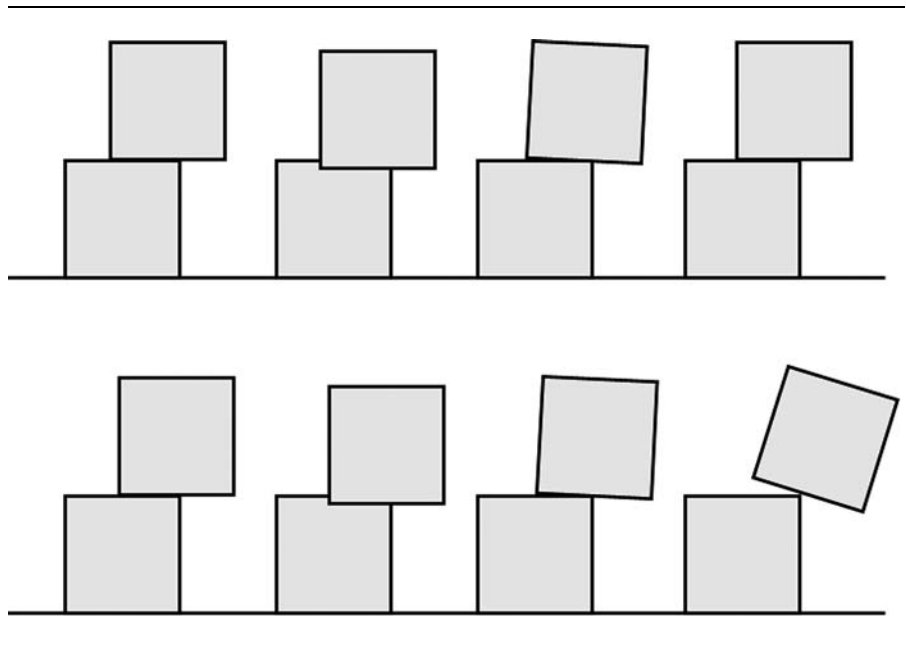
FIGURE 15.6   The problem with sequential contact resolution.

In the first part of the sequence the boxes are in resting contact; in the second part they have interpenetrated slightly as the result of gravity. In the third part they have had their interpenetration resolved, with both linear and angular components to the resolution. In the fourth part of the sequence the contact resolution is complete. The first three parts of the second line of the sequence show another iteration of the same process: interpenetration, penetration resolution, and contact resolution.

Over time it is clear that the boxes are moving apart. They are sliding apart, even though they have infinite friction. By the time we reach the final part of the sequence, the top box in the figure has moved so far it can no longer be supported by the lower box and has begun to fall.

This is caused by the sequential contact resolution scheme. While the resolution algorithm is considering contact A, it cannot also be considering contact B. But when we have friction, the coefficient of friction at B has an effect on how contact A should be resolved. No amount of minor adjustment will solve this problem: to get around it we would need to process contacts simultaneously or create a good deal of special-case code to perform contact-sensitive penetration resolution.

In practice this isn't a major problem unless you are building stacks of blocks. Even in this case the sleeping system we will build in the next chapter ensures that the sliding will occur only after the player disturbs the stack. If you need to build large stacks of objects that are stable to slight knocks, you can either use one of the simultaneous resolution approaches in chapter 18 or use fracture physics, which is described in chapter 17.

15.6 Summary

Resting contacts can be dealt with as if they were tiny little bouncing contacts: the contact interpenetration is resolved, and the closing velocity is killed by applying a small impulse.

By reducing the resting forces over the whole duration of one simulation frame into just an instant of impulse, we were able to simply add friction to the engine. The effects of friction modify the impulse before it is applied to the objects in contact. This is a simple and powerful approach to friction, but it isn't without its problems. It is much more difficult to show the difference between static and dynamic friction using micro-collisions (in fact we've avoided the problem by combining the coefficients into one value).

Another problem with contacts simulated using micro-collisions is that they can appear to vibrate slightly. This is one of a set of stability problems that our current engine implementation faces. In chapter 16 we'll look at stability as a whole, and improve our engine's realism. Then we'll look at how to improve its performance by optimizing the code to do less unnecessary work.
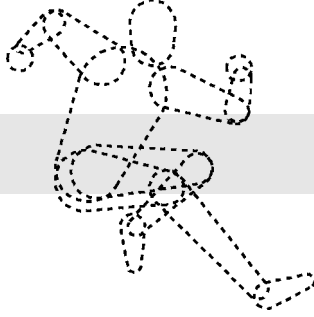
16

STABILITY AND OPTIMIZATION

The physics engine we've built so far is perfectly usable. As it stands, however, there are two criticisms that can be leveled:

- Occasionally strange effects are visible—for example, objects may appear squashed or skewed, objects may slide down hills despite gravity, or fast-moving objects may not behave believably.
- For very large numbers of objects, the simulation can be slow.

We can address these problems to arrive at our final physics engine implementation, one that is powerful and robust enough to be used in a wide range of games. The remaining chapters in the book look at ways of applying or extending the engine, but in this chapter we'll aim to polish our implementation into a stable and fast system.

16.1 STABILITY

Stability problems in our engine, as in all game software, arise from several directions:

- Unpleasant interactions among different bits of the software that individually behave reasonably.
- Inaccuracies of the equations used or adverse effects of assumptions we've made.
- The inherent inaccuracy of the mathematics performed by the computer.

These stability problems can become evident through visually odd behavior, algorithms occasionally not working, or even sudden crashes.

For physics engines in particular, there are a couple of common bugbears that you are almost guaranteed to see during development: sudden, unexpected motion—when an object leaps of the ground, for example—and objects disappearing. The code on the CD shouldn't display either of these critical problems, but chances are you'll see both before long if you make changes and tweaks.

The stability problems we are left with should be more minor, but their causes fall into all three categories. By carefully testing the engine I identified five problems that have relatively easy stability fixes.

- Transform matrices can be malformed and perform skews in addition to rotations and translations.
- Fast-moving objects can sometimes respond oddly to collisions. (This is independent of whether the collisions are actually detected: a fast-moving object can pass right through another object without a collision being detected.)
- Objects resting on an inclined plane (or resting on another object with a sloping surface) tend to slowly slide down.
- Objects experiencing a pair of high-speed collisions in rapid succession can suddenly interpenetrate the ground.
- The simulation can look unrealistic when large and small quantities are mixed: large and small masses, large and small velocities, large and small rotations, and so on.

Fixes for these stability problems solved the odd behaviors my tests generated. Of course no test is ever going to be exhaustive. I have used physics systems for years before noticing some new issue or error.

As with all software maintenance, you never know when some change will need to be made. And by the same token it is a good idea to keep a copy of the test scenarios you run on your engine, so you can go back and check that your new enhancement hasn't broken anything else.

16.1.1 Quaternion Drift

Transform matrices are generated from the position vector and orientation quaternion of rigid bodies. Both position and orientation (in fact all values that take part in mathematical manipulation) suffer numerical errors while being processed.

Errors in the position vector put an object in the wrong place. This is usually such a small error that it isn't noticeable in any short period of time. If the position changes slowly enough, the viewer will not notice any errors.

The same is true of the orientation vector to some extent. But there is an extra problem: we have an additional degree of freedom in the quaternion. If the four quaternion components get out of sync (i.e., if the quaternion is no longer normalized), then it may not correspond to any valid orientation. None of the code in our engine is particularly sensitive to this, but left for long enough it can cause objects to become visibly squashed. The solution, as we saw in chapter 9, is to renormalize the

quaternion. We don't want to do this if we don't have to (such as after every quaternion operation) because that's just a waste of time.

I have added a quaternion normalization step in the rigid-body update routine just after the quaternion is updated by the velocity and before the transform matrix is created. This ensures that the transform matrix has a valid rotation component.

I admit that this stability fix is a bit contrived. It seemed obvious to me when I first wrote the integration routine that it was a good spot for the quaternion normalization, and so I added it.

I have included it here more by way of illustration. The normal size of the quaternion is an assumption we made early on in the development of the engine. It is easily forgotten and has returned to cause strange effects only after we have a completed engine running for long periods of time. Problems may show up only during QA (quality assurance) testing, and they can be very subtle. Checking and enforcing your assumptions in a way that doesn't massacre performance is key to stabilizing and optimizing the engine.

16.1.2 INTERPENETRATION ON SLOPES

The next issue is more significant. Figure 16.1 shows a block resting on a slope. The slope could be an angled plane or the surface of another object. Gravity is acting in a downward direction.
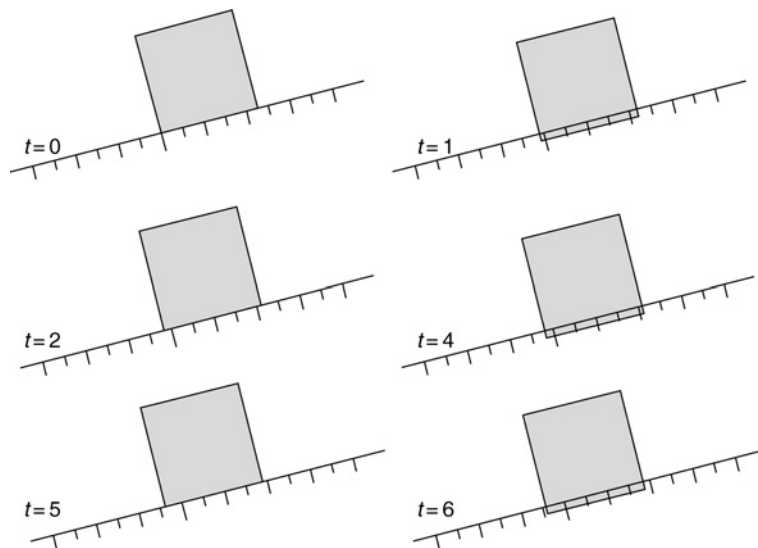
FIGURE 16.1 Objects drift down angled planes.

After one update of the rigid bodies and before collision resolution is performed, the object drops into the plane slightly. This is shown in the second part of the figure. Because the plane contact is in a different direction from the movement of the object, the interpenetration resolution moves the block out to the position as shown in the third part of the figure. Over time, and despite high friction, the block will slowly drift down the slope.

This is a similar problem to the one we saw at the end of chapter 15. In that case the drifting was caused by the interaction between different contacts. In this case there is no interaction: the same thing occurs for objects with only one contact. It is therefore much easier to resolve.

The solution lies in the calculation of the relative velocity of the contact. We'd like to remove any velocity that has built up due to forces in the contact plane. This would allow the object to move into the slope in the direction of the contact normal, but not along it.

To accomplish this we add a calculation of the velocity due to acceleration to the `calculateLocalVelocity` method:

Excerpt from include/cyclone/precision.h

```
Vector3 Contact::calculateLocalVelocity(unsigned bodyIndex, real duration)
{
    RigidBody *thisBody = body[bodyIndex];

    // Work out the velocity of the contact point.
    Vector3 velocity =
        thisBody->getRotation() % relativeContactPosition[bodyIndex];
    velocity += thisBody->getVelocity();

    // Turn the velocity into contact coordinates.
    Vector3 contactVelocity = contactToWorld.transformTranspose(velocity);

    // Calculate the amount of velocity that is due to forces without
    // reactions.
    Vector3 accVelocity = thisBody->getLastFrameAcceleration() * duration;

    // Calculate the velocity in contact coordinates.
    accVelocity = contactToWorld.transformTranspose(accVelocity);

    // We ignore any component of acceleration in the contact normal
    // direction; we are only interested in planar acceleration.
    accVelocity.x = 0;

    // Add the planar velocities - if there's enough friction they will
    // be removed during velocity resolution
    contactVelocity += accVelocity;
```

```
      // And return it.
      return contactVelocity;
}
```

The code finds the acceleration and multiplies it by the duration to find the velocity introduced at the rigid-body integration step. It converts this into contact coordinates, and removes the component in the direction of the contact normal. The resulting velocity is added to the contact velocity, to be removed in the velocity resolution step, as long as there is sufficient friction to do so. If there isn't sufficient friction, then the object will slide down the slope exactly as it should.

16.1.3 INTEGRATION STABILITY

This enhancement needs some background explanation, so we'll return to the integration algorithm from chapters 3 and 10.

For both particles and rigid bodies I have used a similar integration algorithm. It calculates the linear and angular acceleration and applies these to the velocity and rotation, which are in turn applied to the position and orientation. This integration algorithm is called Newton–Euler. Newton refers to the linear component (which is based on Newton's laws of motion), and Euler refers to the angular component (Euler was a mathematician who was instrumental in our understanding of rotation).

Our integrator uses the equations

$$\dot{p}' = \dot{p} + \ddot{p}t$$

and

$$p' = p + \dot{p}t$$

(along with their rotational equivalents), each of which only depends on one level of differentiation. They are therefore termed "first-order." The overall method is more fully called "first-order Newton–Euler," or Newton–Euler 1.

Newton–Euler 2

As we saw in chapter 3, Newton–Euler 2 is an approximation. In high school physics the equation

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}t^2$$

is taught. This depends on two levels of differentiation. With the equivalent equation for angular updates we have a second-order Newton–Euler integrator.

Newton–Euler 2 is more accurate than Newton–Euler 1. It takes into account acceleration when determining the updated position. As we saw in chapter 3, the t^2 term is so small for high frame-rates that we may as well ignore the acceleration term

altogether. This is not the case when acceleration is very large, however. In this case the term may be significant, and moving to Newton–Euler 2 can be beneficial.

Runga–Kutta 4

Both Newton–Euler integrators assume that acceleration will remain constant during the entire update. As we saw in chapter 6 when we looked at springs, the way acceleration changes over the course of an update can be very significant. In fact, by assuming that acceleration does not change, we can run into dramatic instabilities and the complete breakdown of the simulation.

Springs aren't the only thing that can change acceleration quickly. Some patterns of resting contacts (particularly when a simultaneous velocity resolution algorithm is used) can have similar effects, leading to vibration or a dramatic explosion of object stacks.

For both problems a partial solution lies in working out the accelerations needed in mid-step. The fourth-order Runga–Kutta algorithm[1] (or simply RK4) does just this.

If you read around on physics engines, you'll see mention of Runga–Kutta integration. I know some developers have used it quite successfully. Personally I have never had the need. It is much slower than Newton–Euler, and the benefits are marginal. It is most useful when dealing with very stiff springs, but as we saw in chapter 6, there are simpler ways to fake the same behavior.

The biggest problem with RK4, however, is that it requires a full set of forces midway through the step. When combined with a collision resolution system this can get very messy. In our case we do not directly determine the forces due to contacts, and we do not want to run a full collision detection routine in mid-step, so RK4 is of limited use. Even for force-based engines the extra overhead of calculating mid-update forces gives a huge performance hit.

I have seen developers use RK4 for the rigid-body update and then a separate collision resolution step at the end. This could be easily implemented in our engine by replacing the `integrate` function of the rigid body. Unfortunately, with the collision resolution not taking part, RK4 loses most of its power, and I feel that the result is only useful if you have stubborn spring problems.

16.1.4 THE BENEFIT OF PESSIMISTIC COLLISION DETECTION

Our algorithm for collision resolution sometimes misses collisions altogether. Figure 16.2 shows a situation with one collision. The object shown has a low moment of inertia, so the resolution of this collision will leave the object as shown. Since there was only one collision detected, this new interpenetration cannot be resolved at this time step. The player will see the object interpenetrated until the following frame,

---

1. Unlike Newton–Euler, it is fourth-order because it takes four samples, not because it uses four levels of differentiation.
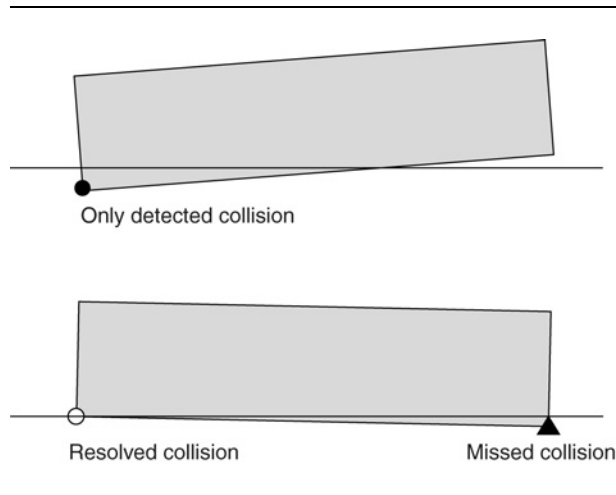
FIGURE 16.2   Collisions can be missed if they aren't initially in contact.

when it can be resolved. Single-frame interpenetration isn't normally visible, but if two or more contacts end up in a cycle, then the object can appear to be vibrating into the surface.

The only way to deal with this situation is to make collision detection more pessimistic. In other words, collision detection should return contacts that are close but not actually touching. This can be achieved by expanding the collision geometry around an object and then using an offset for the penetration value. If the collision geometry is one unit larger than the visual representation of the object, then 1 is subtracted from the penetration value of detected collisions.

In practice it is rare to see any effects of this. The times that I have needed this kind of modification (which crops up in all physics systems, regardless of the method of collision resolution), it has been most noticeable in collisions between long, light objects (such as poles) and the ground. It is a trivial change to move the ground slightly higher for collision detection and subtract a small amount from generated ground collisions.

16.1.5   Changing Mathematical Accuracy

All the mathematics in our engine is being performed with limited mathematical precision. Floating-point numbers are stored in two parts: a series of significant digits (called the "mantissa") and an exponent. This means that numbers with very different scales have very different accuracies.

For regular floating-point numbers (i.e., 32-bit on a 32-bit machine), adding 0.00001 to 1 will probably give you the correct answer; but adding 0.0001 to 10,000,000,000 will not. When you have calculations that involve numbers of very different scales, the effects can be very poor. For example, if you move an object a

small distance and it is close to the origin (i.e., its coordinates are close to zero), the object will be moved correctly. If you move the same object the same distance, but it is far away from the origin, then you may end up with no movement or too large a movement, depending on the order of your mathematical operations.

When you are using the physics engine with a broad range of masses, velocities, or positions, this can be a problem. Visually it can range from objects sinking into the ground or collisions having no effect, to suddenly disappearing bodies and collisions occurring in completely the wrong direction. It is a common problem in collision detection algorithms too, where objects can be reported as touching when they are separate, or vice versa.

There is no definitive solution to this problem, but you can increase the accuracy of the mathematics being performed. In C++ you can switch from floats to doubles, which take up twice the amount of memory, take a little less than twice the amount of time to process, but have millions of times the accuracy.

I have placed all the code on the CD that deals with the accuracy of the engine into the include/cyclone/precision.h file. This defines the real data type, which is used for all floating-point numbers. The real data type can be defined as a float or as a double. As well as the data type, I have given aliases for some mathematical functions in the standard C library. These need to be reset to call the correct precision version.

The single-precision code has been quoted so far. When compiling in double-precision mode these definitions become

<div align="center">Excerpt from include/cyclone/body.h</div>

```
#define DOUBLE_PRECISION
typedef double real;
#define REAL_MAX DBL_MAX
#define real_sqrt sqrt
#define real_abs fabs
#define real_sin sin
#define real_cos cos
#define real_exp exp
#define real_pow pow
#define R_PI 3.14159265358979
```

You can see this code in the precision header, along with an ifdef to select the definitions you need.

I tend to compile with double precision by default. On a PC the performance hit is relatively minor. On some consoles that are very strongly 32-bit, the 64-bit mathematics is very slow (they perform the mathematics in software rather than hardware, and so are much more than twice as slow in most cases), so single precision is crucial. For objects with similar masses, low velocities, and positions near the origin, single precision is perfectly fine to use. The demonstration programs on the CD work well in either precision.

16.2 Optimizations

Having stabilized the major problems out of our engine, we can turn our attention to optimization. There is a wise programming adage: Always avoid premature optimization. Nowhere is this more important than in games.

As game developers we have a pressing need for fast and efficient code, and this can spur you into optimizing code as you write. This is important to some extent, but I've seen many cases where it consumes vast quantities of programming time and ends up making negligible difference to code performance. With all code optimization it is crucial to have a profiler and check what is slow and why. Then you can focus on issues that will improve performance, rather than burning time.

The engine presented in this book has many opportunities for optimization. There are quantities that are calculated several times, data storage that is wasteful, and extraneous calculations that can be abbreviated and optimized. The version of the engine built for this book is meant to be as clear and concise as possible rather than fully optimized for performance.

At the end of this section I will look briefly at some of the areas that could be improved for speed or memory layout. I will not work through the details of the more complex ones, but will leave them as an exercise if your profiler is telling you that it would help.

There is one key optimization we can make first that has such a dramatic effect on the overall performance that it is worth looking at in detail. This isn't a code optimization (in the sense that it doesn't do the same thing in a more efficient way) but is a global optimization that reduces the physics engine's workload.

16.2.1 Sleep

There is a saying in graphics engine programming that the fastest polygons are those you don't draw. Quickly determining which objects the user can see, and then rendering only those, is a key part of rendering technology. There are dozens of common techniques used to this end (including some we've seen in this book, such as BSP trees and quad-trees).

We can't do exactly the same thing in physics: otherwise, if the player looked away and looked back, objects would be exactly as they were when last seen, even if that was mid-collapse. The equivalent optimization for physics engines is to avoid simulating objects that are stable and not moving. In fact this encompasses the majority of objects in a typical simulation. Objects will tend to settle into a stable configuration: resting on the ground or with their springs at an equilibrium point. Because of drag, only systems that have a consistent input of force will fail to settle down (the force may be gravity, however—a ball rolling down an infinitely long slope will never stop, for example).

Stopping the simulation of objects at rest is called putting them to "sleep." A powerful sleep system can improve the performance of a physics simulation by many hundreds of times, for an average game level.

There are two components to the sleep system: one algorithm to put objects to sleep and another to wake them up again. We will look at both, after putting in place some basic structure to support them.

Adding Sleep State

To support sleep we need to add three data members to the `RigidBody` class:

- `isAwake` is a boolean variable that tells us whether the body is currently asleep, and therefore whether it needs processing.
- `canSleep` is a boolean variable that tells us if the object is capable of being put to sleep. Objects that are under the constant control of the user (i.e., the user can add forces to them at any time) should probably be prevented from sleeping, for visual rather than performance reasons.
- `motion` will keep track of the current movement speed (both linear and angular) of the object. This will be crucial for deciding if the object should be put to sleep.

In the `RigidBody` class this is implemented as

```
                        ──── Excerpt from src/body.cpp ────
   class RigidBody
   {
       // ... Other RigidBody code as before ...

       /**
        * Holds the amount of motion of the body. This is a recency-
        * weighted mean that can be used to put a body to sleep.
        */
       real motion;

       /**
        * A body can be put to sleep to avoid it being updated
        * by the integration functions or affected by collisions
        * with the world.
        */
       bool isAwake;

       /**
        * Some bodies may never be allowed to fall asleep.
        * User-controlled bodies, for example, should be
        * always awake.
        */
       bool canSleep;
   };
```

When we come to perform the rigid-body update, we check the body and return without processing if it is asleep:

```
void RigidBody::integrate(real duration)
{
    if (!isAwake) return;

    // ... Remainder of the integration as before ...
}
```

The collision detector should still return contacts between objects that are asleep, as we'll see later in this section. These dormant collisions are important when one object in a stack receives a knock from an awake body.

Despite collisions being generated, when two objects are asleep, they have no velocity or rotation, so their contact velocity will be zero and they will be omitted from the velocity resolution algorithm. The same thing happens with interpenetration. This provides the speed-up in the collision response system.

We need to add a method to the RigidBody class that can change the current state of an object's isAwake member. The method looks like this:

Excerpt from src/contacts.cpp

```
void RigidBody::setAwake(const bool awake)
{
    if (awake) {
        isAwake= true;

        // Add a bit of motion to avoid it falling asleep immediately.
        motion = sleepEpsilon*2.0f;
    } else {
        isAwake = false;
        velocity.clear();
        rotation.clear();
    }
}
```

This code toggles the current value of isAwake. If the body is being put to sleep, it makes sure it has no motion: both linear and angular velocity are set to zero. This makes sure collisions (as we saw before) have no closing velocity, which improves performance for sleeping bodies in contact.

If the body is being awakened, then the motion variable is given a value. As we'll see in the next section, an object is put to sleep when the value of this motion drops below a certain threshold. If the value of this variable is below the threshold, and the

object is awakened, it will fall asleep again immediately. Giving it a value of twice the threshold prevents this and makes sure the object is awake long enough to do something interesting (presumably the setAwake method is being called so the object can be awakened to do something interesting, not to fall right back asleep).

Finally, we add functions to check whether an object is asleep and to set and check the value of canSleep. These are implemented on the CD, and none of them are complex enough to require analysis here.

Putting Objects to Sleep

The algorithm for putting objects to sleep is simple. At each frame we monitor their motion. When their motion stabilizes over several frames, and their velocity is near zero, we put them to sleep.

The "near zero" is controlled by a parameter called sleepEpsilon.[2] When the value of the motion data member drops below this threshold, the body is put to sleep:

```
if (motion < sleepEpsilon)
{
    setAwake(false);
}
```

In the code on the CD the sleep epsilon value is shared for the whole simulation. It is a global variable accessed through a pair of functions: setSleepEpsilon and get-SleepEpsilon. You can fine-tune the value by using body-specific thresholds if you like.

Setting sleep epsilon is a trial-and-error process. The collision-handling system introduces motion into objects at each frame. If you set sleep epsilon too low, objects may never fall asleep. Even if you use a resolution system that doesn't have these problems, too low a value may take a long time to reach. If you set the value too high, then objects that are obviously in motion can be sent to sleep, and that can look odd. I tend to set my sleep threshold as high as possible before strange mid-motion freezes become apparent.

The algorithm is simple, but it relies on calculating the value of motion. The motion value needs to encapsulate the linear and angular velocity of the object in a single scalar. To do this we use the total kinetic energy of the object. In chapter 3 we saw that the kinetic energy of a particle is given by

$$E_k = \frac{1}{2} m |\dot{p}|^2$$

---

2. The Greek letter epsilon is used in engineering to mean a very small quantity of any kind.

where m is the body's mass and $\dot{p}$ is its linear velocity. A similar expression holds for the kinetic energy of a rotating rigid body:

$$E_k = \frac{1}{2}\left(m|\dot{p}|^2 + i_m|\dot{\theta}|^2\right)$$

where i_m is the moment of inertia about the axis of rotation of the body (i.e., it is a scalar quantity) and $\dot{\theta}$ is its angular velocity.

We could use the kinetic energy as the value for motion, but that would create a problem with different masses: two identical objects would fall asleep at different times depending on their mass. To avoid this we remove the masses from the expression to get

$$\text{motion} = |\dot{p}|^2 + |\dot{\theta}|^2$$

In code this looks like

```
currentMotion = velocity.scalarProduct(velocity) +
                rotation.scalarProduct(rotation);
```

because the scalar product of two vectors is equivalent to their lengths multiplied together and multiplied by the cosine of the angle between them. If we take the scalar product of a vector with itself, the cosine is 1, and the result is just the square of the vector's length.

Some developers use variations on this: they either add the two components together without squaring them or calculate the full kinetic energy and then divide by the mass.

In either case this gives us a value for the motion of the object. The final stage is to check whether this value is stable. We do this by keeping a record of the current motion over several frames and seeing how much it changes. This can be neatly tracked by a recency-weighted average (RWA), one of the most useful tools in my programming repertoire.

A recency-weighted average is updated by

```
rwa = bias*rwa + (1-bias)*newValue;
```

It keeps an average of the last few values, with more recent values being more significant. The bias parameter controls how much significance is given to previous values. A bias of zero makes the RWA equal to the new value each time it is updated (i.e., there is no averaging at all). A bias of 1 ignores the new value altogether.

The RWA is an excellent device for smoothing input or for checking that an input has stabilized. In our case we have

```
motion = bias*motion + (1-bias)*currentMotion;
```

If `currentMotion` drops below the sleep epsilon value, but in the previous few frames the object has been moving a great deal, then the overall motion value will still be high. Only when an object has spent a while not moving will the recency-weighted average drop below the epsilon value.

Because objects can move at very high speeds (and because we are working with the square of these speeds), a brief burst of speed can send the RWA sky high, and it will take a long time to get back down to reasonable levels. To prevent this, and to allow objects to fall asleep faster, I have added code to limit the value of the RWA:

```
if (motion > 10*sleepEpsilon) motion = 10*sleepEpsilon;
```

The bias of the RWA should be dependent on the duration of the frame. Longer frames should allow the current value to affect the RWA more than short frames. Otherwise objects will fall asleep faster at faster frame-rates.

We can accomplish this in the same way we did for damping:

```
real bias = real_pow(baseBias, duration);
```

where `baseBias` is the bias we'd expect for one-second frames. I've typically used values around 0.5 to 0.8 here, but again some experimentation is needed.

Waking Objects Up

We have already seen that objects can be awakened manually. We also need to wake objects up when they must respond to new collisions. Collisions between sleeping objects, as we have seen, are generated and automatically ignored by the collision resolution system.

When a new object (the player, for example, or a projectile) comes along and collides with a sleeping object, we want all objects that could be affected by the collision to wake up. For any particular collision this means that if one body involved is asleep and the other is awake, then the sleeping body needs to be awakened. We add a method to the `Contact` class to accomplish this:

```
                          ─── Excerpt from src/contacts.cpp ───
void Contact::matchAwakeState()
{
    // Collisions with the world never cause a body to wake up.
    if (!body[1]) return;

    bool body0awake = body[0]->getAwake();
    bool body1awake = body[1]->getAwake();

    // Wake up only the sleeping one.
    if (body0awake ^ body1awake) {
```

```
            if (body0awake) body[1]->setAwake();
            else body[0]->setAwake();
    }
}
```

This method is called whenever we are about to resolve a collision. Collisions that occur between a sleeping object and an awake object but are not being considered (because they don't have any velocity or penetration to resolve) don't require the sleeping object to be awakened. If the contact isn't severe enough to need resolving, we can assume it isn't severe enough to wake the sleeping object.

If we have a series of collisions in a chain, as shown in figure 16.3, the first collision will be handled by waking up object B. Then the velocity update algorithm will determine that the second contact needs resolving, waking up object C, and so on. Eventually all the objects that need a velocity or position change will be awakened, as required.

The adjustPositions and adjustVelocities methods of the contact resolver have the call added just before they perform the resolution on a single contact. Here is the abbreviated code for penetration resolution:

```
for (unsigned i = 0; i < positionIterations; i++)
{
    // Find worstContact (as before) ...

    if (!worstContact) break;

    worstContact->matchAwakeState();
    worstContact->applyPositionChange();

    updatePenetrations();
}
```

There is a second situation in which we need to wake up a rigid body. That is when a force is applied to it (excluding forces that are always present, such as gravity). This can be done manually, adding a setAwake call each time a force is applied. This is difficult to remember, however, so I have elected to wake the object automatically whenever a force or torque is applied. Each of the addForce, addForceAtPoint, and addTorque functions in the RigidBody class on the CD automatically calls setAwake.

We now have a fully functional sleep system, capable of dramatically improving the performance of the engine.

Typically, when the game level is loaded, all rigid bodies are placed so that they are in their rest position. They can then all be set to sleep when the game begins. This makes the physics simulation code very fast indeed. Objects will require physical
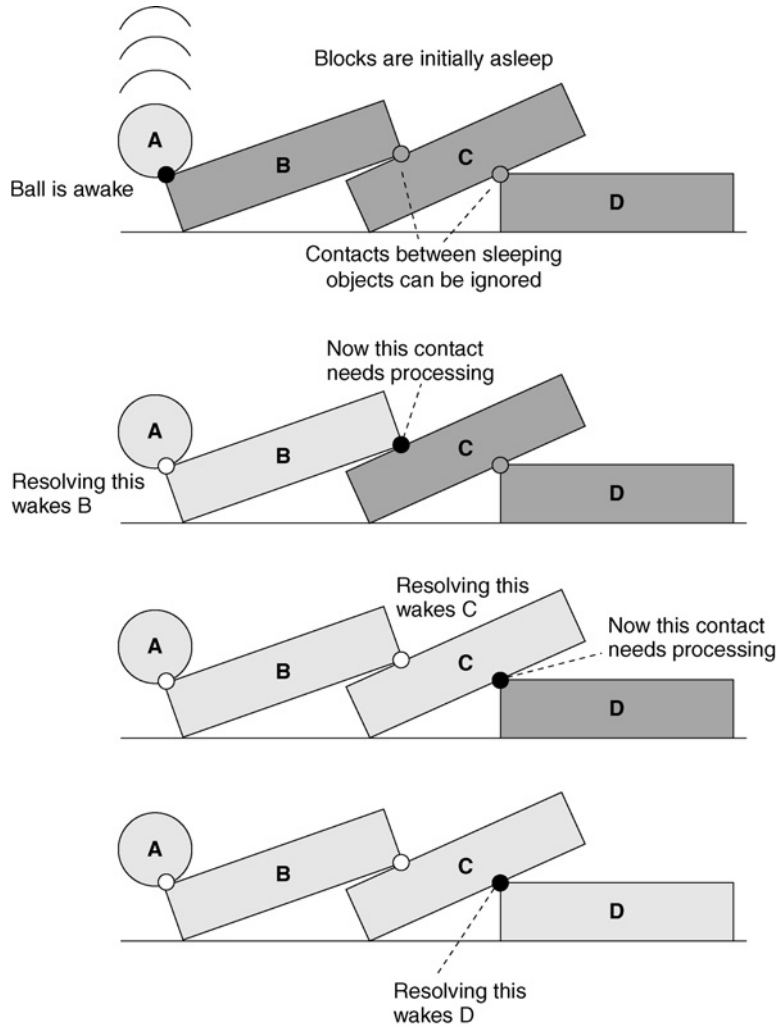
FIGURE 16.3 A chain of collisions is awakened.

simulation only once they have been collided with. Even then, so we hope, they will reach another equilibrium position and be sent back to sleep quickly.

16.2.2 MARGINS OF ERROR FOR PENETRATION AND VELOCITY

Another optimization worth making is one that speeds up the penetration and velocity resolution algorithms dramatically. Figure 16.4 shows our now familiar block-
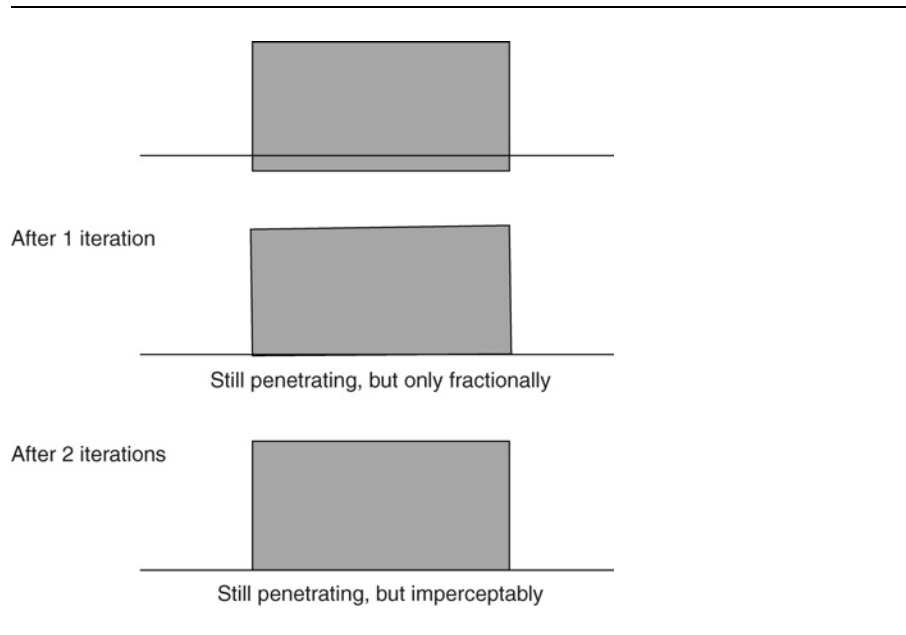
FIGURE 16.4   Iterative resolution makes microscopic changes.

on-a-plane situation. If we run this simulation and look at the resolutions being performed, we see that the two contacts (four in a 3D simulation) are repeatedly considered. Taking just penetration, if we look at the penetration depths at each iteration, we see (as shown in the figure) that the first penetration resolutions get us almost there and then subsequent resolutions make such tiny adjustments that they can never be seen by a player. This kind of sub-visual adjustment is a pure waste of time.

To avoid this situation we can add a tolerance limit to both velocity and penetration collisions. Only collisions that are more severe than this limit will be considered. That way the first time a contact is resolved, it should be brought within the limit and then never reconsidered unless the resolution of another contact disturbs it greatly.

This limit can be simply implemented when we search for the most severe contact to consider. Rather than starting with a worstPenetration value of zero

```
Contact* lastContact = contacts + numContacts;
for (unsigned i = 0; i < positionIterations; i++)
{
    Contact* worstContact = NULL;
    real worstPenetration = 0;
    for(Contact* contact = contacts; contact < lastContact; contact++)
    {
        if (contact->penetration > worstPenetration)
```

```
        {
            worstContact = contact;
            worstPenetration = contact->penetration;
        }
    }
    if (worstContact) worstContact->applyPositionChange();
    else break;
}
```

(which is the code from chapter 14), we start with a value equal to the tolerance we are allowing:

```
Contact* lastContact = contacts + numContacts;
for (unsigned i = 0; i < positionIterations; i++)
{
    Contact* worstContact = NULL;
    real worstPenetration = penetrationEpsilon;
    for(Contact* contact = contacts; contact < lastContact; contact++)
    {
        if (contact->penetration > worstPenetration)
        {
            worstContact = contact;
            worstPenetration = contact->penetration;
        }
    }
    if (worstContact) worstContact->applyPositionChange();
    else break;
}
```

The situation is similar for the velocity. Now no contact will be selected that has a penetration below this epsilon value. This value should be small enough so that the contact is not easily noticed by the player. The first time the contact is resolved, the resolution should bring the contact's penetration below this limit, so it will not be considered again. Tuning is, again, a necessity. For the demos on the CD I have used values around 0.01 for each. If your objects are larger or faster, then higher values should be used. If they are smaller or slower, then use lower values.

Both the velocityEpsilon and penetrationEpsilon values are properties of the collision resolver class. In the code on the CD I have included methods to set and get their current value.

When I added this simple change to the engine, I gained a fivefold speed-up immediately. For complex stacks of objects the improvement was even more significant.

Between the sleep system and this pair of tolerances, we have a physics simulation that is fast enough for real production work. Further optimization can be achieved in

the physics core by code manipulation and trading off memory against speed. I'll say a few things briefly about that at the end of the section.

There remains a significant performance problem with the way contacts and collisions are detected and handled, however.

16.2.3 CONTACT GROUPING

In chapter 14 I mentioned that performance could be improved by batching together groups of contacts. For our engine this provides a useful speed-up. For engines that do simultaneous resolution the speed-up can be critical.

Figure 16.5 shows a simple scene. There are several contacts in the scene, generated by the collision detector. In the collision resolution system, contacts A, B, and C can all affect one another: resolving contact A can cause problems with contact B, and resolving B can affect both A and C. Contacts D and E are likewise related. But notice that A, B, and C cannot affect D, E, or F; D and E cannot affect A, B, C, or F; and F cannot affect any of the others.

In fact two contacts can only affect each other if they are connected through a series of rigid bodies and other contacts. So contact A and C can affect each other because they are connected through bodies 2 and 3 and contact B.

Our resolution algorithm checks all possible contacts to see whether they have been affected by a previous resolution. It also checks through all contacts to find the current, most severe contact to resolve. Both these operations take longer for longer lists of contacts.
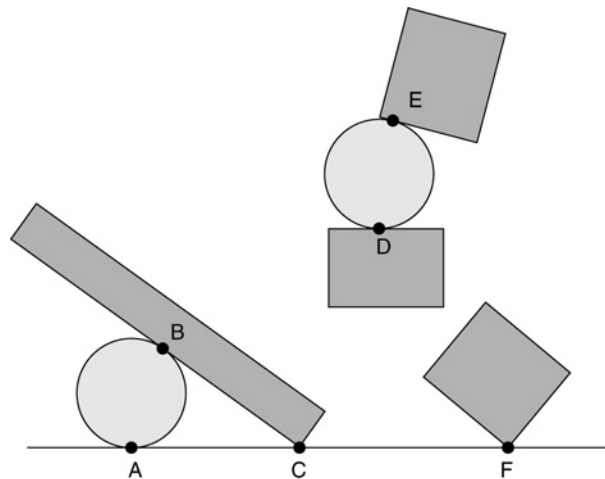
FIGURE 16.5   Sets of independent contacts.

A better solution would be to resolve the contacts in groups. Contacts A, B, and C can be sent to the resolver first; then D and E and then F. Used in this way the contact resolver would have no way of altering contacts D and E based on the results of resolving A, B, and C. But this is okay, since we know those contacts can't possibly interact.

This batching is typically done in one of two places. It can be the job of the collision detector to return groups of contacts. Or the whole list of contacts can be separated by the collision resolution system and processed in batches.

The first approach is the best, if it can be implemented. Typically the collision detector can determine if objects are a long way from one another and batch them. If it is using a coarse collision detection system, for example, it can produce contact batches for each distinct area of the game level. For sets of nearby objects that aren't touching, however, the collision detector will typically return batches that are too large (i.e., batches that contain contacts that can't affect one another). If the game level has many such situations, it can improve performance further to add a batching processor to the resolver as well as the collision detection batching.

A batching processor separates the whole list of contacts into batches. It does this by starting at one contact and following the combination of contacts and rigid bodies to find all contacts in one batch. This is then sent for processing. It then finds another contact that has not yet been placed in a batch and follows the contacts to build another batch. This repeats for as long as there are contacts that have not been processed.

Implementing a batching processor involves being able to quickly find the rigid bodies involved in each contact (we have that data already, since the contact data structure stores the rigid bodies involved) and being able to find the contacts on each rigid body. This is difficult with the current state of the engine, since rigid bodies don't keep track of the contacts that affect them. Searching through all possible contacts to find those belonging to a rigid body would take too long and defeat the object of the optimization.[3]

In chapter 14 we looked at a set of modifications to contact processing that allowed a sorted list of contacts to be retained so they didn't need to be sorted each time. In the update part of this algorithm the effect of one resolution step is propagated through the contacts. This uses the same data structure we would need to efficiently implement batching: a linked list of contacts belonging to each rigid body.

16.2.4 Code Optimizations

The final optimization phase I want to consider is code optimization. Code optimizations are tweaks that don't change the algorithm but make processing it more efficient.

---

3. In fact, while this is true of our engine, it is not necessarily true of engines with much more complex resolution algorithms. In either case, however, there is a better way.

There is a whole range of code optimizations that can be applied to the source code on the CD. I have deliberately avoided making the code more complex by trying to wring additional performance from it.

This section is intended to give some general pointers. The advice is based on the commercial engine I developed and on which `cyclone` is based. Before you embark on any optimization effort, I would strongly advise you to get a good profiler (I use Intel's VTune for PC work) and only optimize areas of the software that you can prove are causing performance problems.

Caching Contact Data

A relatively simple optimization is to retain the calculations performed during contact resolution as data in the `contact` class. When resolving one contact several times, we currently recalculate its `deltaVelocity` and other values. These can instead be stored in the contact data structure and only calculated when first needed.

This is a tradeoff of memory against speed. If you have a large number of contacts that are only likely to be considered once, then it may be better to leave the algorithm as is.

Vectorizing Mathematics

The next optimization takes advantage of the mathematical hardware on PCs and most consoles. This hardware is capable of processing more than one floating-point number at the same time. Rather than performing all our vector and matrix manipulation as a series of floating-point operations, we can have it process a whole vector at a time.

For single-precision builds of the engine (things get considerably more complex for double precision, so we'll ignore that) on a 32-bit PC, we can fit a whole vector into one of the SSE registers. Using SSE mathematics we can perform a matrix transform of a vector in only four operations. Vector cross products, additions, and other manipulations are equally accelerated. Most consoles (older hand-helds being the exception) provide the same facilities. On the Sony PlayStation 2, for example, there is a dedicated vector mathematics unit you can use for the same effect.

I'm not going to dive into detail about vectorizing mathematics. There is reasonable documentation available with Visual Studio for the Windows PC and many excellent introductions to the subject online. For serious PC development I would recommend Intel's *Software Optimization Cookbook* [Gerber, 2002] (whether or not you are targeting Intel processors).

Twizzling Rigid-Body Data

The vector mathematics hardware on PCs is optimized to run the same program on multiple bits of data at the same time. Rather than go through one algorithm per rigid body, it would be better to run the same algorithm for a group of bodies at the same time.

The rigid-body integration algorithm is a particular candidate for this. We can speed things up by having it process four objects at the same time. To do this, however, we would need to rearrange how data is held for each rigid body.

For the sake of an object-oriented programming style, we've used a class containing all the data for one rigid body. To take advantage of simultaneous processing we need to "twizzle" the data, so it is grouped together: the position for each object in an array, followed by all the velocities, and so on. This can be achieved by using a new data structure that holds four rigid bodies at a time.

Personally I have never implemented this in any physics engine I have built. Some of the AI engine development I've been involved with, however, has used this structure, with four characters being updated at once.

Grouping Data for Areas of the Level

Memory management is a crucial part of optimizing game technologies. There is plenty of memory available on most games machines for physics, but its organization can cause slow-downs.

Processors don't have equal access to all parts of the memory. They load data in chunks from the main memory and keep it in high-speed caches. Accessing data from the cache is fast. If the data needed isn't in the cache, then it has to be fetched from main memory, which can be very time consuming. Different machines have different cache sizes, and some have several levels of cache.

To avoid constantly fetching new data into the cache, data should be grouped together. For small game levels all the physics data can be easily kept together. For medium-size game levels care must be taken that the physics data isn't simply added into another data structure. For example:

```
class MyObject
{
    AIData ai;
    Geometry geometry;
    Material material;
    Texture textures[4];
    RigidBody physics;
    CollisionGeometry collisionGeom;
};

MyObject objects[256];
```

This can easily make the rigid-body data for consecutive objects a long distance apart in memory. When resolving collisions among many objects, data can quite easily need to be fetched to the cache on most resolution steps, causing disastrous performance problems.

A better solution is to keep all the sets of data together in a separate array:

```
AIData ai[256];
Geometry geometry[256];
Material material[256];
Texture textures[4][256];
RigidBody physics[256];
CollisionGeometry collisionGeom[256];
```

For large game levels this still won't be enough. In this case it is worth ordering the set of rigid bodies such that objects in different areas of the game level are kept together. That way, when contacts are processed, the bodies involved are likely to appear in the cache together. Contacts will not be generated between objects across the level from each other, so they can be separated in the array.

Cache misses are notoriously difficult to diagnose, and their prevalence tends to change dramatically when you add debugging code or make seemingly unrelated adjustments. A good profiler is essential.

16.3 SUMMARY

By simply adding sleeping objects and tolerance for near-collisions you will have a reasonably efficient physics engine. It's time now to look at how it can be used in some real game applications. If you are creating your own engine as you follow through this book, it's time to put it through its paces. If your profiler detects performance problems, you can return to this chapter and try some of the other optimizations.

Chapter 17 reviews what we have and looks at how the key physics effects seen in many recent games are achieved.

This page intentionally left blank
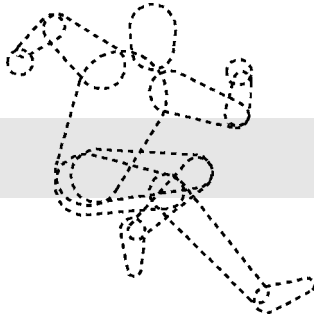
PUTTING IT ALL TOGETHER

We have built a complete physics engine that can simulate any kind of rigid body, detect collisions between objects, and realistically resolve those collisions. It is capable of running the physics for a huge range of games. Now it's time to put it through some paces.

Before we work through the demonstration applications, it is worth taking stock of where we have come from and looking at the physics engine as a whole.

17.1 OVERVIEW OF THE ENGINE

The physics engine we have built has four distinct parts:

- *The force generators (and torque generators)* examine the current state of the game and calculate what forces need to be applied to which objects.
- *The rigid-body simulator* processes the movement of rigid bodies in response to those forces.
- *The collision detector* rapidly identifies collisions and other contacts both between objects and between an object and the immovable parts of the level. The collision detector creates a set of contacts to be used by the collision resolver.
- *The collision resolver* processes a set of contacts and adjusts the motion of rigid bodies to accurately depict their effects.

Each of these components has its own internal details and complexities, but we can broadly treat them as separate units. Notice that there are two kinds of internal data used in the preceding components:

- *The forces and torques generated* are never represented in an explicit data structure. They are applied directly to the appropriate rigid body as soon as they are calculated.
- *Contacts* are generated by the collision detector and stored together, before being sent as a group to the collision resolver.

To represent objects in the game we need three kinds of data:

- *Rendering geometry and materials* are used to display the object on screen. This is normally not used at all by the physics engine, although it can take the place of the collision geometry for very simple objects.
- *Collision geometry* is a simplified set of shapes that represents an object. It is used to speed up collision detection. In some engines the collision geometry is made up of a polygonal mesh, just like the rendering geometry. In other engines objects are made up of sets of primitive shapes such as ellipsoids and boxes. In both cases a comprehensive collision detection system will typically need more than one level of collision geometry: the lowest level will be enclosed in one or more bounding volumes.
- *Rigid-body data* contains information about the physical characteristics of the object. It includes things like mass and inertia tensor, as well as position and velocity. In our engine most of this is encapsulated in the rigid body class. In addition we need to have access to contact data such as the friction between pairs of surfaces and their restitution.

These three kinds of data, along with the four parts of the engine and the two internal lines of communication, work together to provide the physics simulation. Figure 17.1 shows how the data passes through the system.

This is the basic design of most physics engines, although there are some variations. In particular it is possible to add additional components to the pipeline, such as a batch processing algorithm to divide the set of contacts into unconnected batches. Some engines also have another stage of rigid-body update at the end of the pipeline, especially if the result of the collision resolution system is a set of forces to apply.

The whole physics pipeline is typically contained within a single call in the game loop. We could easily create a black-box physics system that keeps track of everything needed to run the physics simulation. In this book, as well as in real game development, I avoid doing this. In real game development physics isn't happening for its own sake; it is just part of the whole game, and the data that the physics system needs overlaps with data needed elsewhere. A black-box system can easily duplicate work and cause a nightmare trying, for example, to make sure that all copies of an object's position are synchronized.

In a real game different objects will also need different additional data. Some objects may be active and so require data for the AI. Other objects may be player-controlled and require network data for synchronization. Any objects can require game-specific data such as hit points or value. Such complexities can make it difficult to ensure that the physics data is correctly initialized and represents realistic objects (the inertia tensor is notoriously easy to get wrong).
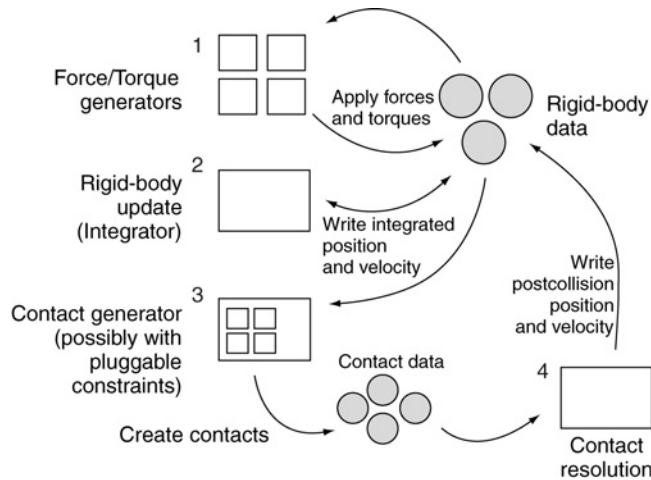
FIGURE 17.1   Data flow through the physics engine.

Setting up new objects with the correct physics can be a challenge. In my experience it is invaluable to have a simple environment set up as part of the level-design process where the physics of objects can be tested interactively. That way, as you develop you can be sure that the object feels right in its environment and that no crucial data is being left uninitialized.

17.2 USING THE PHYSICS ENGINE

We can now do almost anything we want with our physics engine. In this chapter I'll give you a taste of some of the most popular applications for physics: ragdolls, breakable objects, and movie-style explosions. On the way we'll look at some additional techniques, force generators, and ways to configure the engine.

There is one important caveat to these applications, however. If you are building your engine for a single purpose (to run off-road trucks or as part of a procedural animation system, for example), then there may be faster ways to get there directly.

I am going to focus on using our generic engine to power these effects. If all you need is a single-purpose physics system, there may be things we have put in our code that aren't needed. For example, for high-spec racing cars that don't normally leave the ground, you can omit all the rigid-body physics and build special-case spring code to model how their suspension flexes and how it handles.

Our approach is to build a physical approximation of the object and simulate it. Sometimes a better approach is to work out the desired behavior and program that in explicitly. Having said that, the general-purpose versus special-case dilemma is becoming increasingly moot. Modern games typically need several effects at once:

a driving game will model cones and fences, allowing them to break and be scattered realistically, for example. In situations where different kinds of physical behavior need to interact, there is little to substitute for a complete physics engine.

17.2.1 RAGDOLLS

Ragdolls are the hot physics application of the moment: characters that can be thrown around and generate their own realistic animation using physics. They are part of a wider move toward procedural animation: animation that doesn't need an artist creating keyframes.

A ragdoll is made up of a series of linked rigid bodies (see figure 17.2). These rigid bodies are called "bones" (they roughly correspond to the bones used in skeletal animation, although there can be a different number of ragdoll bones and rendering bones). At their most complex, ragdolls can contain dozens of bones, essential for getting a flexible spine or tail.

The bones are connected together with joints: constraints very much like those we saw in chapter 7. Finally, in some games force generators are added to the joints to simulate the way characters would move in flight: shielding their faces and trying to brace their hands against the fall.

On the CD the **ragdoll** demo omits the force generators[1] but includes the joints to keep the bones together.

The constraints are implemented as contacts. In addition to the regular contact generator, a list of joints is considered and contacts are generated to keep them to-
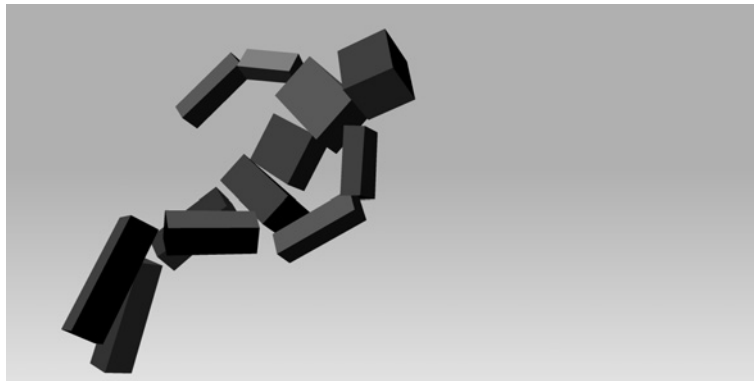
FIGURE 17.2   Screenshot of the **ragdoll** demo.

---

1. I left these out because there are some important complications in their implementation. These complications arise from the way people move: it is a problem of AI rather than of physics.

Permanent contact point
(keeps bones apart if closing
or together if separating)
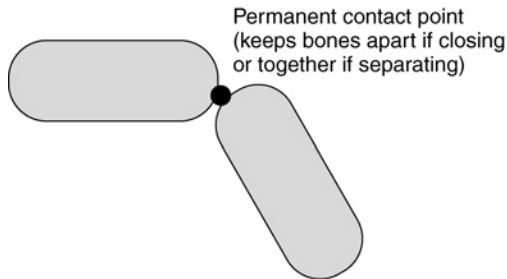
FIGURE 17.3   Closeup of a ragdoll joint.

gether. Figure 17.3 shows a detail of one such joint. Note that the contact is keeping two points together. The contact will always be between these two points, making sure they align.

To prevent the contact from slipping further out of alignment, the friction at the joint should be effectively infinite. To prevent the joint from bouncing out of alignment, the restitution should be zero.

In the code we have the following structure that holds information on one joint:

```
────────── Excerpt from include/cyclone/joints.h ──────────
/**
 * Joints link together two rigid bodies and make sure they do not
 * separate.  In a general physics engine there may be many
 * different types of joint, which reduce the number of relative
 * degrees of freedom between two objects. This joint is a common
 * position joint: each object has a location (given in
 * body coordinates) that will be kept at the same point in the
 * simulation.
 */
class Joint : public ContactGenerator
{
public:
    /**
     * Holds the two rigid bodies that are connected by this joint.
     */
    RigidBody* body[2];

    /**
     * Holds the relative location of the connection for each
     * body, given in local coordinates.
     */
    Vector3 position[2];
```

```
    /**
     * Holds the maximum displacement at the joint before the
     * joint is considered to be violated. This is normally a
     * small, epsilon value.  It can be larger, however, in which
     * case the joint will behave as if an inelastic cable joined
     * the bodies at their joint locations.
     */
    real error;

    /**
     * Generates the contacts required to restore the joint if it
     * has been violated.
     */
    unsigned addContact(Contact *contact, unsigned limit) const;
};
```

Within this class there is a checkJoint method that generates contacts based on the
current configuration of the joint. In this way it acts very much like a collision detec-
tor: looking at the state of rigid bodies and generating contacts accordingly.

In the demo the joints are considered in order during the physics update:

—— **Excerpt from src/demos/ragdoll/ragdoll.cpp** ——

```cpp
void RagdollDemo::generateContacts()
{
    // Create the ground plane data.
    cyclone::CollisionPlane plane;
    plane.direction = cyclone::Vector3(0,1,0);
    plane.offset = 0;

    // Perform exhaustive collision detection on the ground plane.
    cyclone::Matrix4 transform, otherTransform;
    cyclone::Vector3 position, otherPosition;
    for (Bone *bone = bones; bone < bones+NUM_BONES; bone++)
    {
        // Check for collisions with the ground plane.
                if (!cData.hasMoreContacts()) return;
        cyclone::CollisionDetector::boxAndHalfSpace(*bone, plane, &cData);
    }

    // Check for joint violation.
    for (cyclone::Joint *joint = joints;
        joint < joints+NUM_JOINTS; joint++)
    {
                if (!cData.hasMoreContacts()) return;
```

```
        unsigned added =
            joint->addContact(cData.contacts, cData.contactsLeft);
        cData.addContacts(added);
    }
}
```

When run, this is a fast and effective ragdoll model. It isn't the most stable method, however. For very large ragdolls a lot of interpenetration resolution iterations are needed to keep the extremities from wandering too far from their correct place.

More Complex Joints

The approach of the **ragdoll** demo is about as simple as possible to get useful joints. Joints are a common feature of physics engines, and they can be considerably more complex. In particular joints are used to remove the freedom of one object to move relative to another.

The joints we have used (called "ball-joints") take away the freedom of one object to move linearly with respect to another. There are also joints that restrict movement even more: hinges that restrict the ability of one object to rotate with respect to another and piston joints that allow relative movement in one direction only.

Implementing these more flexible joints in the engine we have built is, quite frankly, inconvenient. What I have done here, trying to represent joints in terms of contacts, works for ball-joints but becomes very difficult for other kinds of joints.

In creating joints for this kind of engine I have followed approximately the same algorithm as that used for contacts (which are effectively joints that limit the motion of two objects from overlapping), but used different sets of tests to determine the adjustments needed. A hinge joint, for example, needs to check how twisted the two objects are and implement interpenetration-like resolution to bring them back into alignment.

Force-based engines with simultaneous resolution of contacts normally use a mathematical structure that makes it very easy to create a huge range of joints with minimal additional implementation effort. In the next chapter we'll look at the algorithms that support this. If you are going to make a lot of use of joints and need something more comprehensive than the simple contact-based joints in this section, it may be worth biting the bullet and upgrading your contact resolution scheme. For the sake of efficiency, ease of implementation, and programmer sanity, however, it is worth giving the simple approach a try.

17.2.2 Fracture Physics

If ragdoll physics is the current hot physics application, then fracture physics isn't far behind. Particularly in shooters, players want to see objects destroyed in a realistic way: wood should splinter, glass should shatter, and falling crates should crack to reveal their contents.
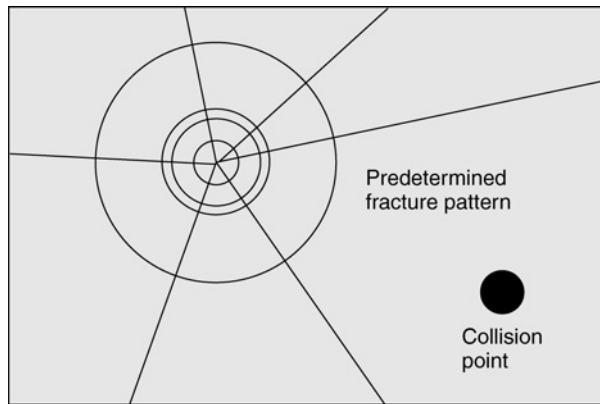
FIGURE 17.4 Pre-created fractures can look very strange for large objects.

Fracture physics can be as simple or as complex as you want it to be. Early implementations used two sets of rigid bodies: one for the whole object and another for its components. The whole rigid body has a breaking strain value: the maximum impulse it can suffer before being destroyed. During the velocity phase of the resolution algorithm the impulse applied to the object is compared against its breaking strain. If the impulse is too great, the whole rigid body is replaced by its component objects.

This is a very quick, efficient, and easy-to-implement fracture physics. Unfortunately two identical objects will always be destroyed in exactly the same way, and the pattern of destruction will not bear any relationship to the location of the impulse. Figure 17.4 illustrates this problem on a glass window.

This can be mitigated to some extent by using several possible decompositions for an object and determining which to use when the fracture is initiated. Players are good at spotting patterns, however, and most developers want a more flexible approach.

More complex fracture physics uses the same basic principle of breaking strains, but adds two, more complex algorithms. The first is a geometric algorithm to construct the components of the fractured object on-the-fly. The decomposition method depends on the type of material.

Decomposing wood needs long, splintered components; glass cracks into panes; safety glass shatters into smalls nuts; and so on. Typically this is achieved either by decomposing the object into different-sized tetrahedrons and keeping groups of these together, or by using a set of fracture patterns, and 3D boolean operations to separate components. The specifics of this decomposition are highly geometric and depend on algorithms beyond the scope of this book. The Schneider and Eberly [2002] book, in the same series as this, has a comprehensive set of algorithms for manipulating geometry.
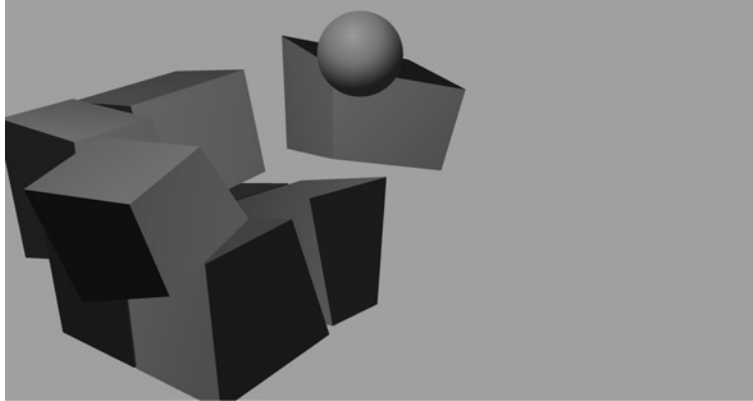
FIGURE 17.5   Screenshot of the **fracture** demo.

The second, more complex algorithm is for assigning correct physical characteristics to the component objects. In particular, assigning a correct inertia tensor for a general fractured shape is a nontrivial process. Appendix A gives formulae and algorithms for calculating the inertia tensor of various regular objects. For a general shape, however, these are complex and can be inefficient. Most developers opt for a simplification and approximate shattered pieces with geometry that has easy inertia tensors: boxes are a firm favorite.

Figure 17.5 shows the **fracture** demo on the CD. It contains a single large block, made of a relatively dense, brittle material, such as concrete. You can move to aim and fire a ball at the block. The block will shatter on impact, depending on where the ball strikes. The decomposition scheme splits the block into eight components, dividing it in each direction. The collision point is used as the center of two collisions, and the other direction is split in half, as shown in figure 17.6. To make the results look more realistic the splits are angled randomly.

The geometric division algorithm looks like this:

```
                    Excerpt from src/demos/fracture/fracture.cpp
/**
 * Performs the division of the given block into four, writing the
 * eight new blocks into the given blocks array. The blocks array can be
 * a pointer to the same location as the target pointer: since the
 * original block is always deleted, this effectively reuses its storage.
 * The algorithm is structured to allow this reuse.
 */
void divideBlock(const cyclone::Contact& contact,
    Block* target, Block* blocks)
{
```
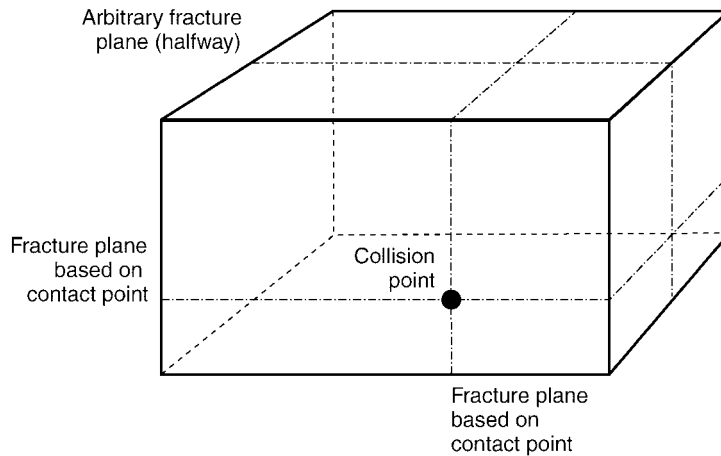
FIGURE 17.6 The fractures of a concrete block.

```
// Find out if we're block one or two in the contact structure, and
// therefore what the contact normal is.
cyclone::Vector3 normal = contact.contactNormal;
cyclone::RigidBody *body = contact.body[0];
if (body != target->body)
{
    normal.invert();
    body = contact.body[1];
}

// Work out where on the body (in body coordinates) the contact is
// and its direction.
cyclone::Vector3 point =
    body->getPointInLocalSpace(contact.contactPoint);
normal = body->getDirectionInLocalSpace(normal);

// Work out the center of the split: this is the point coordinates
// for each of the axes perpendicular to the normal, and 0 for the
// axis along the normal.
point = point - normal * (point * normal);

// Take a copy of the half size, so we can create the new blocks.
cyclone::Vector3 size = target->halfSize;

// Take a copy also of the body's other data.
```

```
cyclone::RigidBody tempBody;
tempBody.setPosition(body->getPosition());
tempBody.setOrientation(body->getOrientation());
tempBody.setVelocity(body->getVelocity());
tempBody.setRotation(body->getRotation());
tempBody.setLinearDamping(body->getLinearDamping());
tempBody.setAngularDamping(body->getAngularDamping());
tempBody.setInverseInertiaTensor(body->getInverseInertiaTensor());
tempBody.calculateDerivedData();

// Remove the old block.
target->exists = false;

// Work out the inverse density of the old block.
cyclone::real invDensity =
    halfSize.magnitude()*8 * body->getInverseMass();

// Now split the block into eight.
for (unsigned i = 0; i < 8; i++)
{
    // Find the minimum and maximum extents of the new block
    // in old-block coordinates.
    cyclone::Vector3 min, max;
    if ((i & 1) == 0) {
        min.x = -size.x;
        max.x = point.x;
    } else {
        min.x = point.x;
        max.x = size.x;
    }
    if ((i & 2) == 0) {
        min.y = -size.y;
        max.y = point.y;
    } else {
        min.y = point.y;
        max.y = size.y;
    }
    if ((i & 4) == 0) {
        min.z = -size.z;
        max.z = point.z;
    } else {
        min.z = point.z;
        max.z = size.z;
    }
```

```
            // Get the origin and half size of the block, in old-body
            // local coordinates.
            cyclone::Vector3 halfSize = (max - min) * 0.5f;
            cyclone::Vector3 newPos = halfSize + min;

            // Convert the origin to world coordinates.
            newPos = tempBody.getPointInWorldSpace(newPos);

            // Set the body's properties (we assume the block has a body
            // already that we're going to overwrite).
            blocks[i].body->setPosition(newPos);
            blocks[i].body->setVelocity(tempBody.getVelocity());
            blocks[i].body->setOrientation(tempBody.getOrientation());
            blocks[i].body->setRotation(tempBody.getRotation());
            blocks[i].body->setLinearDamping(tempBody.getLinearDamping());
            blocks[i].body->setAngularDamping(tempBody.getAngularDamping());
            blocks[i].offset = cyclone::Matrix4();
            blocks[i].exists = true;

            // Finally calculate the mass and inertia tensor of the new block.
            blocks[i].calculateMassProperties(invDensity);
        }
}
```

This assumes that the collision will occur on the YZ plane of the block (which it must in our demo). More complete code would have similar algorithms for the other possible collision planes.

Because the resulting pieces are roughly rectangular, they are treated like rectangular blocks for calculating their inertia tensors. This is done simply as

```
                    ── Excerpt from src/demos/fracture/fracture.cpp ──
/**
 * Calculates and sets the mass and inertia tensor of this block,
 * assuming it has the given constant density.
 */
void calculateMassProperties(cyclone::real invDensity)
{
    // Check for infinite mass.
    if (invDensity <= 0)
    {
        // Just set zeros for both mass and inertia tensor.
        body->setInverseMass(0);
        body->setInverseInertiaTensor(cyclone::Matrix3());
```

```
    }
    else
    {
        // Otherwise we need to calculate the mass.
        cyclone::real volume = halfSize.magnitude() * 2.0;
        cyclone::real mass = volume / invDensity;

        body->setMass(mass);

        // And calculate the inertia tensor from the mass and size.
        mass *= 0.333f;
        cyclone::Matrix3 tensor;
        tensor.setInertiaTensorCoeffs(
            mass * halfSize.y*halfSize.y + halfSize.z*halfSize.z,
            mass * halfSize.y*halfSize.x + halfSize.z*halfSize.z,
            mass * halfSize.y*halfSize.x + halfSize.z*halfSize.y
            );
        body->setInertiaTensor(tensor);
    }

}
```

Creating a general-purpose fracture physics system involves more geometric processing than physics knowledge. Some developers have gone this route, and there are a couple of middleware vendors with similar technologies. But to trap all useful scenarios is a moderately long task—certainly as long as the contact resolution or rigid-body algorithms we have created.

17.2.3 Explosive Physics

Explosions have been around from the earliest days of gaming and were the application of the first physics engines: particle engines creating smoke and debris. Explosions are a whole lot more fun with proper physics; there's something gratifying about watching debris scattered around the level.

Explosions are easy to create with a custom force generator. We could create a force generator that simply imparts a force to objects near the blast point. This would send objects cascading, but ultimately would be dull to look at. It has two problems. First, the explosion effect is quite monotonous: objects just fly out. Second, applying forces alone doesn't cause objects to spin.

A movie-quality explosion effect has three components: an initial implosion, an all-around explosion, with an expanding concussion wave, and a convection chimney. Each of these components has a slightly different behavior.

Implosion

When the explosion first occurs, the heat in the explosion consumes the oxygen in a ball around the explosion point and can ionize the air (this is what causes the flash). A sudden, dramatic drop in pressure results, and nearby air rushes into the gap. This is the implosion stage, and it is the same process that occurs in a lightning strike.

There is a military technology called "thermobaric weapons" that does its damage in this way, using very high temperatures to cause a huge pressure change and a powerful compression wave (see the next section) that can destroy buildings and devastate life.

In a real explosion of modest size this effect is barely noticeable, and can even be completely lost in the concussion phase. For games and movies, however, it looks good and gives the explosion an added sense of power. A longer concussion, particularly when associated with a geometry-stretching graphical effect, can add tension to the explosion and even suggest some kind of alien technology.

The implosion stage of the force generator applies a force to all objects within some threshold radius, in the direction of the point of explosion. We'll put all three stages of the explosion into one force generator. So far it looks like this:

```
                    ┌─ Excerpt from src/demos/fracture/fracture.cpp ─┐
/**
 * A force generator showing a three-component explosion effect.
 * This force generator is intended to represent a single
 * explosion effect for multiple rigid bodies. The force generator
 * can also act as a particle force generator.
 */
class Explosion : public ForceGenerator,
                  public ParticleForceGenerator
{
    /**
     * Tracks how long the explosion has been in operation, used
     * for time-sensitive effects.
     */
    real timePassed;

public:
    // Properties of the explosion: these are public because
    // there are so many and providing a suitable constructor
    // would be cumbersome.

    /**
     * The location of the detonation of the weapon.
     */
    Vector3 detonation;
```

```
        /**
         * The radius up to which objects implode in the first stage
         * of the explosion.
         */
        real implosionMaxRadius;

        /**
         * The radius within which objects don't feel the implosion
         * force. Objects near to the detonation aren't sucked in by
         * the air implosion.
         */
        real implosionMinRadius;

        /**
         * The length of time that objects spend imploding before the
         * concussion phase kicks in.
         */
        real implosionDuration;

        /**
         * The maximal force that the implosion can apply. This should
         * be relatively small to avoid the implosion pulling objects
         * through the detonation point and out the other side before
         * the concussion wave kicks in.
         */
        real implosionForce;

    public:
        /**
         * Creates a new explosion with sensible default values.
         */
        Explosion();

        /**
         * Calculates and applies the force that the explosion
         * has on the given rigid body.
         */
        virtual void updateForce(RigidBody * body, real duration);

        /**
         * Calculates and applies the force that the explosion has
         * on the given particle.
         */
```

```
            virtual void updateForce(Particle *particle, real duration) = 0;

    };
```

The implosion can only impose a linear force. Because it is so short, we don't need to set objects spinning.

Concussion Wave

The concussion wave (also called the "shockwave") is initiated by the initial implosion: air rushes into the vacuum, creating an expanding wavefront. This may be combined, near the explosion site, with shrapnel and munition fuel expanding from the weapon. For very high-temperature devices the wavefront may comprise burning air, known as a "fireball" (characteristic in atomic and nuclear devices).

The concussion wave throws objects outward from the explosion. In movies and games it is responsible for cars flying through the air and characters being knocked off their feet.

The characteristic of a concussion wave is its propagation. It spreads out from the point of explosion, getting weaker as it goes. Like a surfer always on the outside edge of a water wave, light objects can ride the outside edge of the concussion wave and accelerate to very high speeds. But like a surfer who doesn't catch the wave, most objects will receive an initial boost at the wave boundary, and then will behave normally when inside the wavefront.

We can implement this in our force generator by applying forces to objects within an expanding interval from the blast point. The interval should be wide enough so that no objects are missed. Its width depends on the frame-rate and the speed of the wavefront, according to the formula

$$w \geqslant \frac{s}{\text{fps}}$$

where s is the speed of the wavefront, w is the width of the interval, and fps is the number of frames per second. In other words, w is the distance the wave travels in one frame. In practice, objects on either side of this peak should also get some force, but to a lesser extent. The force equation

$$f_a = \begin{cases} f_b(1 - (st - d)/kw) & \text{when } st - kw \leqslant d < st \\ f_b & \text{when } st \leqslant d < st + w \\ f_b(d - st - w)/kw & \text{when } st + w \leqslant d < st + (k+1)w \\ 0 & \text{otherwise} \end{cases}$$

has proved useful for me. In it st is the position of the back of the wavefront (i.e., the speed times the time), k is the width of the tail-off on either side of the wave, d is the
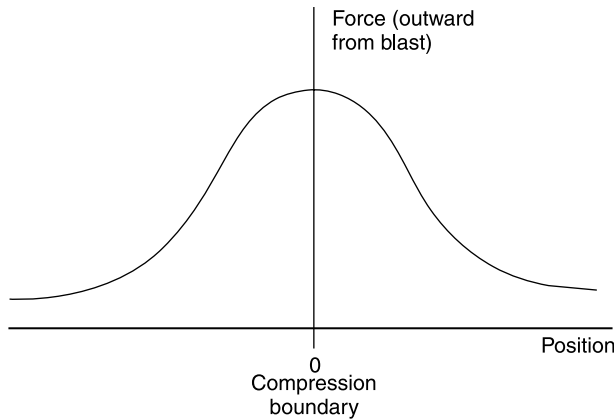
FIGURE 17.7   The cross section of force across a compression wave.

distance of an object from the center of the blast, f_a is the applied force, and f_b is the peak blast force, which we'll calculate in a moment. The equation simply provides a linear fall-off of force on either side of the wave. The force cross section is shown in figure 17.7. Note that the force is always acting outward from the center of the blast.

We need to calculate the peak force for this equation. The force applied to an object depends on both its aerodynamic drag (since the compression wave is primarily a moving-air effect) and its current velocity. We could do this by simply using the aerodynamic effects from chapter 11, but if you aren't using that already, it is probably overkill.

We can approximate the force effect by applying a force that depends on the difference between the object's velocity and the wavefront. To get exploding objects to spin as they are moved, we apply the force off-center. This can be as simple as selecting a random, off-center point for each object when the force generator is created. The same point should be used from frame to frame to prevent objects from looking like they are jiggling in mid-air. It also means that once the point is pushed so it is in line with the force vector, the object stops rotating. Otherwise objects could rotate faster and faster under the influence of the explosion, and that looks odd.

With the concussion wave implemented the explosion force generator looks like this:

```
━━━━━━━━━━ Excerpt from include/cyclone/fgen.h ━━━━━━━━━━
/**
 * A force generator showing a three-component explosion effect.
 * This force generator is intended to represent a single
 * explosion effect for multiple rigid bodies. The force generator
 * can also act as a particle force generator.
 */
```

```
class Explosion : public ForceGenerator,
                  public ParticleForceGenerator
{
    /**
     * Tracks how long the explosion has been in operation, used
     * for time-sensitive effects.
     */
    real timePassed;

public:
    // ... Other Explosion code as before ...

};
```

Convection Chimney

The final part of the explosion is another Hollywood exaggeration of a real explosion. As well as the pressure effects from the initial explosion, the heat generated will set up a convection current above the blast point. In most conventional weapons this is a minor effect and isn't very noticeable. It is significant and iconic in atomic and nuclear weapons, however; the mushroom cloud has become a potent indicator of explosive violence. While it should be used sparingly (big mushroom clouds after a grenade goes off look peculiar), it can be a great way to indicate a superior weapon.

Convection chimneys provide a very small amount of upward force for a long time after the explosion. It is not enough to lift anything but the lightest objects off the ground. Because light objects are unlikely to be around the blast point after the concussion wave, developers typically introduce extra particles that only respond to the convection. These particles are light enough to be carried upward.

The convection chimney has an equation similar to that of the concussion wave, but it doesn't move outward. The linear fall-off works fine:

$$f_a = \begin{cases} f_b d_{xz}/w & \text{when } d_{xz} < w \text{ and } d_y < h \\ 0 & \text{otherwise} \end{cases}$$

where w is the width of the chimney, h is the maximum height of the chimney, d_{xz} is the distance of the object from the blast center, in the XZ plane only (because we want the chimney to be a cylinder shape), and d_y is the height of the object above the blast point.

The force should again be applied in a line from the blast center. If we apply the force in just the up direction, then objects will rise up the chimney and bob at the top. If the force is angled out, the characteristic mushroom cloud shape is formed. The peak force is calculated in the same way as for the concussion wave: it is another moving-air phenomenon. The code to produce this effect looks like this:

```
                        ─── Excerpt from include/cyclone/fgen.h ───
    /**
     * A force generator showing a three-component explosion effect.
     * This force generator is intended to represent a single
     * explosion effect for multiple rigid bodies. The force generator
     * can also act as a particle force generator.
     */
    class Explosion : public ForceGenerator,
                      public ParticleForceGenerator
    {
        /**
         * Tracks how long the explosion has been in operation, used
         * for time-sensitive effects.
         */
        real timePassed;

    public:
        // ... Other Explosion code as before ...

        /**
         * This is the peak force for stationary objects in
         * the center of the convection chimney. Force calculations
         * for this value are the same as for peakConcussionForce.
         */
        real peakConvectionForce;

        /**
         * The radius of the chimney cylinder in the xz plane.
         */
        real chimneyRadius;

        /**
         * The maximum height of the chimney.
         */
        real chimneyHeight;

        /**
         * The length of time the convection chimney is active. Typically
         * this is the longest effect to be in operation, as the heat
         * from the explosion outlives the shockwave and implosion
         * itself.
         */
        real convectionDuration;
    };
```
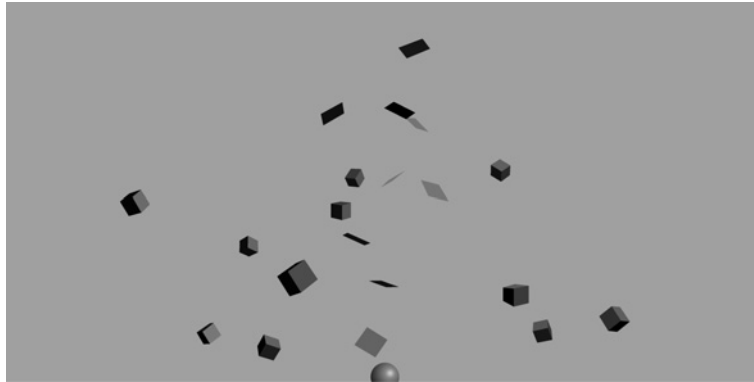
FIGURE 17.8   Screenshot of the **explosion** demo.

All together the explosion looks quite good. The **explosion** demo on the CD shows the three components in action (see figure 17.8). There are no lighting or fire particle effects, which would normally be used in a real game explosion (neither of which is typically driven by the physics of the explosion). For a huge explosion a neat effect is to set fire to (i.e., add fire particles to the surface of) objects as they first come into the range of the concussion wave. This gives the effect of a consuming fireball.

17.3 LIMITATIONS OF THE ENGINE

We have built a usable physics engine and had some fun putting it through its paces in different game situations. This is about as far as we'll be going in detailed code. The rest of the book looks in a more general way at other issues and approaches.

As I've said from the beginning, the approach we've taken is a sound one, with a good blend of implementation ease and simulation power. Ultimately, however, any approach has limitations. While I have tried to be clear about the limitations as we have gone along, before looking at the benefits of other approaches, it is worth recapping those issues that our engine finds difficult to handle.

17.3.1 STACKS

Large stacks of objects may not be too stable in our engine. Of course we can set the stack up and put it to sleep, and have it fall when knocked, but a slight touch is likely to set it jiggling. At its worst it can cause blocks at the top of the stack to move visibly and vibrate their way off the edge.

This is caused by the iterative penetration resolution algorithm. The algorithm doesn't perfectly position objects after resolving the resolution. For one object (or

even a small number of stacked objects) this isn't a problem. For large stacks the errors can mount until at the top they are very noticeable.

Judicious use of putting objects to sleep means that stacks can be made to appear stable. If you need a brick wall to be blown apart, this is a good strategy and won't show the engine's limits.

17.3.2 Reaction Force Friction

As we saw in the last chapter, reaction force friction is honored when a contact is being resolved, but not when a contact is moved as a side effect of another resolution. This makes it difficult for one movable object leaning against another to stay in place. The objects will appear to slide off each other, regardless of the frictions imposed. The best that can be hoped for is that the sleep system kicks in to stop them from sliding apart.

This is another side effect of the interpenetration resolution algorithm: it doesn't honor the friction of one contact when considering the penetration resolution of another.

17.3.3 Joint Assemblies

The same cumulative errors that cause stacks to become unstable can also lead to noticeable artifacts when long strings of rigid bodies are connected by joints. In addition to making a full range of joints a burden to program, our engine considers each joint in series. Joints at one end of a chain can be dramatically affected by adjustments at the other end.

This can be as mild as a slight stretching of some of the joints, through a slowdown in the processing (where all the available iterations are used up), to vibration, and at the most extreme, catastrophic failure.

Iterative resolution isn't the best option for highly constrained assemblies of rigid bodies (although it can cope with modest groupings like our ragdoll): if this is your primary application, then it's best to go for a simultaneous solver.

17.3.4 Stiff Springs

Finally we get to the bugbear from the second part of the book. Stiff springs are as much a problem for our full rigid-body engine as they were for the particle engine, and for exactly the same reason. While it is possible to use faked force generators, as we did in chapter 6, the problem can't be entirely solved.

17.4 Summary

Almost anything that can be done with a game physics engine can be done with the physics engine we've built in this book. As we've seen in this chapter, it can be used to run all the "hot" applications for physics in games.

But no physics engine is perfect. We've built a system that is very fast indeed, but we've sacrificed some accuracy, particularly when it comes to how contacts are resolved.

In the final chapter of this book we'll look at other ways to approach building a physics engine. You can use these either as inspiration for building your second physics engine, or to extend the engine we've built with some extra features.

PART VI

What Comes Next?

This page intentionally left blank

18

OTHER TYPES
OF PHYSICS

We've developed our physics engine as far as we're going to. There are things you can add to it: more force generators, joints, and so on. You can use it in a wide variety of game genres as long as you understand its limitations and are willing to work around them.

I've referred to this chapter more than any other in this book. As we've built the engine up, I've made decisions about approximations, assumptions, and implementation options that I would use. In each case there were other alternatives. The engine I've built is good and useful, but there are a couple of other approaches that would have been equally good and would have had a different set of limitations.

This chapter looks at the main differences between our physics engine and those other approaches. It will not give a step-by-step guide to building those engines or any detailed implementation advice. Because building an engine involves a whole series of interdependent decisions, this chapter would be twice the length of the book if we worked through each approach. Instead I hope it will give you enough information to understand the alternatives and to get you started if you want to go that way too.

18.1 SIMULTANEOUS CONTACT RESOLUTION

In our engine we resolve contacts one at a time. This is fast, but as we've seen it has limitations. In particular we have no way of knowing whether the action we take to resolve one contact might cause other contacts to move in an unrealistic way. In our engine this is seen when a set of connected contacts with friction appear to slide against one another.

The alternative is to resolve a set of contacts at the same time. Rather than calculating the impulse of each in turn, we need to find the impulses of all simultaneously.

Most physics engines that perform this simultaneous calculation are based on force calculations rather than impulses. In other words, two objects in resting contact are kept apart by a constant force, not by a series of single-frame impulses as we have done. So the resolution calculation tries to find the forces and impulses to apply at each contact, taking the interaction of all contacts into account.

The most common approach to doing this is called the "linear-complementary problem." It involves building a mathematical structure called a Jacobian, which encodes the interactions between different contacts. This can then (usually) be turned into a single set of forces to apply.

Because this is such a common and important approach, we'll look at it from a high level in this chapter. I won't go into the finer points of implementation, however, because getting the algorithms to work in a stable way involves numerous special-case problems and unusual complications.

18.1.1   The Jacobian

The Jacobian is a mathematical construct that says how one contact affects another. It is used to determine the right balance of adjustments to make with the full knowledge of the side effects of any tweak. The Jacobian is a matrix and may be of any size.

All the forces and torques for all objects are combined into one very long vector. There will be three force entries and three torque entries for each rigid body, so the vector will have $6n$ entries, where n is the number of rigid bodies. In the same way all the accelerations (linear and angular) for all objects are treated as one long vector (again having $6n$ entries).

The entries in the Jacobian matrix relate the two together. The value of row a, column b in the matrix tells us the amount of acceleration of component a that a unit of force or torque in direction b would cause. Some of the entries in the matrix are very simple: they are the equations we've used throughout the book to determine the movement of an object. For example, a force in the X direction causes an acceleration of magnitude m^{-1} (from $F = ma$); so in the Jacobian the value that relates X-direction force to X-direction acceleration will be m^{-1}.

While many of the values in the Jacobian are based on the simple laws of motion, some are due to the interaction of objects at contact points. Each value in the matrix gives the change that will occur in the row's component given a unit change in the column's component.

Calculating the entries in the Jacobian involves working out the forces at each contact given a unit force at each other contact. The process is similar to what we used in our engine to calculate the effects of one contact resolution on others.

Entries in the Jacobian don't only exist because one contact affects another. It is also possible for one axis of one contact to affect another. For a contact with friction, the friction force generated will depend on the normal reaction force. As the reac-

tion force increases in one direction, the friction force will also increase. There will therefore be an entry in the Jacobian to represent this connection.

The flexibility of the Jacobian to represent interactions between objects, as well as the basic motion of the object itself, allows it to be used to create a much wider range of joints. In our engine joints are explicitly specified and handled with their own code. The Jacobian provides a mechanism to link the movement of any two objects in the simulation. In fact it can link different elements of each object, so for example the motion of one object along one axis can be fixed (i.e., any force that tries to break this joint is resisted by an equal and opposite reaction force). In addition motors can be implemented by adding elements to the Jacobian that generate forces regardless of anything else going on. If you look at a physics engine such as ODE (and several commercial middleware packages), they allow very flexible joints and motors to be created by adding entries directly into the Jacobian.

Most force components will not directly interact with one another at all, so the Jacobian will have zeros at the corresponding locations. In fact most of the matrix will be filled with zeros. The Jacobian is sometimes called a "sparse matrix" for this reason. The mathematics of sparse matrices can be dramatically simpler than for regular matrices, but the algorithms are often more complex.

You may come across books or papers on game physics that talk about Lagrange multipliers, the Lagrange method, or Featherstone's algorithm. Each of these is related to the method shown here. The Lagrange method works with a more complex equation than ours, where the Jacobian is decomposed into several parts, one of which specifies the connections between objects and another (the so-called Lagrange multipliers) specifies the amount of interaction.[1] Most game physics engines use the raw Jacobian as shown, but you may find it useful to read up on the Lagrange method. A lot of textbooks on the mathematics of physics were not written with games in mind, so they use the Lagrange formulation extensively. It can be useful to understand how it works.

18.1.2   THE LINEAR COMPLEMENTARY PROBLEM

Armed with the Jacobian we can formulate the mathematical problem of resolving all contacts at the same time. It has the basic form of

$$Jf = \ddot{p}$$

where f is a vector of force and torque components for all rigid bodies and $\ddot{p}$ is the resulting accelerations. But f is made up of two components,

$$f = f_{\text{contacts}} + f_{\text{known}}$$

---

1. Note that the Lagrange formulation isn't just an expanded Jacobian. With all the bits extracted, it can be used in other ways and with variations on the equation I have introduced in this chapter. But all this is well beyond the scope of this book, and I've never needed it and have never invested the time to understand it in depth. It is also not for the mathematically faint-hearted.

where f_{known} is the set of forces and torques we know we are applying (forces due to gravity or due to other force generators) and f_{contacts} is the set of forces that are generated in the contacts, which is what we're trying to find out.

Most often you see the equation written as

$$Jf_{\text{contacts}} + \ddot{p}_{\text{known}} = \ddot{p} \qquad [18.1]$$

(although it is often given with different symbols such as $Jf + b = a$). In other words, the Jacobian is multiplied by the known force vector to get a known acceleration. This relies on a fact of matrix multiplication that I haven't explicitly stated before—namely, it is distributive. For any valid matrix multiplication, $A \times (B + C) = A \times B + A \times C$.

Calculating $\ddot{p}_{\text{known}}$ is a step before contact resolution, because this value will not change as we try to work out the contact forces.

On its own, equation 18.1 could be solved by working out the inverse of the Jacobian (a time-consuming problem and one often without a solution). But to make things worse we have an additional constraint:

$$0 \geqslant f_{\text{contacts}} \geqslant r$$

where r is a vector of limits on how big the forces can be. Normal reaction forces can be as large as they need to be, but friction forces are limited. A particular entry in the force vector may represent a friction force and will therefore need to be limited.

The final calculation, finding f so that it fulfills both equations, is called the "linear complementary problem" (or LCP for short). An alternative approach tries to find the smallest possible values for the components in f contacts. This becomes an optimization problem called "quadratic programming" (or QP). Some physics systems build and solve the QP, but it is more common to work with the LCP.

A commonly used algorithm for solving the LCP is called the "pivot algorithm." It works by making guesses for components in f and checking the results. The errors from one set of guesses can be used to modify components one at a time and converge at a solution. Under some assumptions that are commonly met in rigid body simulations, the guesses will always converge toward the solution. The pivot algorithm (based on an algorithm called the "Lemke pivot") was popularized in rigid-body simulation by David Baraff: see Baraff and Witkin [1997] for a step-by-step introduction to the approach.

Complications arise because of numerical instability and the approximation of fixed-length time steps. It is possible (and not uncommon) for a rigid-body simulation to end up in a physically impossible situation where there is no solution to the LCP. In this case the pivot algorithm can loop endlessly. A robust implementation needs to take account of these kinds of problems and provide alternatives. A common alternative is to impose impulses when there is no valid force solution. But getting it right while remaining efficient can be very difficult. In my experience (I have created two pivot-based engines, one of some significant complexity) it is easy to get something working, but it takes months to end up with a general and robust engine.

How It Is Used

The force-based pivot algorithm works in a different way from the engine we've been building in this book. In our engine, forces are accumulated and applied, and then collision detection and response occurs.

The Jacobian includes the calculations for applying forces, however, so everything is done in one go: contact forces are calculated and applied along with the rest of the forces.

There are three problems this raises, however: When do we do collision detection? How do we handle interpenetration? What about non-resting contacts?

The architecture of different physics engines handles these steps in different ways. The second two problems are often approached in much the same way as we have them in our engine (but removing micro-collisions: if two objects are determined to be in resting contact, then they are handled by the force system). Separate collision and interpenetration steps are taken after the forces have been applied.

In some engines the collision response is performed first, and its results are embedded in the known force vector and incorporated into the force calculations discussed before. This relies on the fact that if we know the length of time for one update frame t, then we can convert an impulse g into a force f using the formula

$$f = gt$$

allowing us to represent the calculated collision impulse as if it were just another force applied to the rigid body.

The first problem is more sticky: when do we perform collision detection? If we perform collision detection at the start of the frame, before the force calculations, then the result of applying the forces may cause new collisions that will still be visible when the objects are next drawn.

If we perform collision detection after the force calculation, then we can remove all interpenetration before the user sees the frame. But how do we determine the contacts we need to fill in the Jacobian at the start of the frame?

We could do both, but that would be very time consuming. In effect the second solution is what is normally used.

The collision detection is performed before interpenetration is resolved, and the user sees non-interpenetrating bodies. The same collision data is then stored until the next update, and it is used to fill the Jacobian. This affords a good compromise between efficiency (extra data is stored between frames) and removing visible interpenetration (which is typically very obvious to the viewer).

As I mentioned in chapter 12, commercial systems often improve efficiency further by using frame coherence: keeping track of the last frame's collisions to speed up collision checks at this frame. If the collision detector does this, then the data is already being stored and can be made available to the physics engine.

18.2 REDUCED COORDINATE APPROACHES

Another technique often mentioned in game development circles (although very rarely implemented) is the reduced coordinate approach.

In our physics engine we've given each rigid body twelve degrees of freedom: three each for position, orientation, velocity, and rotation. The orientation uses four values but has only three degrees of freedom: the fourth value can always be determined from the other three (because the size of the quaternion must be 1).

When objects are in contact, or have joints between them, they are constrained. They can no longer have any value for each of the twelve degrees of freedom. For example, if an unliftable block is placed on the flat ground, it can only be pushed in two directions and can only be oriented along one axis. It has only six actual degrees of freedom (two for position, one for orientation, two for velocity, and one for rotation).

The physics system we've built in this book allows all objects to move with twelve degrees of freedom; then it uses impulses and non-penetration code to make sure they behave properly. An alternative is to work out exactly how many degrees of freedom the object has and allow only those to change.

For the block on the ground this is simple, and an example is fully worked through in Eberly [2004], the other physics book in this series.

It involves working out the equations of motion, using Newton's laws of motion, in terms of the degrees of freedom that are left. For anything beyond a block on a plane this can become quite involved. When the constraints represent joints, then degrees of freedom can be a combination of rotation and position. Finding the equations of motion can be very difficult indeed. Once the equations of motion are calculated, they can often be solved very rapidly. It is therefore a useful approach when the degrees of freedom don't change, and the equations can be hard-coded beforehand and solved quickly (as is the case for the example in Eberly [2004]).

For some assemblies of bodies connected by joints (such as a ragdoll where there is a branching tree of bones and joints with no loops) there are well-known methods for calculating the degrees of freedom and the corresponding equations of motion. There have been a couple of single-purpose ragdoll simulators that I've seen using this approach, but they have been in technical demos rather than in production games. For general sets of joints the procedure is tougher, and to all intents and purposes impractical, especially since in a game the constraints may appear and disappear at different times (particularly the case with contact constraints).

This technique is also more difficult when it comes to introducing general force generators. Having the ragdoll float on water or be buffeted by wind, for example, introduces major complications into calculating the equations of motion for each degree of freedom.

For this reason I'm not aware of any general-purpose game physics engines that are based on reduced coordinate approaches. You may like to look at reduced coordinate approaches to get specific effects, but they are unlikely to be a general solution.

18.3 Summary

The purpose of this whirlwind tour of other approaches is to give you a basic vocabulary and understanding. When looking through the bewildering array of physics and simulation resources available on the Internet, you should now be able to understand how they fit into the whole picture and how our engine relates to them.

There are a couple of open source physics systems on the Net that you can compare with the one we've built. ODE (the Open Dynamics Engine) in particular is widely used in hobby projects. It is a solid implementation, but finding your way around how the code works can be difficult. Like our engine, it is not primarily built for speed.

For a more comprehensive mathematical survey of different physics techniques that are useful in games, I'd recommend Eberly [2004]. David assumes more mathematical knowledge than I have here, but if you've followed this book through, you're probably ready to get started. The Eberly book doesn't cover how to build a complete general-purpose engine but looks at a massive range of techniques and tricks that can be used on their own or incorporated into our engine.

David Baraff has done more than anyone in disseminating information on Jacobian-based approaches to physics. His work is on the desk of many physics developers I know. A good introduction to his work is Baraff and Witkin [1997].

This page intentionally left blank

A P P E N D I X $\mathbf{A}$

COMMON INERTIA TENSORS

This appendix provides formulae for calculating the inertia tensor of a range of physical primitives. This can be used to generate an inertia tensor for almost any game object.

Inertia tensors are discussed in chapter 10.

A.1   DISCRETE MASSES

The inertia tensor of any set of masses, connected together to form a rigid body, is

$$
I = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix} \qquad \text{[A.1]}
$$

where

$$
I_a = \sum_{i=1}^{n} m_i a_{p_i}^2
$$

and

$$
I_{ab} = \sum_{i=1}^{n} m_i a_{p_i} b_{p_i}
$$

431

In each case a_{p_i} is the distance of particle i from the center of mass of the whole structure, in the direction of axis a; m_i is the mass of particle i; and there are n particles in the set.

A.2 CONTINUOUS MASSES

We can do the same for a general rigid body by splitting it into infinitesimal masses. This requires an integral over the whole body, which is considerably more difficult than the rest of the mathematics in this book. The formula is included here for completeness.

$$I = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix} \tag{A.2}$$

as before, where

$$I_a = \int_m a_{p_i}^2 \, dm$$

and

$$I_{ab} = \int_m a_{p_i} b_{p_i} \, dm$$

The components of both are as before. The integrals are definite integrals over the whole mass of the rigid body.

A.3 COMMON SHAPES

This section gives some inertia tensors of common objects.

A.3.1 CUBOID

This includes any rectangular six-sided object, where the object has constant density:

$$I = \begin{bmatrix} \frac{1}{12}m(d_y^2 + d_z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d_x^2 + d_z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d_x^2 + d_y^2) \end{bmatrix}$$

where m is the mass and d_x, d_y, and d_z are the extents of the cuboid along each axis.

A.3.2 SPHERE

This inertia tensor corresponds to a sphere with constant density:

$$I = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$$

where m is the mass and r is the radius of the sphere.

The same sphere that is just a shell (i.e., has all its mass around the surface of the sphere) has the following inertia tensor:

$$I = \begin{bmatrix} \frac{2}{3}mr^2 & 0 & 0 \\ 0 & \frac{2}{3}mr^2 & 0 \\ 0 & 0 & \frac{2}{3}mr^2 \end{bmatrix}$$

A.3.3 CYLINDER

A cylinder, of uniform density, whose principal axis is along the Z axis, has an inertia tensor of

$$I = \begin{bmatrix} \frac{1}{12}mh^2 + \frac{1}{4}mr^2 & 0 & 0 \\ 0 & \frac{1}{12}mh^2 + \frac{1}{4}mr^2 & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix}$$

where m is the mass, r is the radius of the cylinder, and h is its height.

A.3.4 CONE

A cone, of uniform density, whose principal axis is along the Z axis, has an inertia tensor of

$$I = \begin{bmatrix} \frac{3}{80}mh^2 + \frac{3}{20}mr^2 & 0 & 0 \\ 0 & \frac{3}{80}mh^2 + \frac{3}{20}mr^2 & 0 \\ 0 & 0 & \frac{3}{10}mr^2 \end{bmatrix}$$

where m is the mass, r is the radius of the cone, and h is its height. Unlike the other shapes, the cone's center of mass isn't through its geometric center. Assuming the center of the base of the cone is at the origin, the center of mass is at

$$\begin{bmatrix} 0 \\ 0 \\ \frac{1}{4}h \end{bmatrix}$$

where h is the height of the cone, as before.

USEFUL FRICTION COEFFICIENTS FOR GAMES

This appendix provides a table of useful static and dynamic friction values for materials used in games. Both static and dynamic values are given for completeness. If you are using only one friction coefficient, then you can average these, or use the dynamic value for both.

Friction is discussed in chapter 15.

Materials	Static Friction	Dynamic Friction
Wooden crate on concrete	0.5	0.4
Wooden crate on ice	0.2	0.1
Glass on ice	0.1	0.03
Glass on glass	0.95	0.4
Metal on metal	0.6	0.4
Lubricated metal on metal	0.1	0.05
Rubber on concrete	1.0	0.8
Wet rubber on concrete	0.7	0.5
Performance tire on concrete	c.1.5	1.0
Velcro on velcro	6.0	4.0

APPENDIX **C**

OTHER PROGRAMMING LANGUAGES

This appendix gives notes on converting the engine source code into other programming languages.

C.1 C

Most of the source code can be translated into C fairly easily. All methods in classes are replaced by functions, and the overloaded operators used to represent vector and matrix operations are replaced by the regular function version.

The tricky part about using C is the force generators. Polymorphism used in this book is a convenient way to create force generators without having to understand their properties.

You can do a similar thing in C, however, by using a fixed function signature of the form:

```
(void)(*forceGenerator)(void* inData,
                        RigidBody* inOutBody);
```

where RigidBody is a typedef of a structure.

The additional data parameter is used to pass data to the force generator function (in C++ this isn't needed because the implicit `this` pointer contains the generator's data).

C.2   Java

Although Java has dramatically improved in terms of its just-in-time compilation efficiency, it is still usually better to implement highly localized, time-critical parts of the code in C++.

If you decide to go ahead with implementing the engine in Java, then the key thing to be aware of is the overhead of objects. If you have each vector in your simulation as a separate instance, then on some Java platforms this can cause a huge bloat in the amount of memory required.

It is often considerably faster, therefore, to expand some of the references to vectors and quaternions into their constituent fields. This has the downside of making the mathematics much less modular and the whole physics engine more complex.

In Java it doesn't make sense to have force and torque generators as classes. They should be interfaces, and classes such as `GravityForceGenerator` can implement them. This also allows a single class to be both a force and torque generator, if required.

C.3   Common Language Runtime (.NET)

The .NET languages (and other languages that target the common language runtime) have many of the same properties as Java. As with Java, an ideal solution would be to use a C++-coded physics engine as a service, calling it from a DLL, for example.

Even if your chosen .NET language is C++, then it is more efficient to have the engine running in unmanaged code and call it from your managed code.

If you implement using a .NET language, then be aware of similar just-in-time overheads as for Java.

C.4   Lua

Lua makes an excellent language for implementing game logic and anything beyond the low-level routines used to run the game.

At the risk of sounding like a stuck record, the physics can be one of these low-level routines implemented in C++ and called when needed from Lua.

Another option I've used, however, is to mix Lua into the physics engine. It is relatively easy to expose Lua code as a force or torque generator in the physics engine (this is particularly useful in my experience to create controllers for player-characters).

To set this up, create a `LuaForceGenerator` in C++ that can call Lua code. The `RigidBody` class will needs its `addForce` and `addTorque` methods exposed via a table to Lua so the code can then affect the rigid body when it has completed its calculation.

Lua is remarkably quick, easily fast enough to be called a few times each frame in this way.

MATHEMATICS
SUMMARY

This appendix summarizes the mathematics used in the book. It serves as a quick look-up for the appropriate formulae and equations needed when implementing physics.

D.1 VECTORS

A vector $\boldsymbol{a}$ multiplied by a scalar k:

$$k\boldsymbol{a} = k \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} kx \\ ky \\ kz \end{bmatrix}$$

Vector addition:

$$\boldsymbol{a} + \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{bmatrix}$$

and subtraction:

$$\boldsymbol{a} - \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}$$

Vectors can be multiplied in several ways. The component product has no geometric correlate:

$$\boldsymbol{a} \circ \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \circ \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x \\ a_y b_y \\ a_z b_z \end{bmatrix}$$

and the symbol shown is a personal convention.

The scalar product

$$\boldsymbol{a} \cdot \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \cdot \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = a_x b_x + a_y b_y + a_z b_z$$

has the trigonometric form

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_x b_x + a_y b_y + a_z b_z = |\boldsymbol{a}||\boldsymbol{b}| \cos \theta$$

where θ is the angle between the two vectors.

The vector product

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

has the trigonometric form

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}||\boldsymbol{b}| \sin \theta$$

and is non-commutative:

$$\boldsymbol{a} \times \boldsymbol{b} = -\boldsymbol{b} \times \boldsymbol{a}$$

D.2 QUATERNIONS

A quaternion

$$\begin{bmatrix} \cos \frac{\theta}{2} \\ x \sin \frac{\theta}{2} \\ y \sin \frac{\theta}{2} \\ z \sin \frac{\theta}{2} \end{bmatrix}$$

represents an orientation of θ about the axis:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Two quaternions can be multiplied together:

$$\begin{bmatrix} w_1 \\ x_1 \\ y_1 \\ z_1 \end{bmatrix} \begin{bmatrix} w_2 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\ w_1x_2 + x_1w_2 - y_1z_2 - z_1y_2 \\ w_1y_2 - x_1z_2 + y_1w_2 - z_1x_2 \\ w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2 \end{bmatrix}$$

A quaternion representing orientation can be adjusted by a vector representing amount of rotation according to

$$\hat{\theta}' = \hat{\theta} + \frac{1}{2}\hat{\Delta\theta}\hat{\theta}$$

where the rotation is converted into a quaternion according to

$$[\Delta\theta_x \Delta\theta_y \Delta\theta_z] \rightarrow [0\Delta\theta_x \Delta\theta_y \Delta\theta_z]$$

D.3 MATRICES

An $n \times m$ matrix has n rows and m columns.

Matrices can be post-multiplied (we don't use pre-multiplication in this book) by vectors with the same number of elements as the matrix has columns:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Matrices can be multiplied together, providing that the number of columns in the first matrix is the same as the number of rows in the second:

$$C_{(i,j)} = \sum_k A_{(i,k)} B_{(k,j)}$$

where $C_{(i,j)}$ is the entry in matrix C at the ith row and jth column; and where k ranges up to the number of columns in the first matrix.

A 3×3 matrix

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

has its inverse given by

$$M^{-1} = \frac{1}{\det M} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix}$$

where $\det M$ is the determinant of the matrix:

$$\det M = aei + dhc + gbf - ahf - gec - dbi$$

A quaternion

$$\hat{\theta} = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

represents the same rotation as the matrix

$$\Theta = \begin{bmatrix} 1 - (2y^2 + 2z^2) & 2xy + 2zw & 2xz - 2yw \\ 2xy - 2zw & 1 - (2x^2 + 2z^2) & 2yz + 2xw \\ 2xz + 2yw & 2yz - 2xw & 1 - (2x^2 + 2y^2) \end{bmatrix}$$

A transformation matrix M_t can be changed into a new coordinate system, using a transform M_b to the new coordinate system according to

$$M'_t = M_b M_t M_b^{-1}$$

D.4 INTEGRATION

To update an object's position

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}\,t^2$$

is normally replaced by the less accurate

$$p' = p + \dot{p}t$$

Velocity is updated with

$$\dot{p}' = \dot{p} + \ddot{p}t$$

Orientation is updated with

$$\hat{\theta}' = \hat{\theta} + \frac{\delta t}{2}\hat{\omega}\hat{\theta}$$

where $\hat{\omega}$ is the quaternion form of the angular velocity and t is the duration to update by.

Angular velocity is updated exactly the same as linear velocity:

$$\dot{\theta}' = \dot{\theta} + \ddot{\theta}t$$

D.5 Physics

Newton's second law of motion gives us

$$f = ma = m\ddot{p}$$

where m is the mass and p is the position of an object. Or

$$\ddot{p} = m^{-1}f$$

in terms of position.

Euler's equivalent for rotation is

$$\ddot{\theta} = I^{-1}\tau$$

where I is the inertia tensor and τ is the torque.

The force of gravity is

$$f = mg$$

where g is around 10 m/s^2 on earth, but is often replaced by 20 m/s^2 for added speed in games.

Forces through an object's center of mass can be combined using D'Alembert's principle:

$$f = \sum_i f_i$$

Forces not through the center of mass also have a torque component:

$$\tau = p_f \times f$$

where p_f is the position of application of the force, relative to the center of mass.

D'Alembert's principle also applies to rotations:

$$\tau = \sum_i \tau_i$$

which include the torques generated from off-center forces.

The separating velocity of two colliding objects v_s is related to their velocity immediately before the collision v_c by

$$v_s = -cv_c$$

where c is the coefficient of restitution.

D.6 OTHER FORMULAE

Hook's law relates the force of a spring f to its length:

$$f = -k\big(|d| - l_0\big)\widehat{d}$$

where d is the vector from one end of the spring to another.

Simple fluid flow can be modeled with an aerodynamic tensor:

$$f_a = Av_w$$

where f_a is the resulting force, A is the aerodynamic tensor, and v_w is the velocity of the air.

This page intentionally left blank

BIBLIOGRAPHY

David Baraff and Andrew Witkin [1997]. *Physically Based Modeling: Principles and Practice*. SIGGRAPH.

David Eberly [2003]. Conversion of Left-Handed Coordinates to Right-Handed Coordinates. Available *www.geometrictools.com/Documentation*.

David Eberly [2004]. *Physics for Games*. San Francisco: Morgan Kaufmann.

Christer Ericson [2005]. *Real-Time Collision Detection*. San Francisco: Morgan Kaufmann.

Richard Gerber [2002]. *The Software Optimization Cookbook*. Santa Clara, CA: Intel Press.

Roger A. Horn and Charles R. Johnson [1990]. *Matrix Analysis*. Cambridge: Cambridge University Press.

Roger A. Horn and Charles R. Johnson [1994]. *Topics in Matrix Analysis*. Cambridge: Cambridge University Press.

Philip Schneider and David H. Eberly [2002]. *Geometric Tools for Computer Graphics*. San Francisco: Morgan Kaufmann.

Gino van den Bergen [2003]. *Collision Detection in Interactive 3D Environments*. San Francisco: Morgan Kaufmann.

This page intentionally left blank

Index

SOFTWARE LICENSE

IMPORTANT: PLEASE READ THE FOLLOWING AGREEMENT CAREFULLY. BY COPYING, INSTALLING OR OTHERWISE USING THIS SOURCE CODE, YOU ARE DEEMED TO HAVE AGREED TO THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT.

1. This LICENSE AGREEMENT is between the IPR VENTURES, having an office at 2(B) King Edward Road, Bromsgrove, B61 8SR, United Kingdom ("IPRV"), and the Individual or Organization ("Licensee") accessing and otherwise using the accompanying software ("CYCLONE") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, IPRV hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use CYCLONE alone or in any derivative version, provided, however, that IPRVs License Agreement is retained in CYCLONE, alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates CYCLONE or any part thereof, and wants to make the derivative work available to the public as provided herein, then Licensee hereby agrees to indicate in any such work the nature of the modifications made to CYCLONE.

4. IPRV is making CYCLONE available to Licensee on an "AS IS" basis. IPRV MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, IPRV MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF CYCLONE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. IPRV SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING CYCLONE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by and interpreted in all respects by the law of England, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between IPRV and Licensee. This License Agreement does not grant permission to use IPRV trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using CYCLONE, Licensee agrees to be bound by the terms and conditions of this License Agreement.